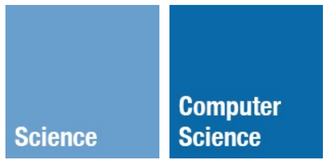
# Filesystems

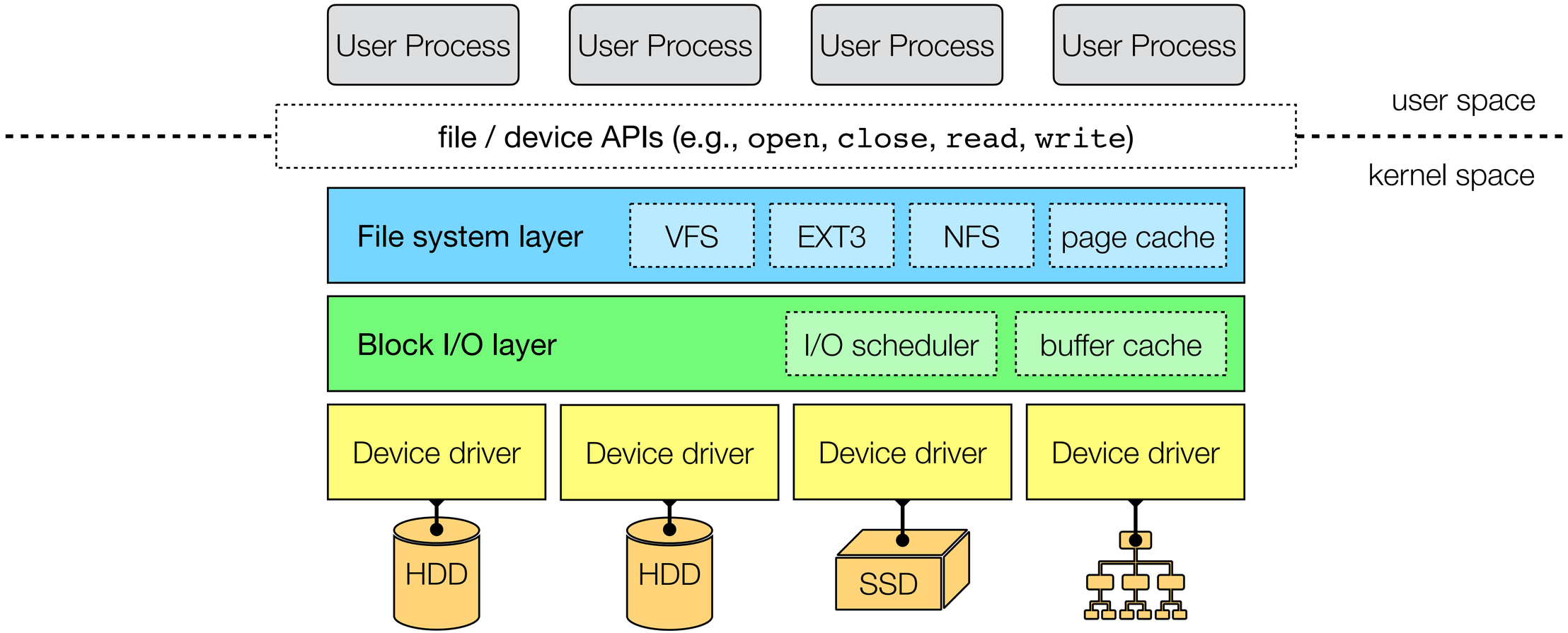



CS 450: Operating Systems
Michael Lee <lee@iit.edu>

# Agenda

- Unix file I/O API
- A simple filesystem
- Fast File System (FFS)
- FS Consistency and Journaling

# The big picture



# § Unix file I/O API

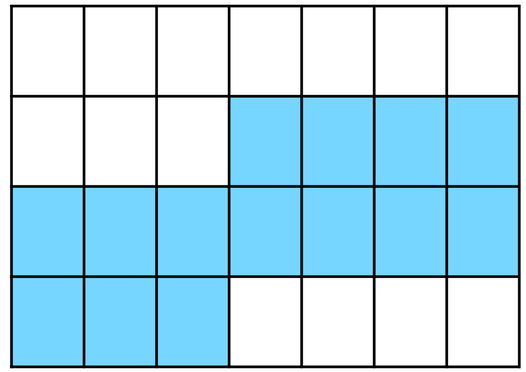
# What *is* a file?

- A named sequence of bytes
  - “Name” = some unique identifier
  - Supports reading, writing, and/or seeking
- A file does not imply any particular backing storage device!
  - But we will be focusing on disk-based files and filesystems

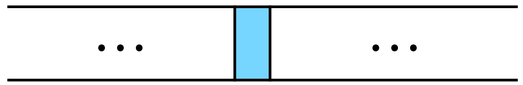
# Block vs. Stream

- *Block-oriented* files use fixed-size blocks as the unit of transfer/access
  - Typically allow blocks to be addressed individually and in arbitrary order
  - E.g., disk-based files
- *Stream-oriented* (aka character-oriented) files can be read/written a byte at a time
  - Typically doesn't support random seeks
  - E.g., network data, mouse input

