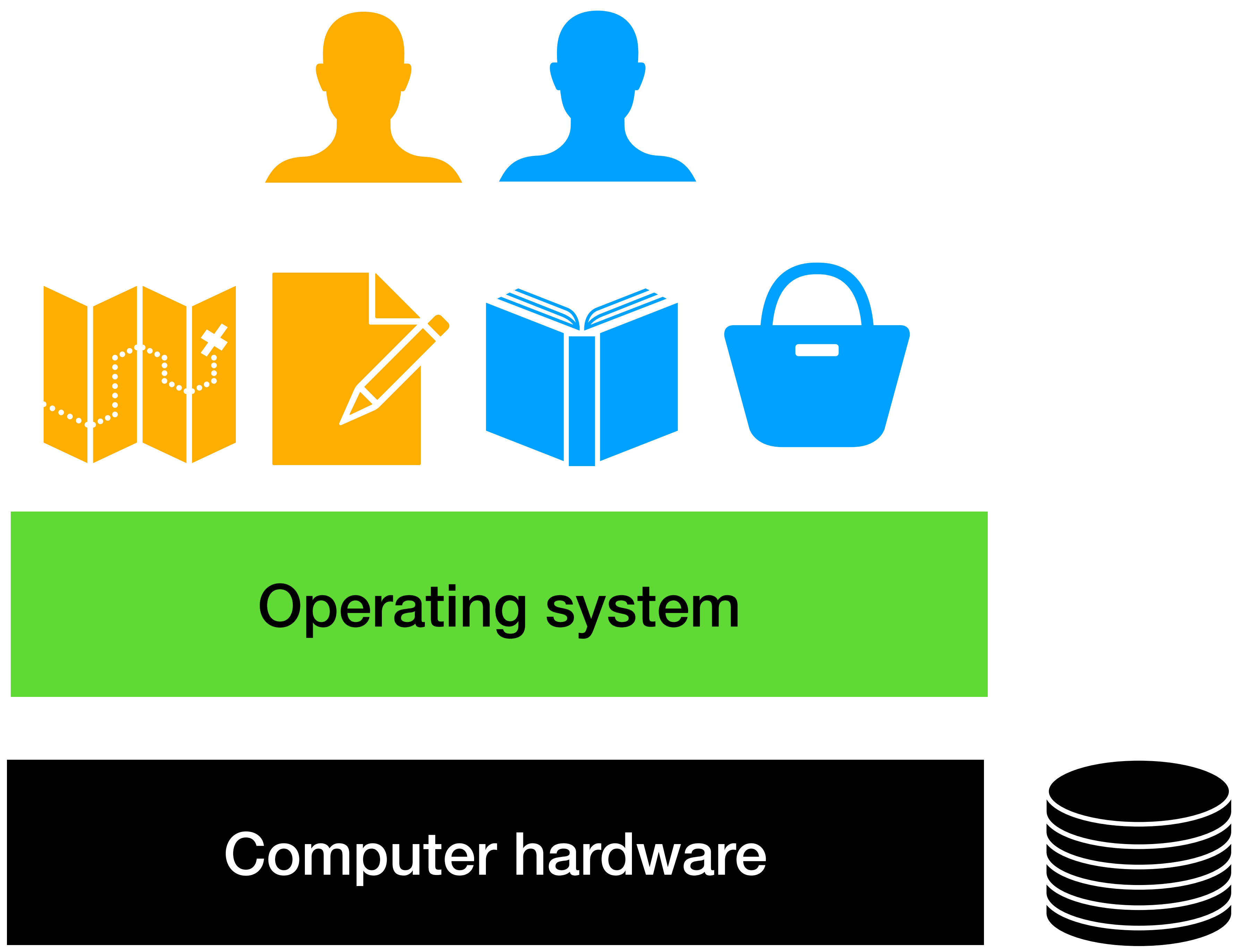
File system

Abhilash Jindal

File system



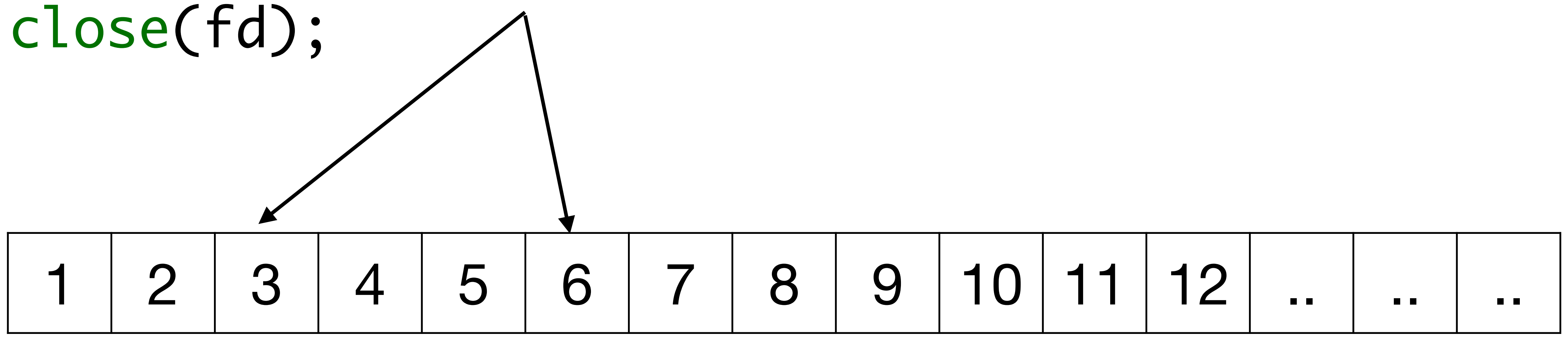
Example: io.c

Disk interface: List of blocks

File system OS interface: Folders and files.

