INF113: Simple File System

Kirill Simonov

31.10.2025

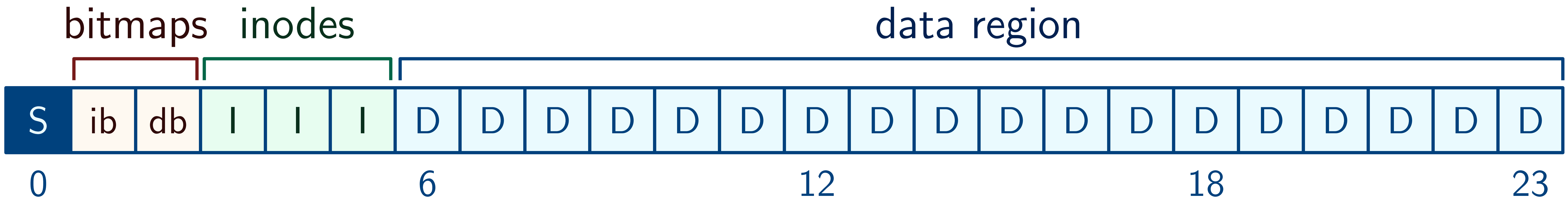
Implementing a file system

- We went through the various operations that the file system provides
- **Next:** How to implement a file system?
- Low frequences, different devices—file system is purely in software
  - Hence, a myriad of file systems with different trade-offs
- **Main questions:**
  - What kind of data structure to use for keeping track of files?
  - How to provide `open`/`read`/`write`/...?
- **Today:** A "minimal" example of a file system, `vsfs`
- **Next week:** More real file systems, including journaling (e.g., `ext3`)

APFS
...
Ext3
Ext4
...
FAT
HFS
HFS+
...
NTFS
...
UFS
XFS
ZFS

VSFS: Overall organization

- The disk is split into 4KB blocks
- **Data region:** Most of the blocks are used to store files
  - Only one file owns each block
- **Inode table:** Array of inodes
  - Multiple inode entries per block
- **Allocation structures:** Inode bitmap and data bitmap
  - Track which inodes and data blocks are allocated
- **Superblock:** File system information—type, size, block segments