*block-oriented*



*stream-oriented*



# Unix file “names”

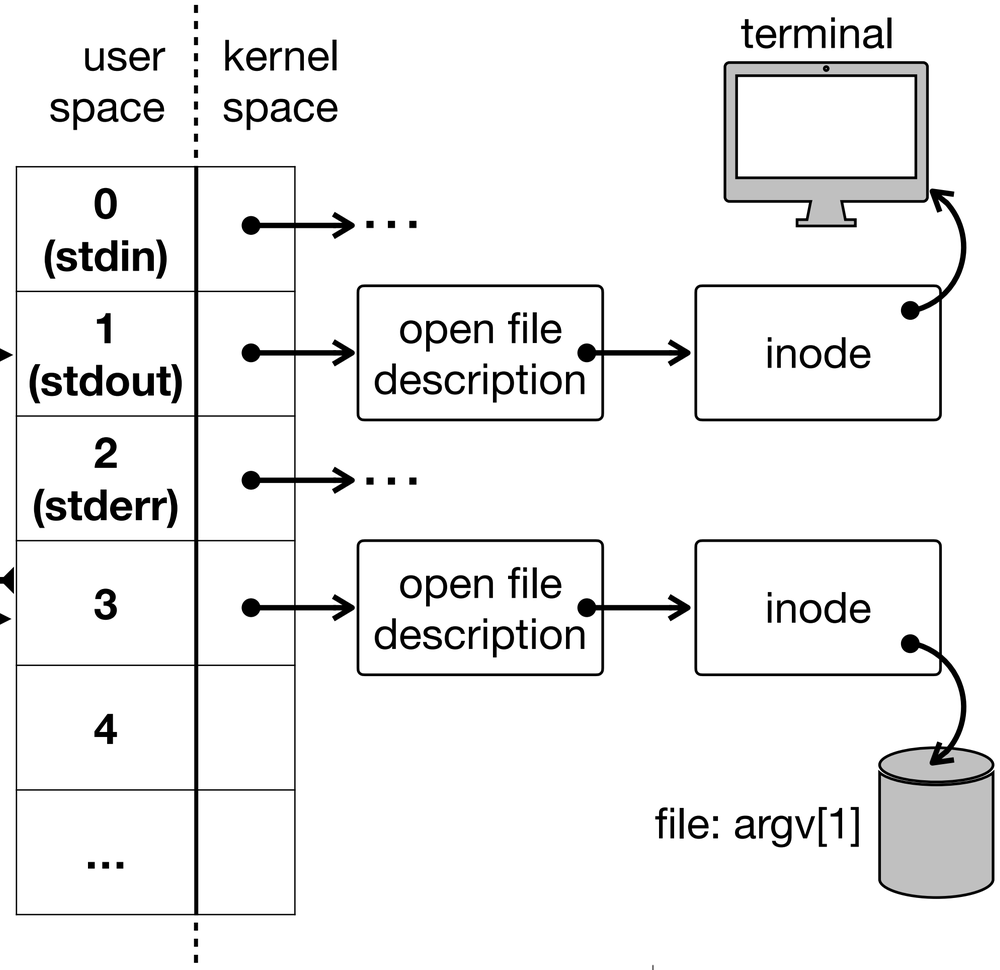
- **Inodes**: each file is assigned a unique inode (“index node”) number
- **Paths**: for user convenience, names in a hierarchical namespace are mapped to inode numbers
  - e.g., `/usr/bin/gcc` → `inode#` — represents a “link” to the file
- **File descriptors**: after opening a file, OS maintains position and access mode in an open file description table
  - Returns index as a file descriptor to user for subsequent access

# Basic file I/O API

```c
// Looks up the inode linked from `path` (creating the underlying file if
// needed) and allocates an open file description. Returns the index (FD).
int open(const char *path, int oflag, ...);

// Read up to `count` bytes from `fd` into `buf`.
ssize_t read(int fd, void *buf, size_t count);

// Write up to `count` bytes from `buf` into `fd`.
ssize_t write(int fd, const void *buf, size_t count);

// Releases `fd`. If there are no remaining references to the open file
// description indexed by `fd`, the OFD is deallocated.
int close(int fd);
```

# E.g., disk I/O

```c
// Simple `cat` utility clone
int main (int argc, char *argv[])
{
    char buf[512];
    int nread;
    int fd = open(argv[1], O_RDONLY);

    while ((nread = read(fd, buf, 512)) > 0)
    {
        write(1, buf, nread);
    }
    close(fd);
    return 0;
}
```



# § A simple filesystem

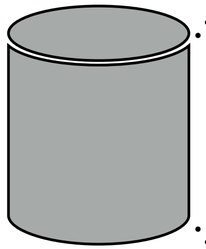
# Essential questions

- How do we keep track of disk/file metadata?
- How do we keep track of free space?
- How do we keep track of disk blocks belonging to a given file?
- How do we associate paths with files? (i.e., human legible name → inode)

# Metadata stores

- Key responsibility of the filesystem is to keep track of various metadata
- **Superblock** used to store centralized global metadata
- Simple yet efficient structure for mapping free space = **free space *bitmap***
- Each file is associated with an **inode block**, which contains metadata and pointers to blocks containing file data
- **Directories** map paths to inodes

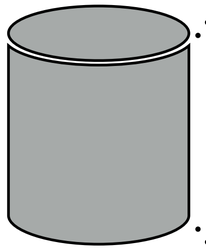
# Disk as an array of blocks



| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 |
| 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |
| 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 |

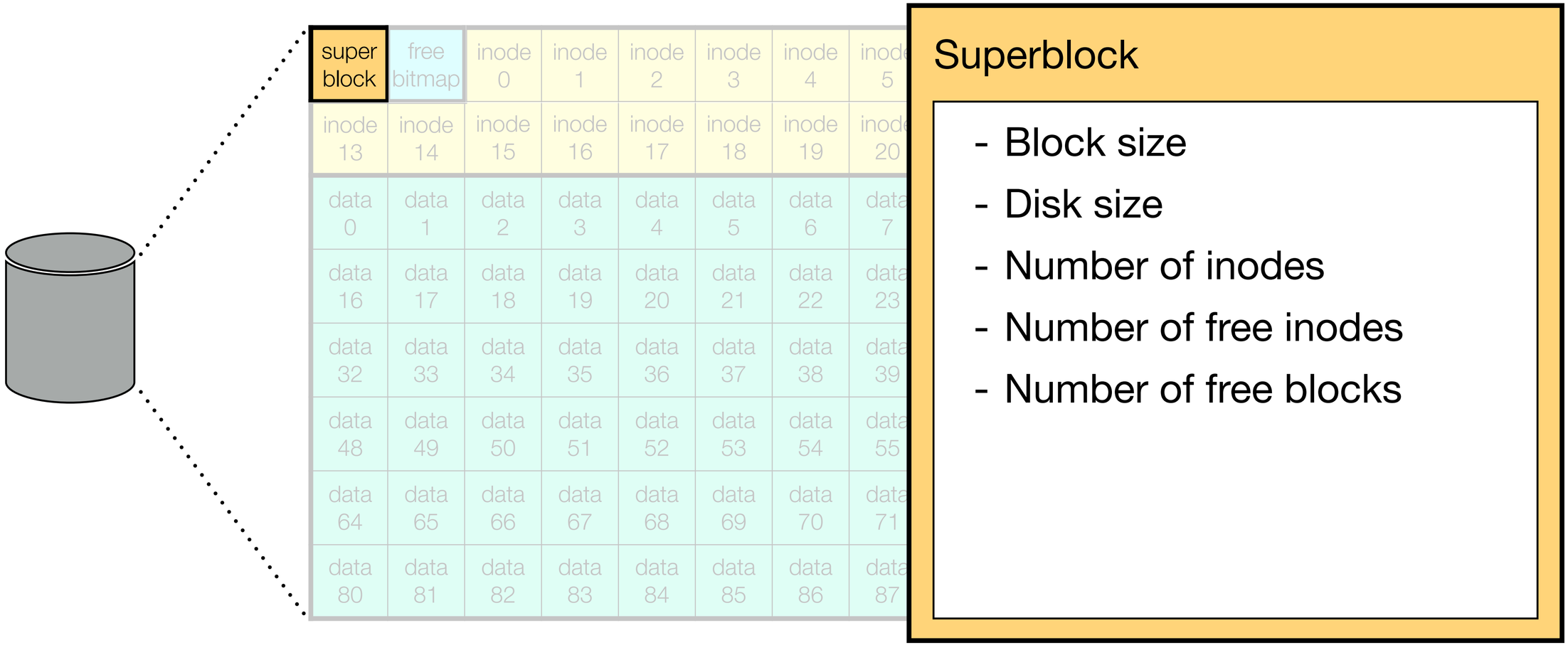
downsides?