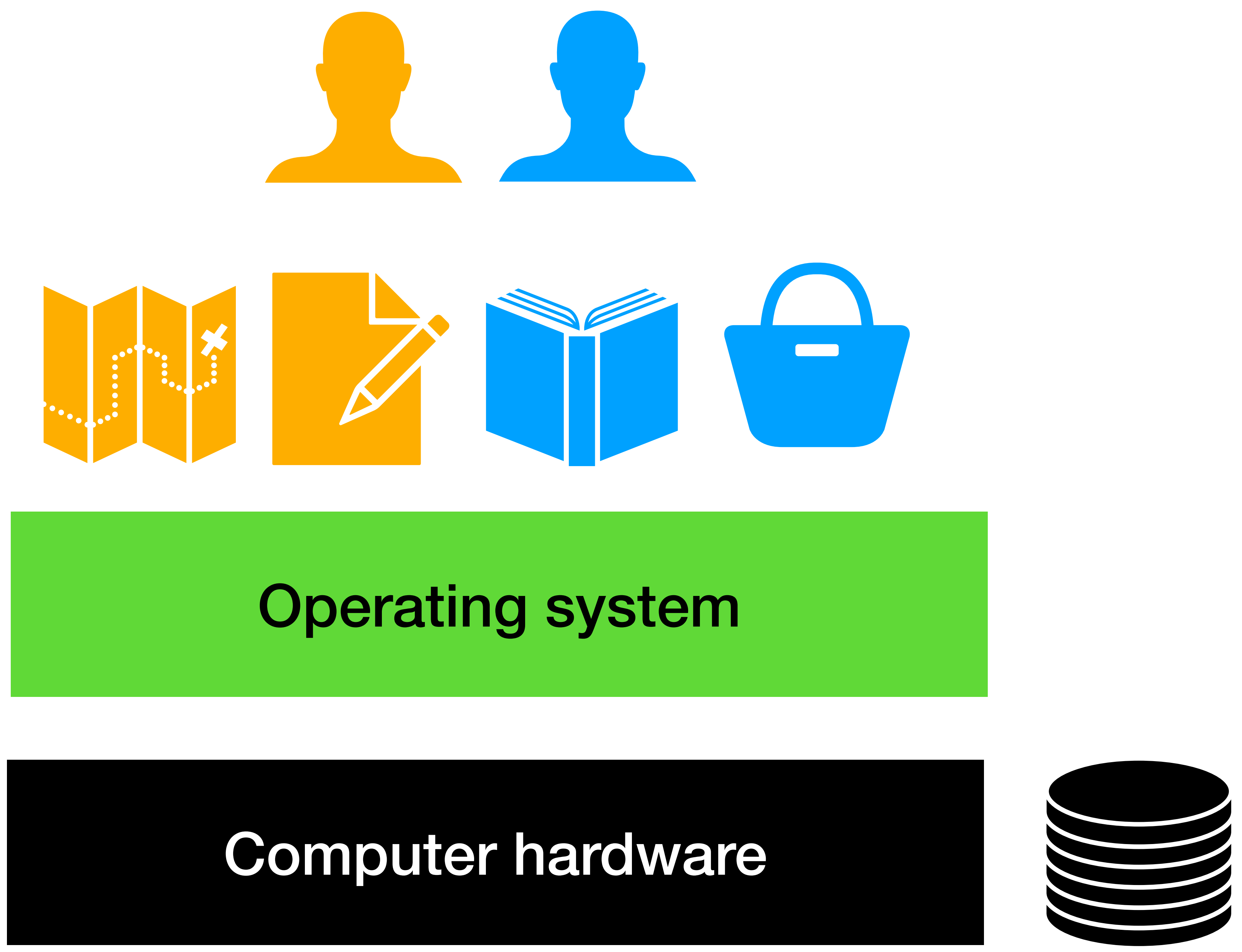
open, write, read, lseek, close, append, permissions, truncation, file descriptor offset

```c
int fd = open("/tmp/file", O_WRONLY | O_CREAT);
int rc = write(fd, "hello world\n", 12);
close(fd);
```