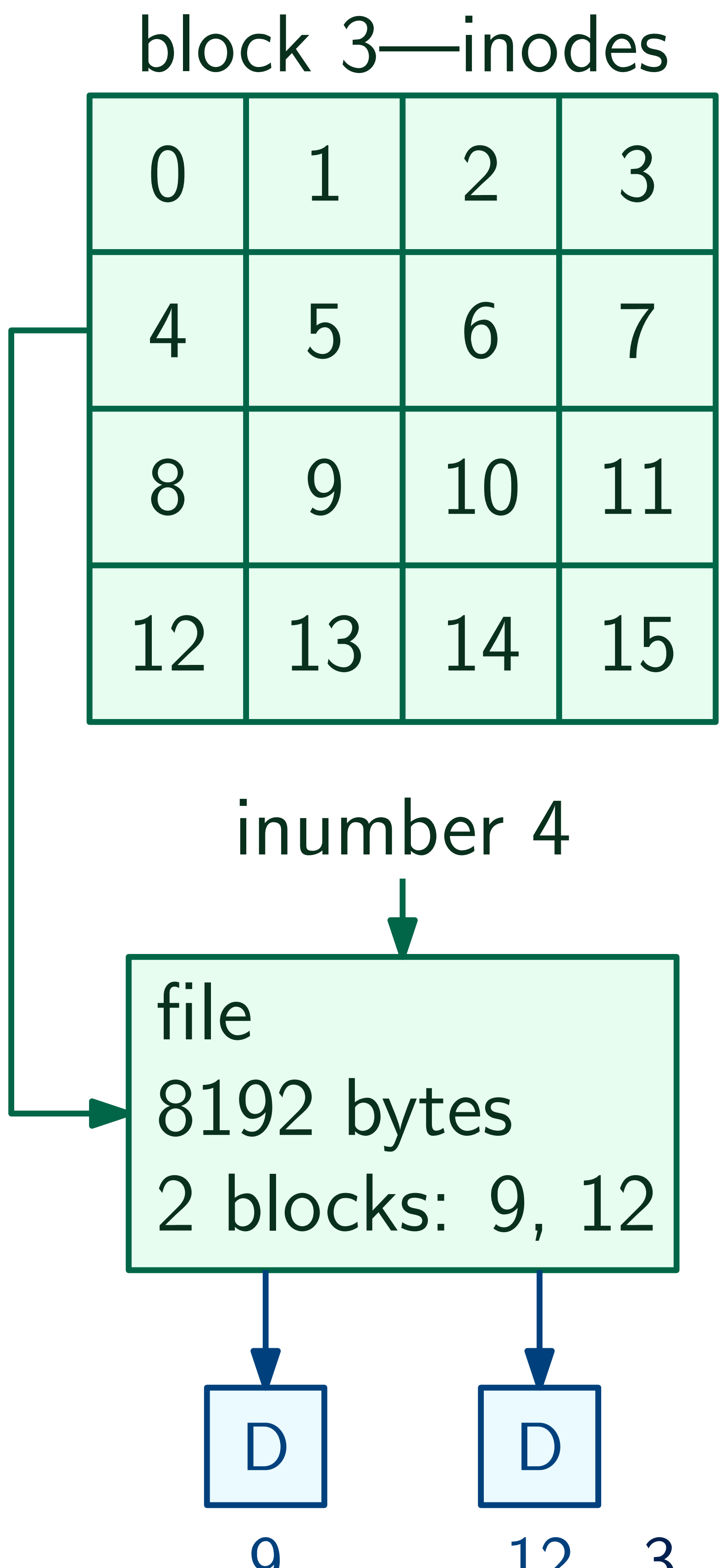


Inode

- For inodes of size 256B, each block contains 16 inodes
- An allocated inode stores information about a file:
  - size
  - pointers to data blocks
  - other metadata
- By knowing the inode number of a file, we can access inode contents

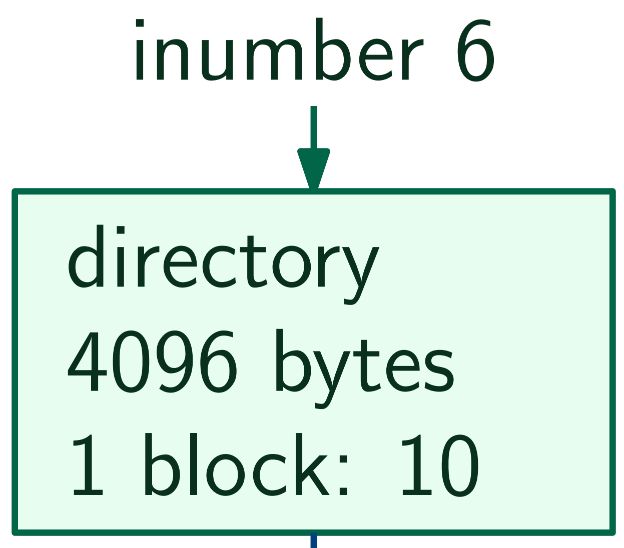
```
block = (inumber * sizeof(inode_t)) / blockSize;
```

- From inode contents, we get where the parts of a file are located



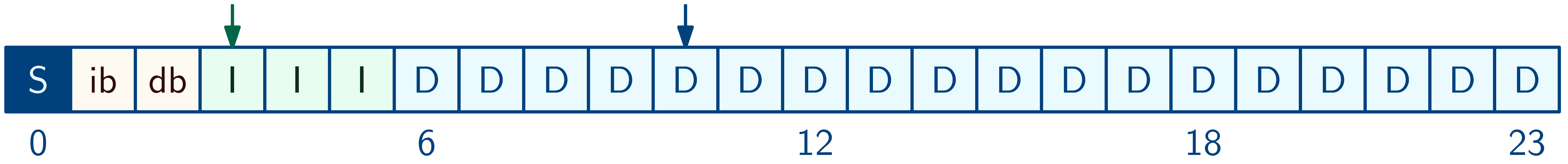
Directories

- Directory is stored as a file: has its own inode and data blocks
- Contents of the directory: list of (inode, filename) pairs
- Finding file by name: iterate over entries
- Once we find inode number of a file, we can access it
- Most directories are small
- Deleting a file: mark record as free