# On-disk structures

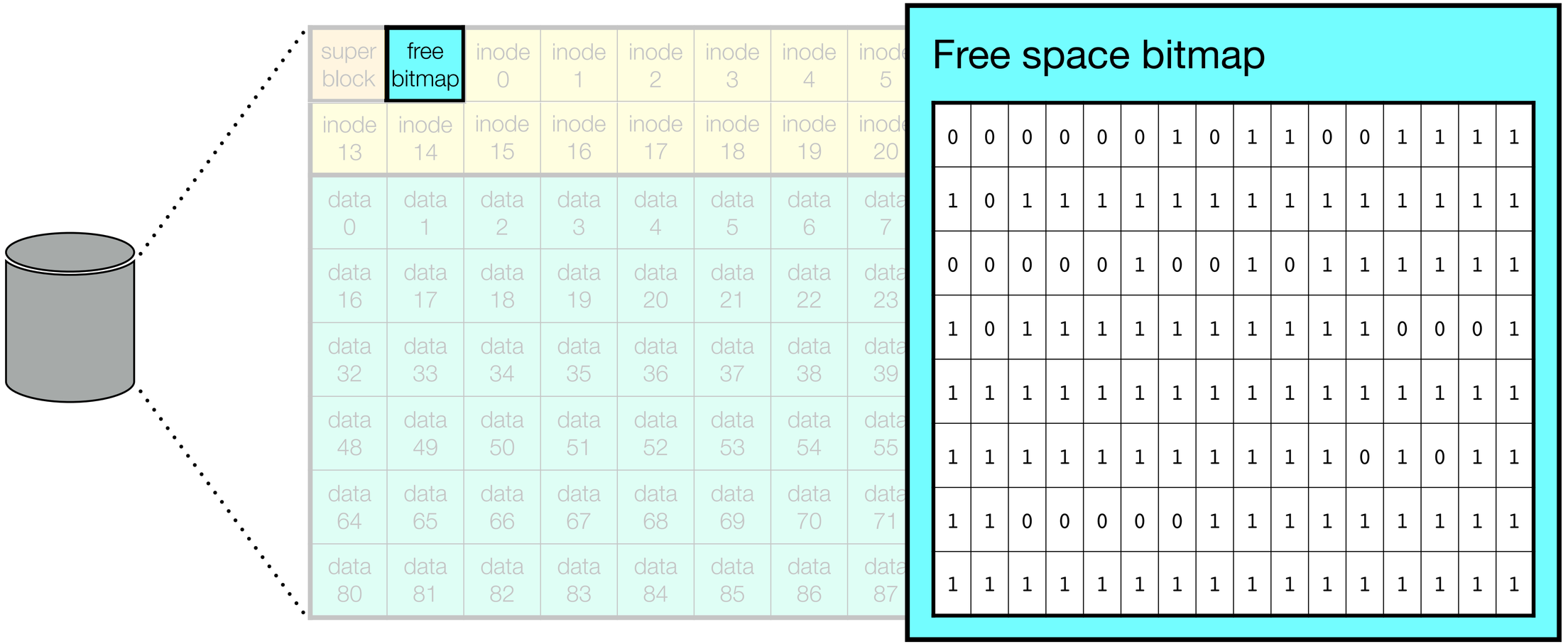


| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| super block | free bitmap | inode 0 | inode 1 | inode 2 | inode 3 | inode 4 | inode 5 | inode 6 | inode 7 | inode 8 | inode 9 | inode 10 | inode 11 | inode 12 | inode 13 |
| inode 13 | inode 14 | inode 15 | inode 16 | inode 17 | inode 18 | inode 19 | inode 20 | inode 21 | inode 22 | inode 23 | inode 24 | inode 25 | inode 26 | inode 27 | inode 28 |
| data 0 | data 1 | data 2 | data 3 | data 4 | data 5 | data 6 | data 7 | data 8 | data 9 | data 10 | data 11 | data 12 | data 13 | data 14 | data 15 |
| data 16 | data 17 | data 18 | data 19 | data 20 | data 21 | data 22 | data 23 | data 24 | data 25 | data 26 | data 27 | data 28 | data 29 | data 30 | data 31 |
| data 32 | data 33 | data 34 | data 35 | data 36 | data 37 | data 38 | data 39 | data 40 | data 41 | data 42 | data 43 | data 44 | data 45 | data 46 | data 47 |
| data 48 | data 49 | data 50 | data 51 | data 52 | data 53 | data 54 | data 55 | data 56 | data 57 | data 58 | data 59 | data 60 | data 61 | data 62 | data 63 |
| data 64 | data 65 | data 66 | data 67 | data 68 | data 69 | data 70 | data 71 | data 72 | data 73 | data 74 | data 75 | data 76 | data 77 | data 78 | data 79 |
| data 80 | data 81 | data 82 | data 83 | data 84 | data 85 | data 86 | data 87 | data 88 | data 89 | data 90 | data 91 | data 92 | data 93 | data 94 | data 95 |