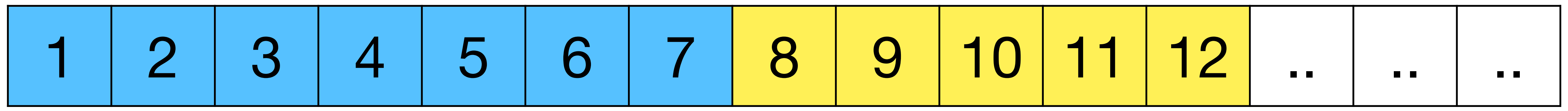


Why file system?

Why not just multiplex disk blocks like memory?



- Disk blocks live after programs exits, computer restarts
- Different programs read / write same file
  - vim writes io.c
  - gcc reads io.c, write io
  - We finally run io



Files as sequence of bytes

- Other options: Files have structured records
  - Can build structure on top
  - But may not optimise disk accesses
- Also expose raw disk blocks
  - Databases
  - File system checker (fsck)
  - Disk defragmenter

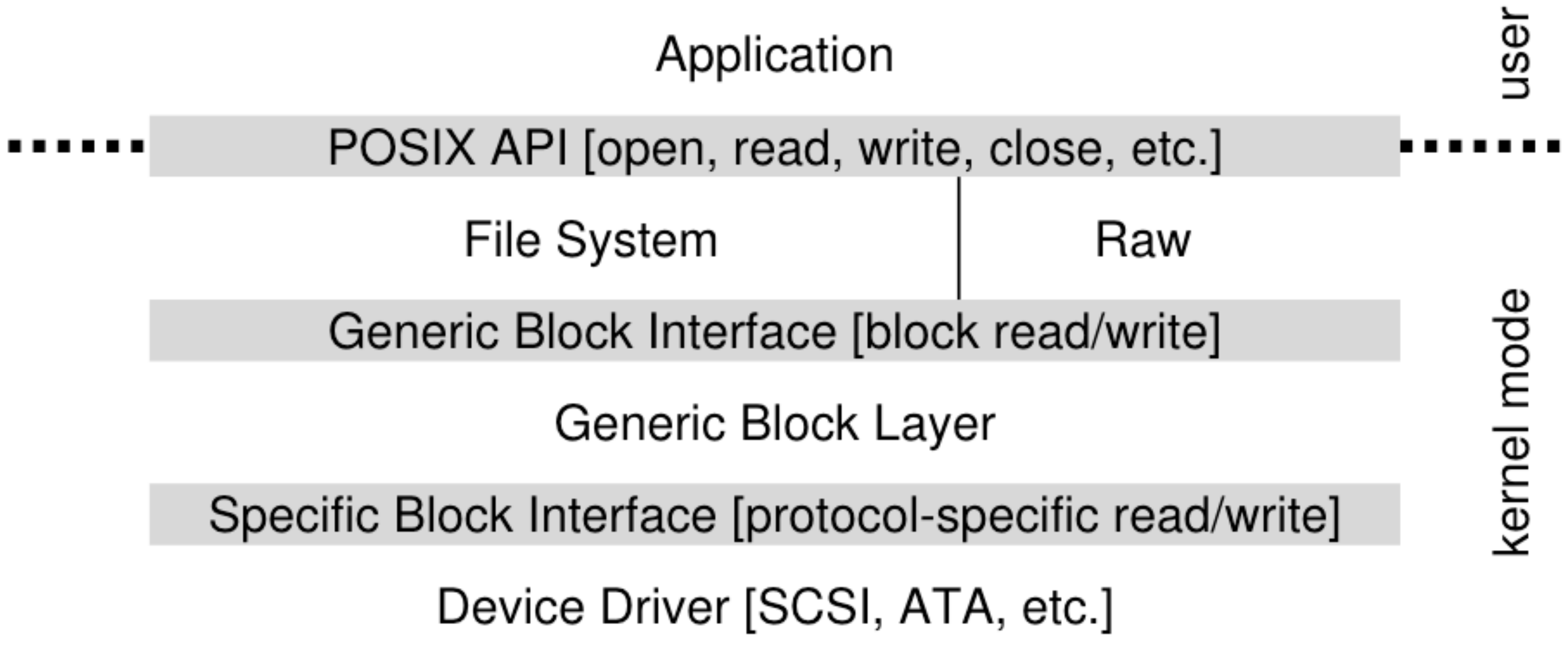


Figure 36.4: **The File System Stack**

Flexible abstraction

- Stitch multiple file systems into a common directory tree
  - `mount -t ext3 /dev/sda1 /home/abhilash/photos/`
  - `mount -t ext2 /dev/sdb1 /home/abhilash/docs/`
- /proc
- Run tty. cat <filename>
- /sys

Agenda

- Build a file system (OSTEP Ch. 40, xv6 Ch. 6)
  - On-disk data structure. Organize disk blocks to expose files and directories
- Optimizations (OSTEP Ch. 41)
- Crash consistency: Don’t lose data when computer restarts (OSTEP Ch. 42)

File system characteristics

- File system contains lots of files ~100K
- Most files are small ~2KB
- A few big files use most of the disk space
- Directories have typically < 20 files and directories

xv6 file system

File system implementation , OSTEP Ch.40, xv6 Ch. 6

How to store files?

Contiguous allocation

1	2	3	4	5	6	7	8	9	10	11	12

- “a.txt” -> (base = 1, size = 2)
- “b.txt” -> (base = 8, size = 2)

- Growth. “b.txt” wants to use 6 blocks. Need to copy to a new location.

- Fragmentation. Want to create a file “c.txt” with 6 blocks.