data block 10

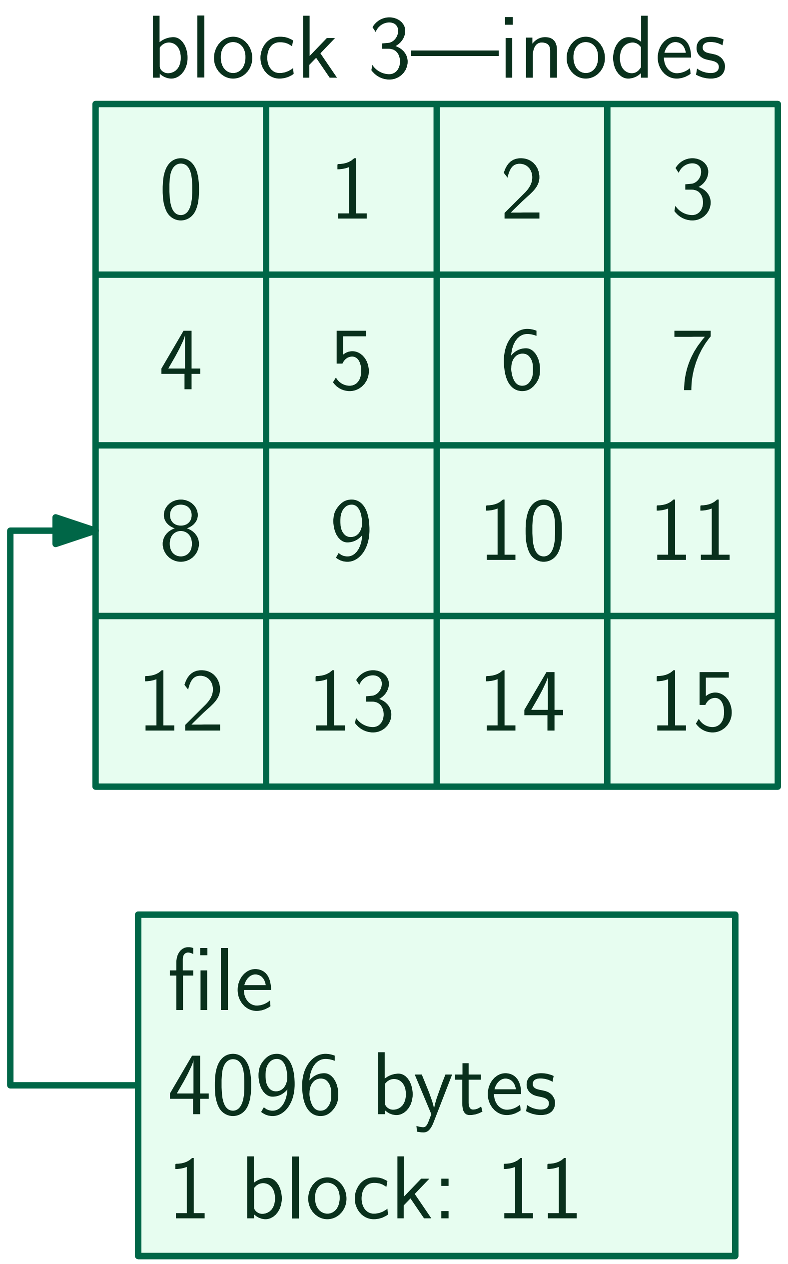
inum	reclen	strlen	name
6	12	2	.
2	12	3	..
-	12	-	-
7	12	3	bar

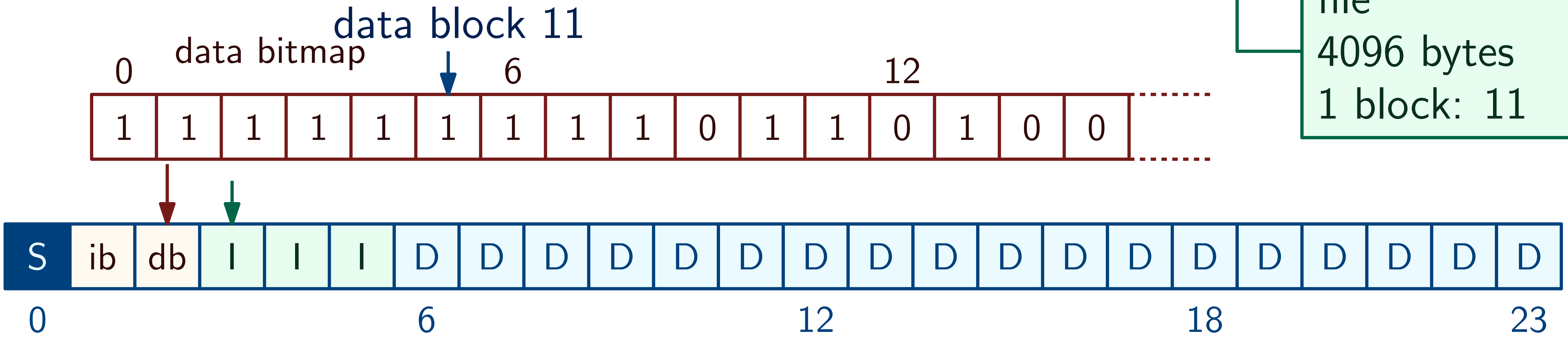


Free space management

Creating a file:

1. Iterate over inode bitmap to find an empty inode number
2. Set the inode number as taken
3. Write the new inode
4. Iterate over data bitmap to find an empty data block
5. Set the data block as taken
6. Write the data block address in the inode

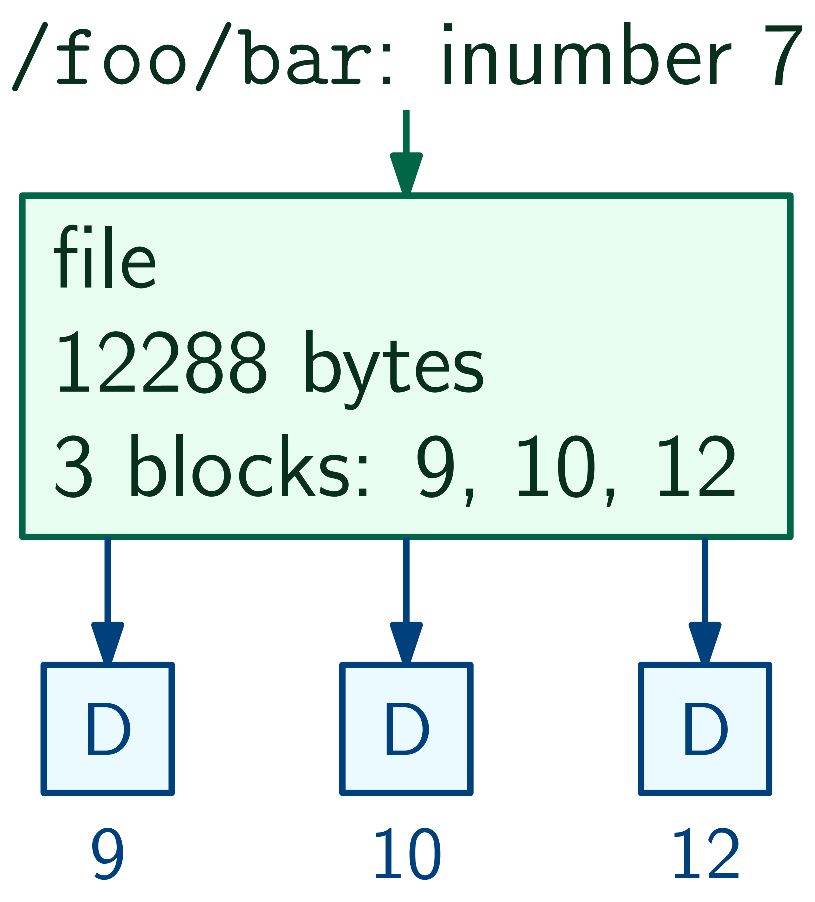


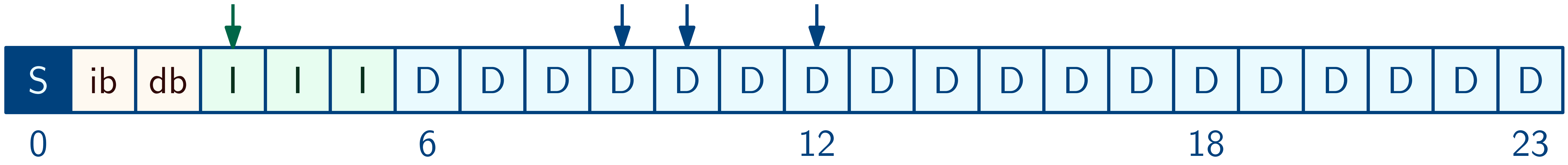


Reading a file

Open file `/foo/bar` and read its contents:

- Read inode 2 (/): stored in data block 6
- Read data block 6: find inode of directory `bar` in /
- Read inode 4 (`/foo/`): stored in data block 8
- Read data block 8: find inode of file `foo` in `/bar/`
- Read inode 7 (`/foo/bar`): stored in data blocks 9, 10, 12
- Read data blocks 9, 10, 12 to get the contents of `/foo/bar`