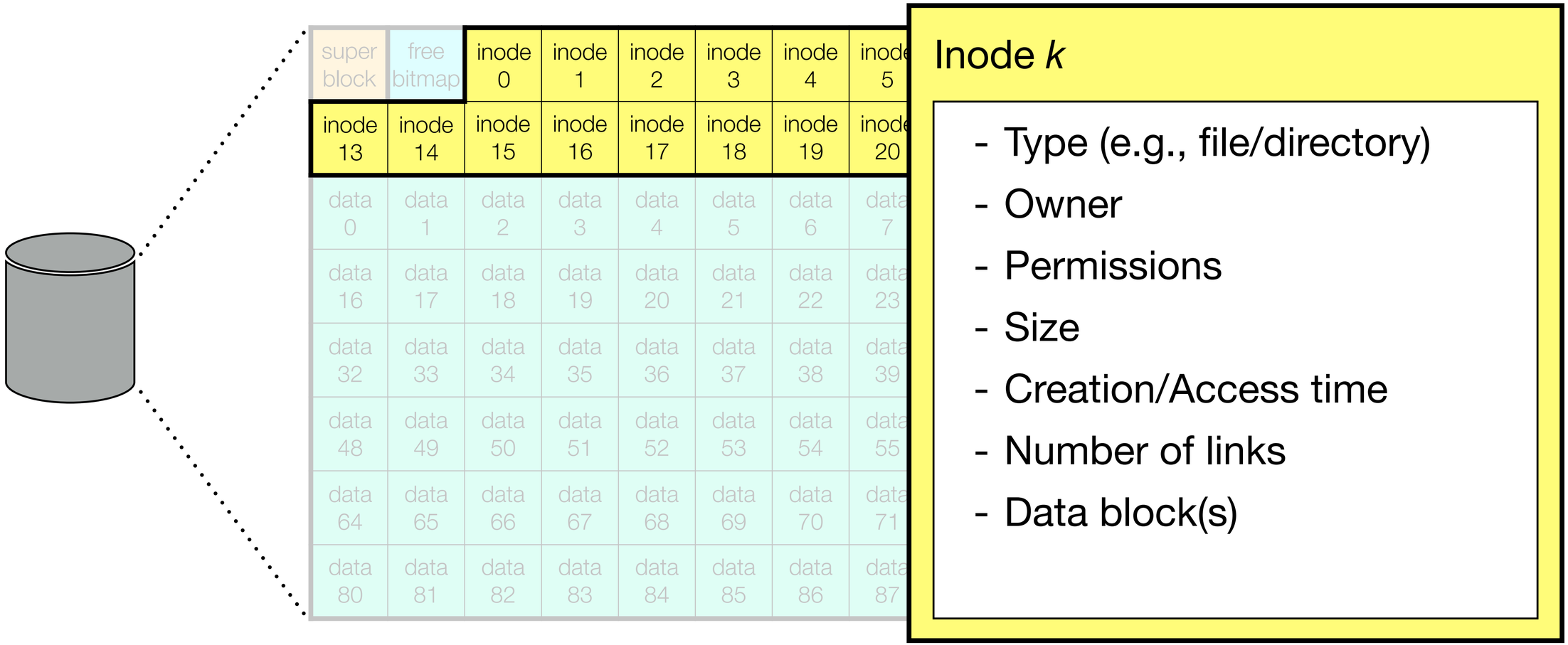
# Superblock


super block
free bitmap
inode 0
inode 1
inode 2
inode 3
inode 4
inode 5
inode 13
inode 14
inode 15
inode 16
inode 17
inode 18
inode 19
inode 20
data 0
data 1
data 2
data 3
data 4
data 5
data 6
data 7
data 16
data 17
data 18
data 19
data 20
data 21
data 22
data 23
data 32
data 33
data 34
data 35
data 36
data 37
data 38
data 39
data 48
data 49
data 50
data 51
data 52
data 53
data 54
data 55
data 64
data 65
data 66
data 67
data 68
data 69
data 70
data 71
data 80
data 81
data 82
data 83
data 84
data 85
data 86
data 87
Superblock
- Block size
- Disk size
- Number of inodes
- Number of free inodes
- Number of free blocks