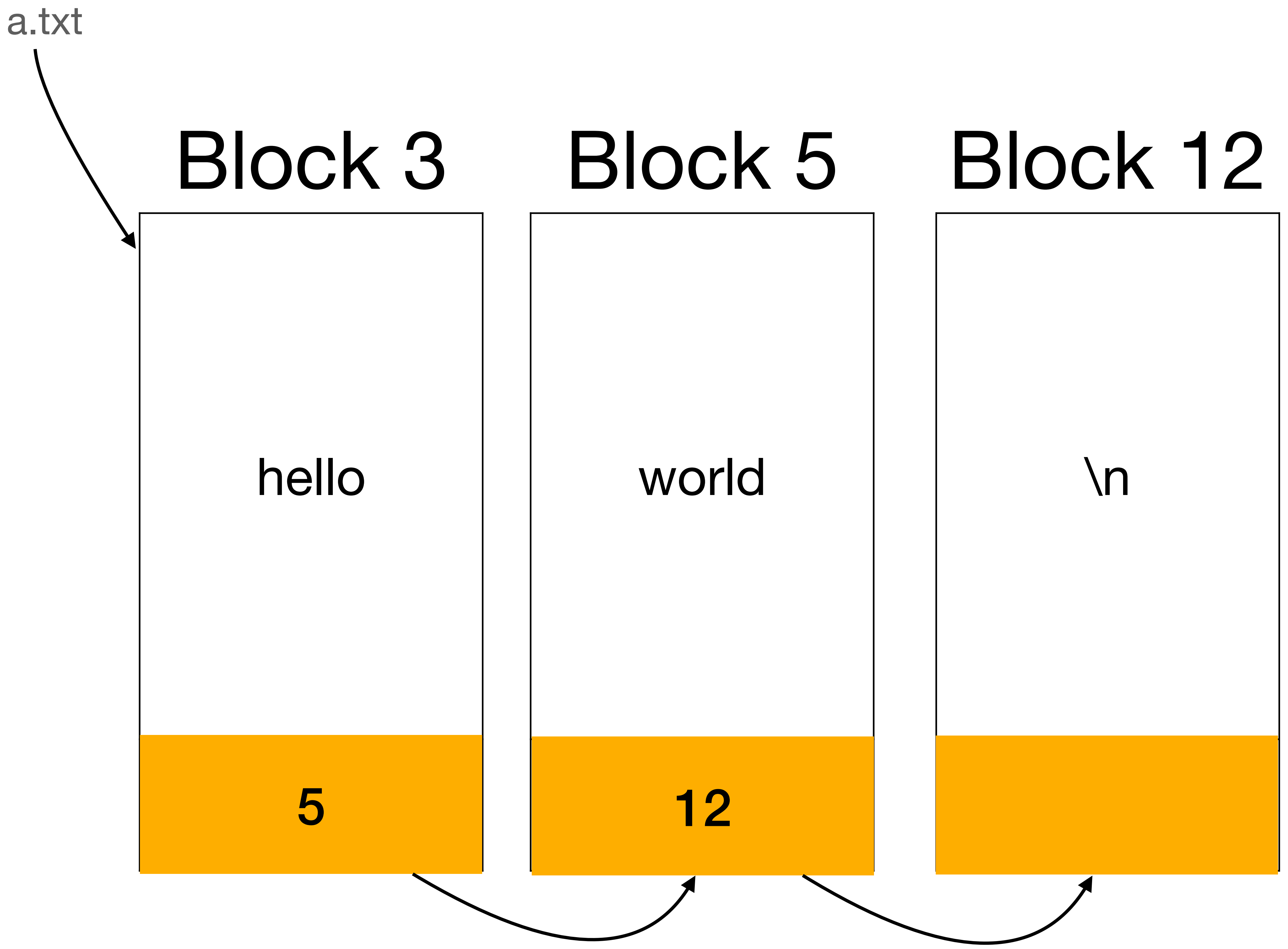
✓ Sequential file rw is sequential disk rw

How to store files?

Linked list of blocks

✓ Files can grow easily

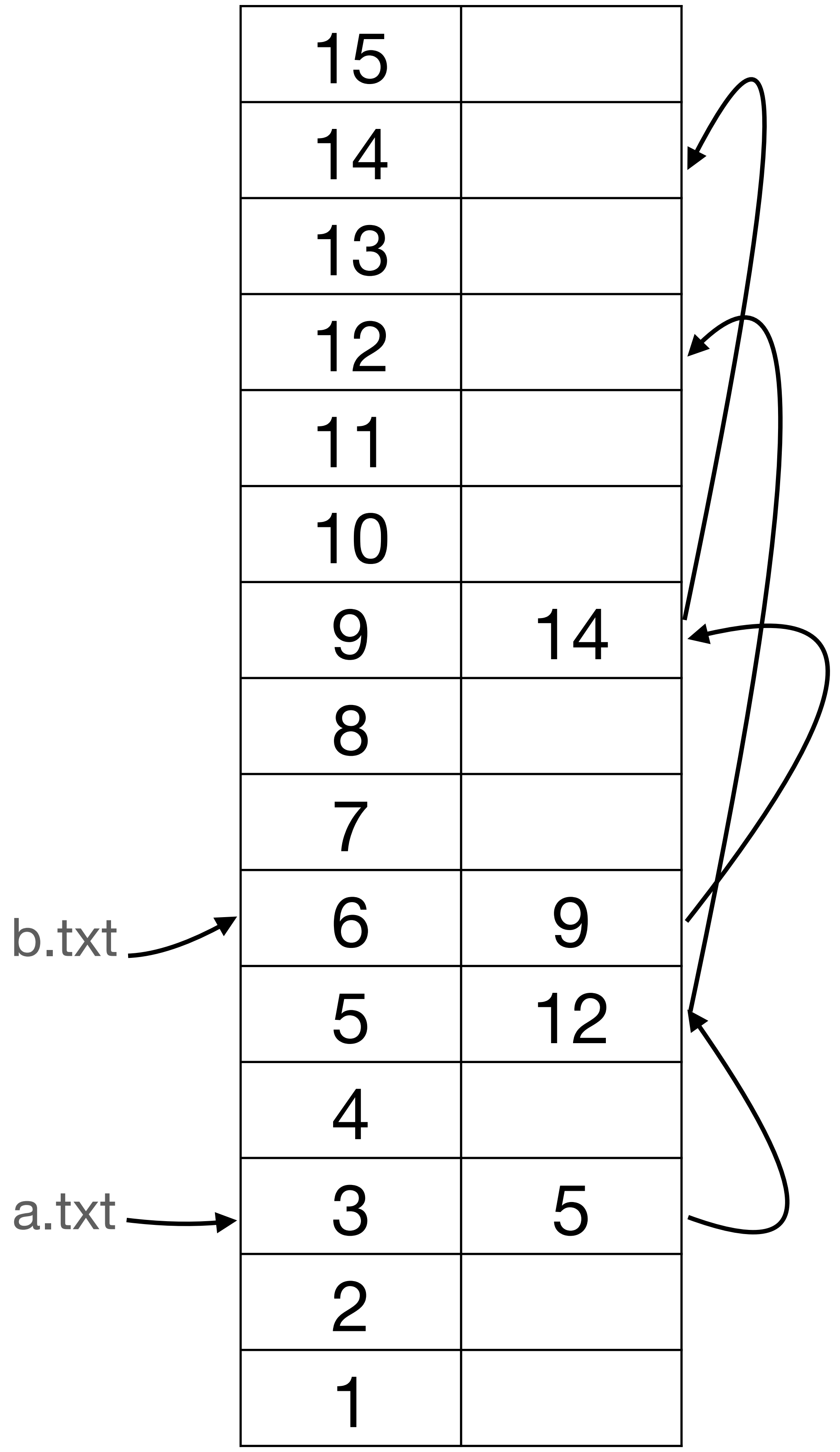
- Seeks / appends are terrible:
  - Need to read the whole file
- Sequential rws become random disk rws
  - Cannot send >1 in-flight IO requests. Lose disk scheduling potential.
- If one block gets corrupted, parts of the file is lost