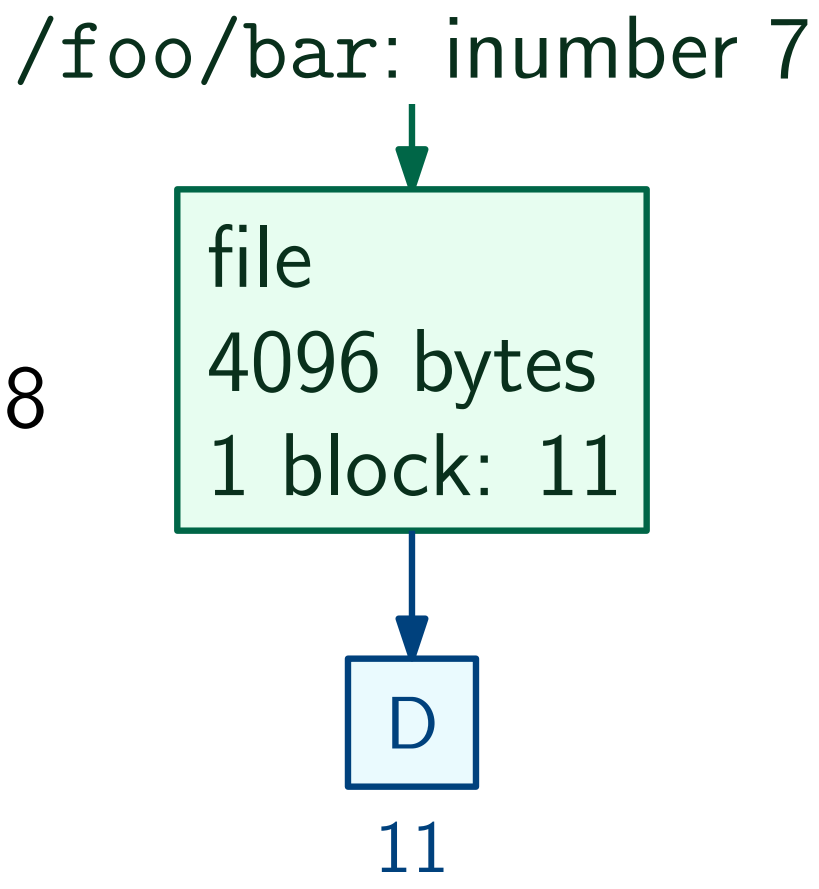


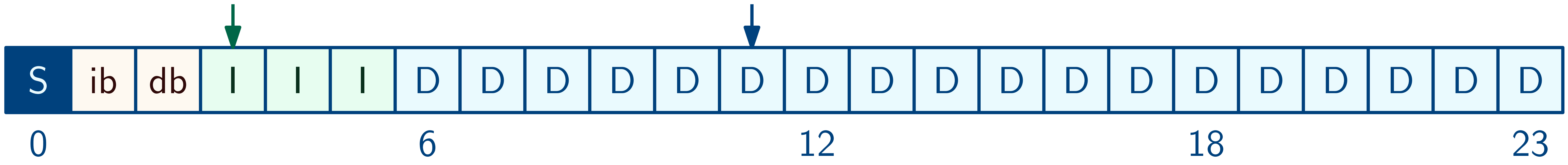


Writing a file

Write to a (new) file `/foo/bar`:

- Inode 2 (/) → data block 6 → inode 4 (`/foo/`) → data block 8
- Search inode bitmap for a free inode: 7
- Add the entry (7, `bar`) to data block 8
- Search data bitmap for a free data block: 11
- Complete inode 7
- Write data to data block 11





Caching and buffering

- Accessing a file generates a lot of reads

  `/1/2/3/.../100/a.txt` $\rightarrow$ more than 200 reads!

- **Solution:** Caching
  - Use a fraction of memory pages to store popular blocks
  - Static partitioning: 10% of total memory
  - Dynamic partitioning: mix memory pages and cached fs pages based on the load

- Writes cannot be cached—memory is not permanent!

- **Solution:** Buffering—accumulate writes, then send to disk
  - Normally, every 5–30 seconds
  - Writing in batch is faster
  - Disk controller can **schedule** writes more efficiently
  - Overwriting the same location will be superceded by the latest request

Summary

- We've seen how to implement the basic functionality of a file system
- Real file systems could have various underlying data structures, and faster/safer ways to perform reads and writes
- Chapter 40: experiments with `vsfs`
- **Assignment 3** out today: in Task 2, you will improve implementation of another very simple file system
- **Next week:** More file systems!