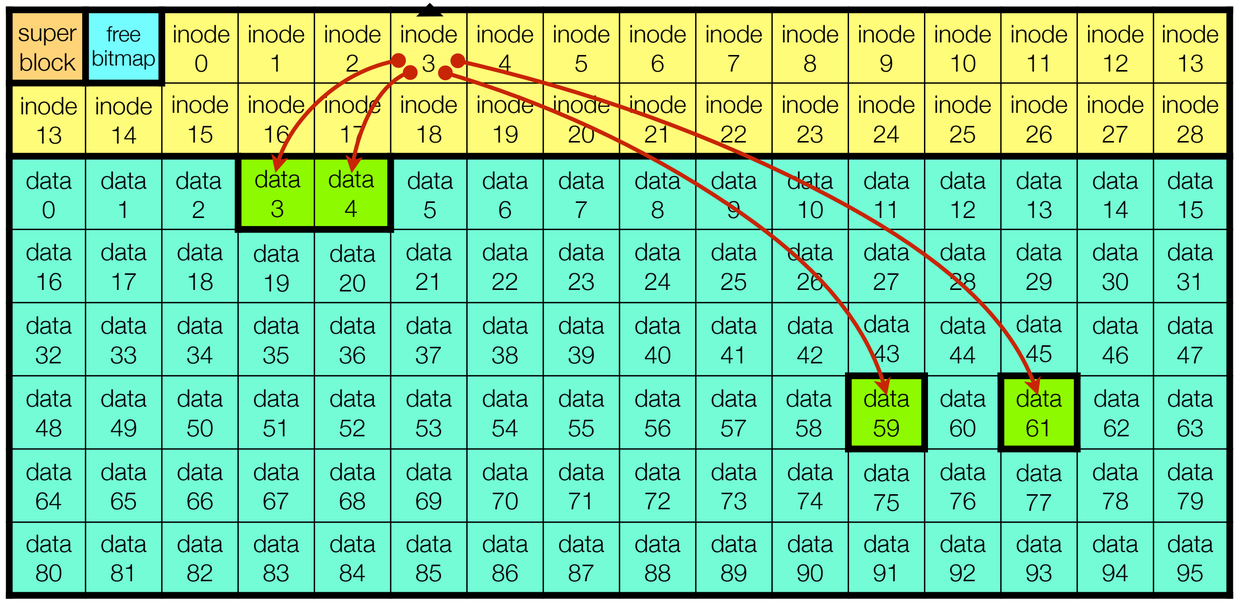
# Free space bitmap



# Inode blocks



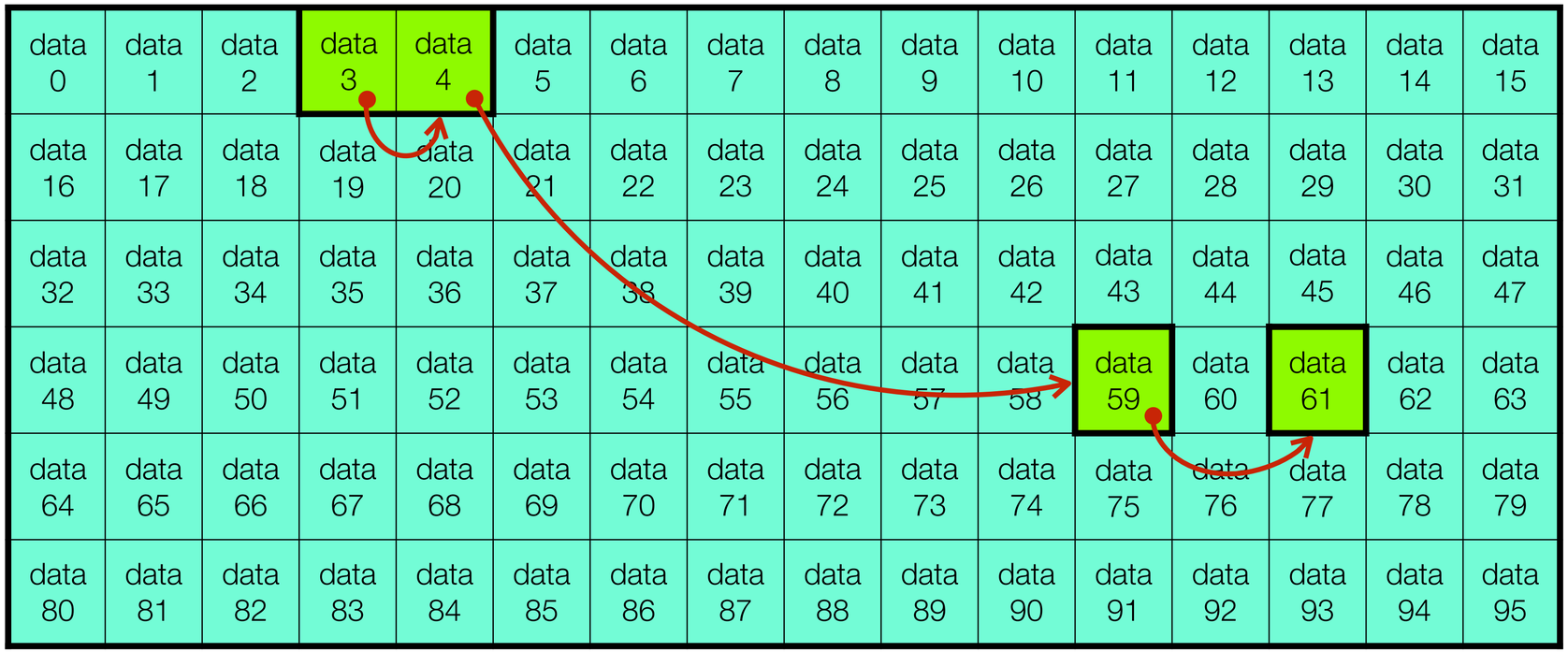
# E.g., inode block contents




## Inode 3

- Type: regular file
- Owner: michael
- RWX bits: 0644
- Size: 2 KB / 4 blocks
- Created/Accessed: [now]
- Number of links: 1 (?)
- Data blocks: 3, 4, 59, 61