How to store files?

File Allocation Table (FAT filesystem)

- Fast seeks/appends
  - Bring table into memory, do pointer chasing in memory
- Size of block: 2KB to 32KB.
  - FAT16
    - 2^16 entries. Maximum disk size: 2^16 * 2KB = 128 MB
    - Size of table = 2^16*(2 bytes) = 128KB
  - FAT32
    - 2^28 entries. Maximum disk size: 2^28 * 2KB = 512 GB
    - Size of table = 2^28 * (4 bytes) = 1GB
- Reliability:
  - Lose file system if we lose FAT table. Keep two copies.



Block size in FAT

- Large block size:

  ✓ Support larger disks

  ✓ Reduced random IO

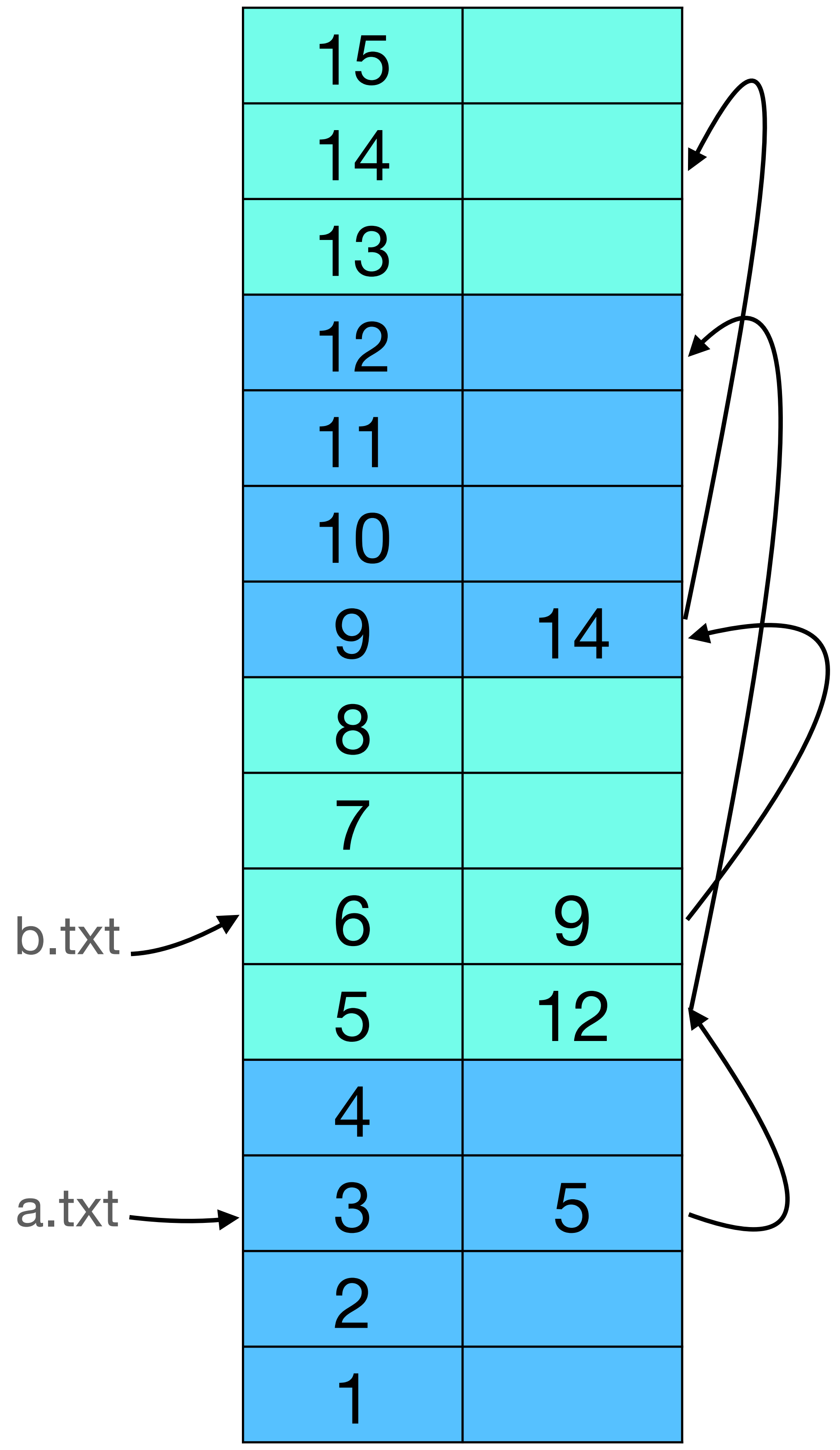
  ✓ Reduced Metadata overhead. FAT32 overhead: 4 bytes / 2KB ~ 0.2%; 4 bytes / 32KB ~ 0.01%