# Alternative: linked allocation

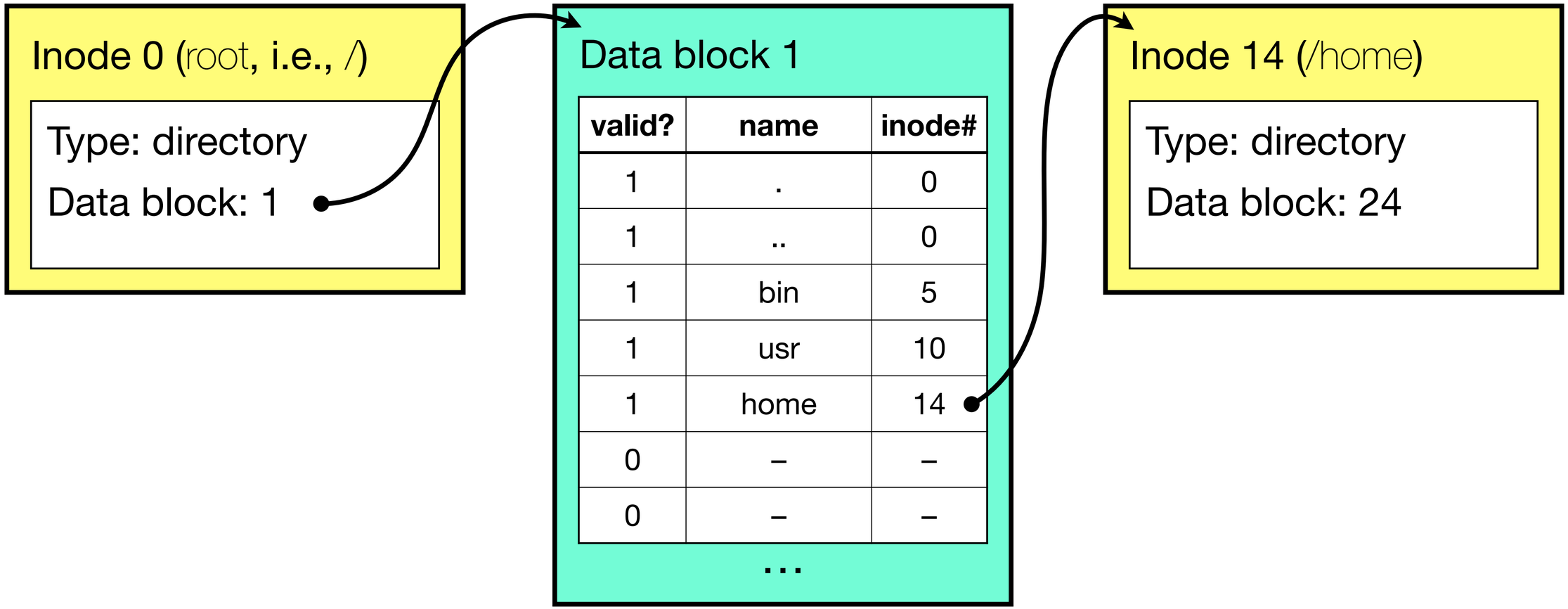



- Store only start block in inode
- Pros/Cons?

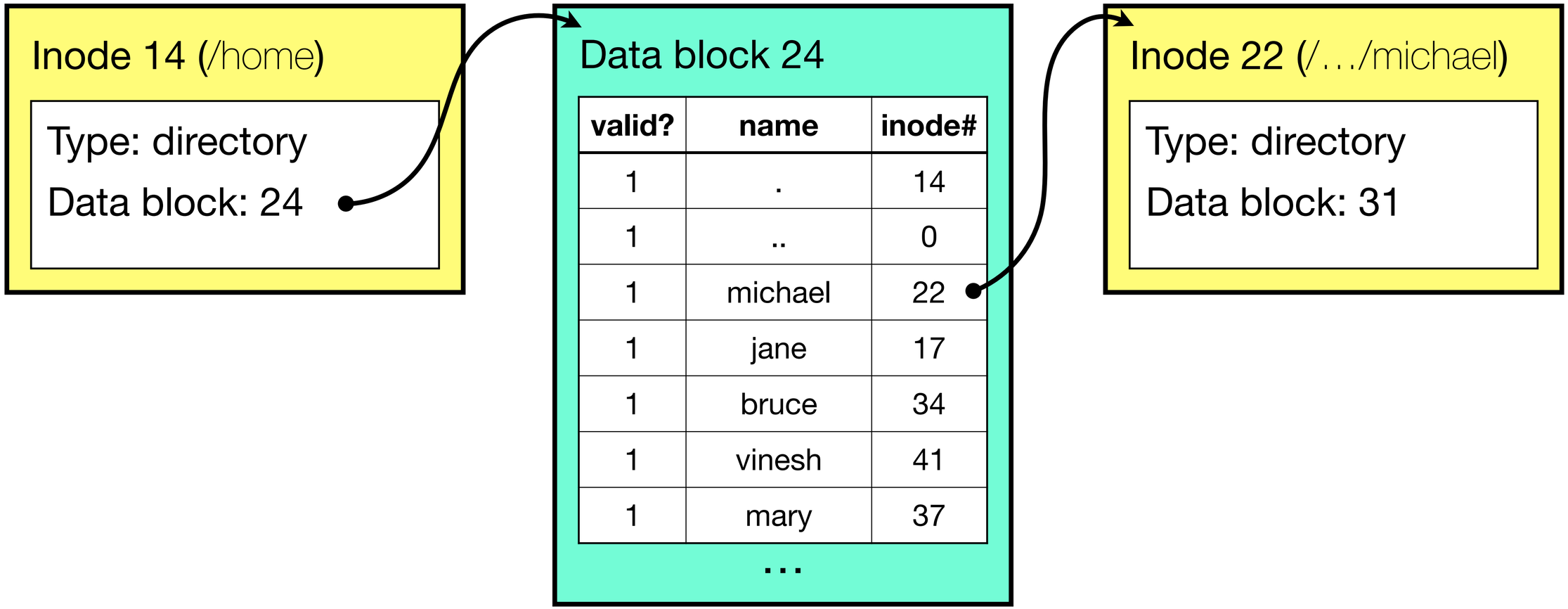
# Path lookup

- Require a path → inode number mapping mechanism
- Could use a monolithic lookup structure, but this would be expensive to maintain and search!
- Instead, each directory maintains a map of contained names to inodes
  - Mappings stored in data blocks, in a format understood by filesystem
  - Root directory inode must be at a known location

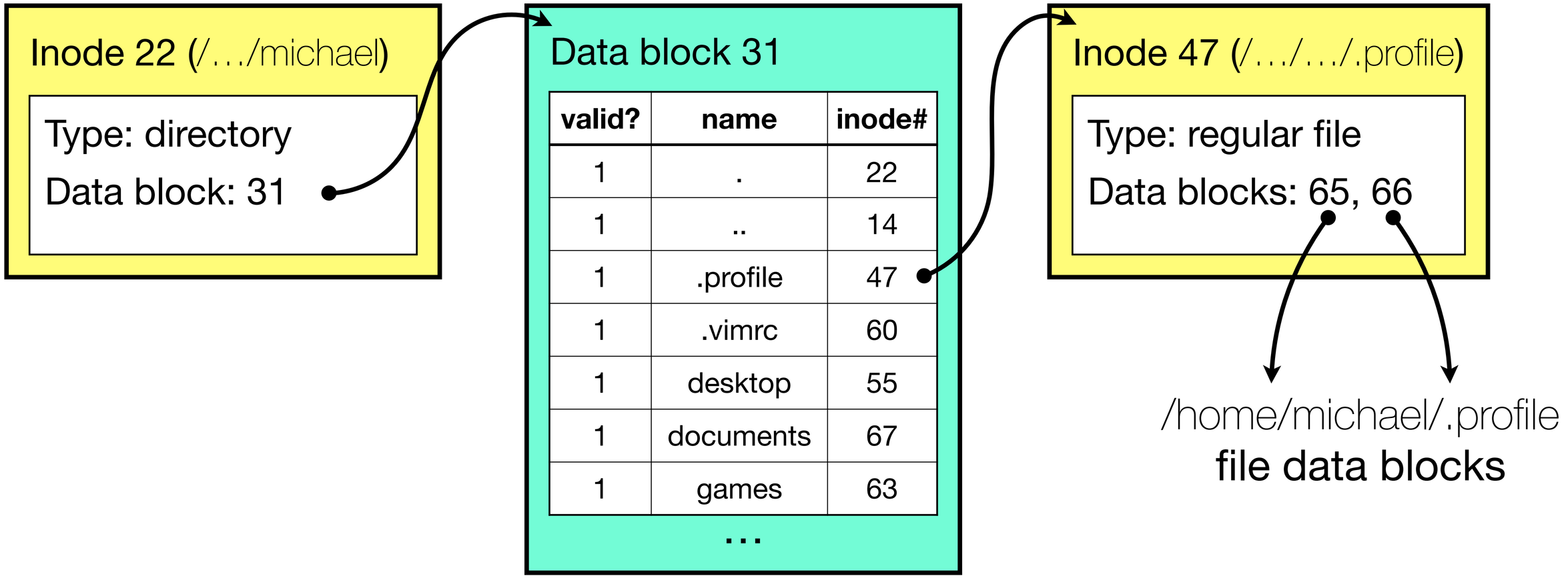
# E.g., read /home/michael/.profile



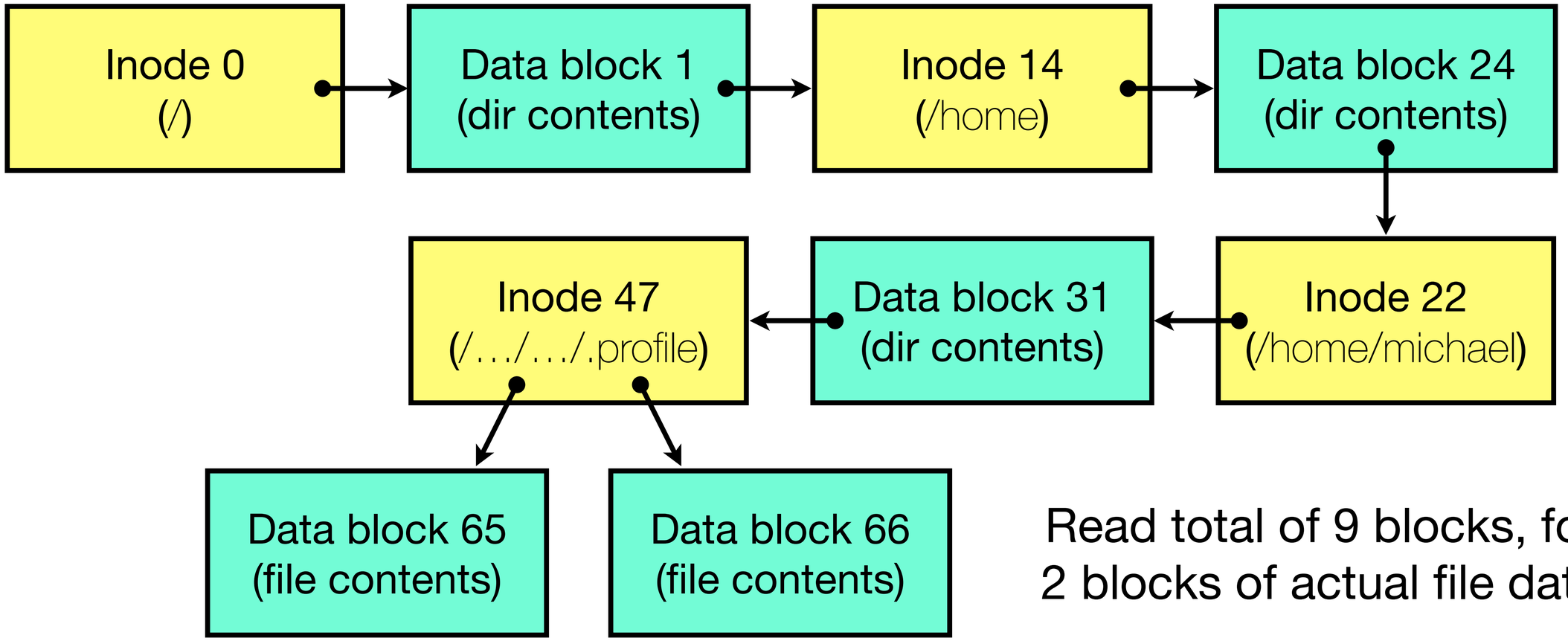
# E.g., read /home/michael/.profile



# E.g., read /home/michael/.profile



# E.g., read /home/michael/.profile

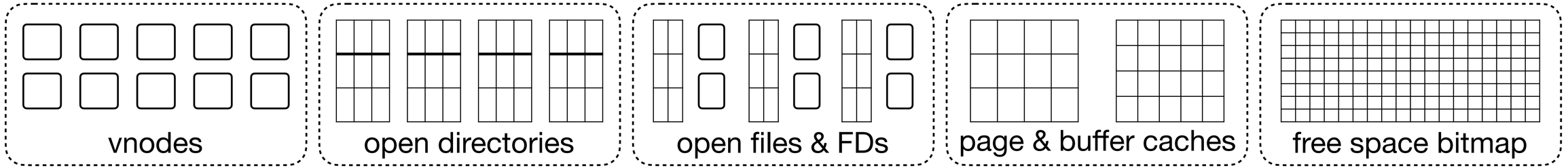


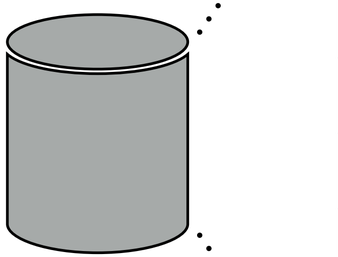

Read total of 9 blocks, for
2 blocks of actual file data

# Caches

- To avoid constant reloading of frequently used metadata/data from disk, filesystem should maintain a cache of structures
  - Vnodes = in-memory inodes
  - Directory mappings
  - Free space bitmap
  - Block-level / File-level data caches
    - aka Buffer / Page caches