  - Increased internal fragmentation: minimum file size is block size

  - Increase buffer cache pressure: lesser number of blocks can be cached

Performance

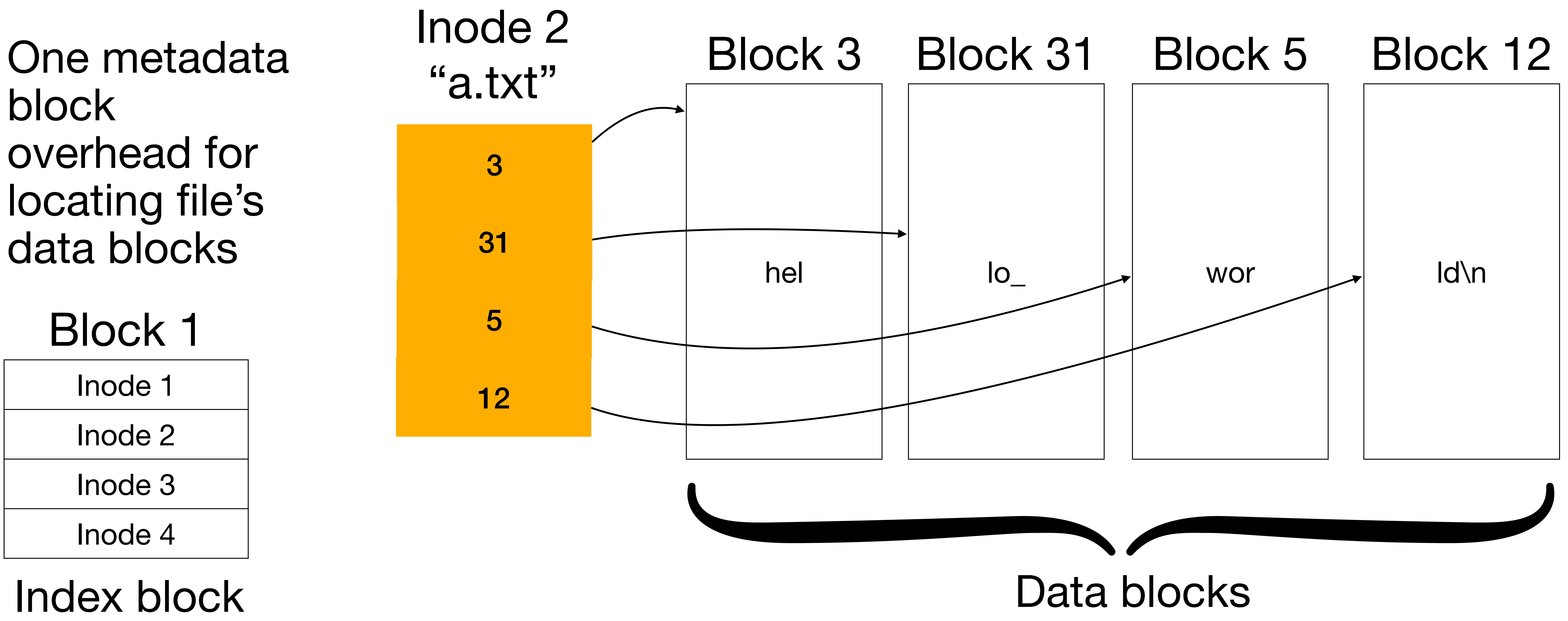
- Sequential IO
  - Better than linked list. Can find the list of blocks apriori and send requests. Disk controller can schedule them.
  - Worse than contiguous allocation since it did only 1 seek.
- Random IO