# In-memory vs. On-disk structures



| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| super block | free bitmap | inode 0 | inode 1 | inode 2 | inode 3 | inode 4 | inode 5 | inode 6 | inode 7 | inode 8 | inode 9 | inode 10 | inode 11 | inode 12 | inode 13 |
| inode 13 | inode 14 | inode 15 | inode 16 | inode 17 | inode 18 | inode 19 | inode 20 | inode 21 | inode 22 | inode 23 | inode 24 | inode 25 | inode 26 | inode 27 | inode 28 |
| data 0 | data 1 | data 2 | data 3 | data 4 | data 5 | data 6 | data 7 | data 8 | data 9 | data 10 | data 11 | data 12 | data 13 | data 14 | data 15 |
| data 16 | data 17 | data 18 | data 19 | data 20 | data 21 | data 22 | data 23 | data 24 | data 25 | data 26 | data 27 | data 28 | data 29 | data 30 | data 31 |
| data 32 | data 33 | data 34 | data 35 | data 36 | data 37 | data 38 | data 39 | data 40 | data 41 | data 42 | data 43 | data 44 | data 45 | data 46 | data 47 |
| data 48 | data 49 | data 50 | data 51 | data 52 | data 53 | data 54 | data 55 | data 56 | data 57 | data 58 | data 59 | data 60 | data 61 | data 62 | data 63 |
| data 64 | data 65 | data 66 | data 67 | data 68 | data 69 | data 70 | data 71 | data 72 | data 73 | data 74 | data 75 | data 76 | data 77 | data 78 | data 79 |
| data 80 | data 81 | data 82 | data 83 | data 84 | data 85 | data 86 | data 87 | data 88 | data 89 | data 90 | data 91 | data 92 | data 93 | data 94 | data 95 |

# Problems with caching?

- The more data we cache, and the longer we cache it, the more *out of sync* in-memory and on-disk structures may get
  - Problem?
    - OS/FS crash before sync happens can create big problems!
      - Lost data if not written yet
      - Filesystem structure inconsistencies

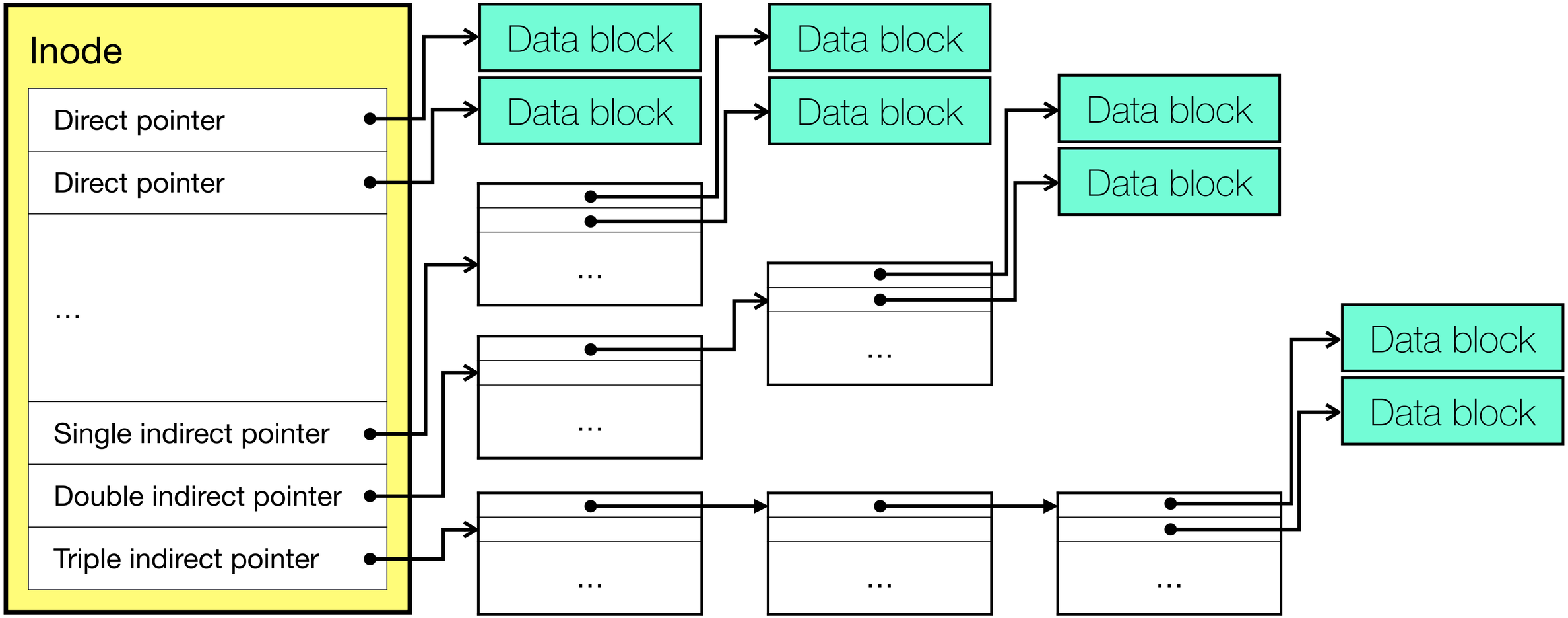
# File size limitation

- Recall, inodes reside in fixed size blocks, so can only hold a finite number of pointers to data blocks
- Typically, an inode holds 12 pointers to data blocks
  - With a block size of 512 bytes, max file size = 6 KB
- How to fix this?

# Indirect pointers

- We can store pointers to blocks that contain more pointers …
  - A single-indirect pointer references a block that contains multiple direct pointers to data blocks belonging to the file
  - A double-indirect pointer references a block that contains multiple single-indirect pointers
  - Etc.

# Indirect pointers



# E.g., computing max file size

Given a block size of 512 bytes, 32-bit block pointers, and that an inode contains 12 direct, 1 single indirect, 1 double indirect, and 1 triple indirect pointers, what is the maximum file size?

- Direct: 512 bytes × 12 = 6 KB
- Pointers per block: 512 / 4 = 128 ($2^7$)
- Single indirect: 512 bytes × 128 = $2^9 \times 2^7 = 2^{16}$ bytes (64 KB)
- Double indirect: $2^9 \times 2^7 \times 2^7 = 2^{23}$ bytes (8 MB)
- Triple indirect: $2^9 \times 2^7 \times 2^7 \times 2^7 = 2^{30}$ bytes (1 GB) **Ans: ~1.008 GB**