  - As fast as it can be. Find the block in memory and send disk request
- Use buffer cache for FAT table when it does not fit in memory (1GB for FAT-32)
  - To locate file's blocks, we might have to read many metadata blocks



How to store files?

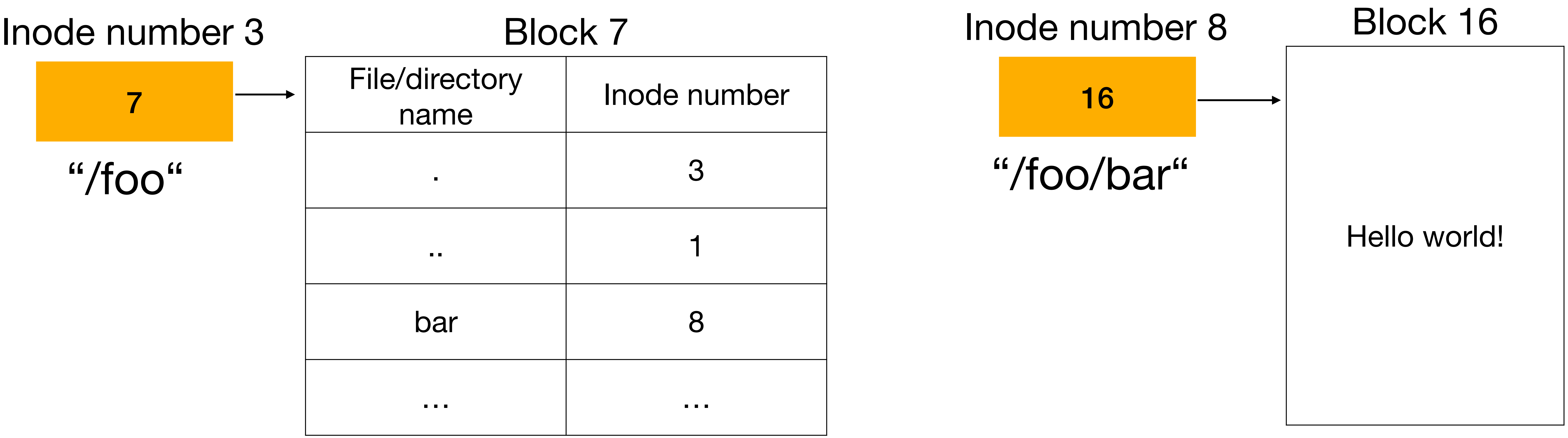
Index and data blocks

- One metadata block overhead for locating file's data blocks



How to store directories?

/foo/bar



File/directory name	Inode number
.	3
..	1
bar	8
…	…

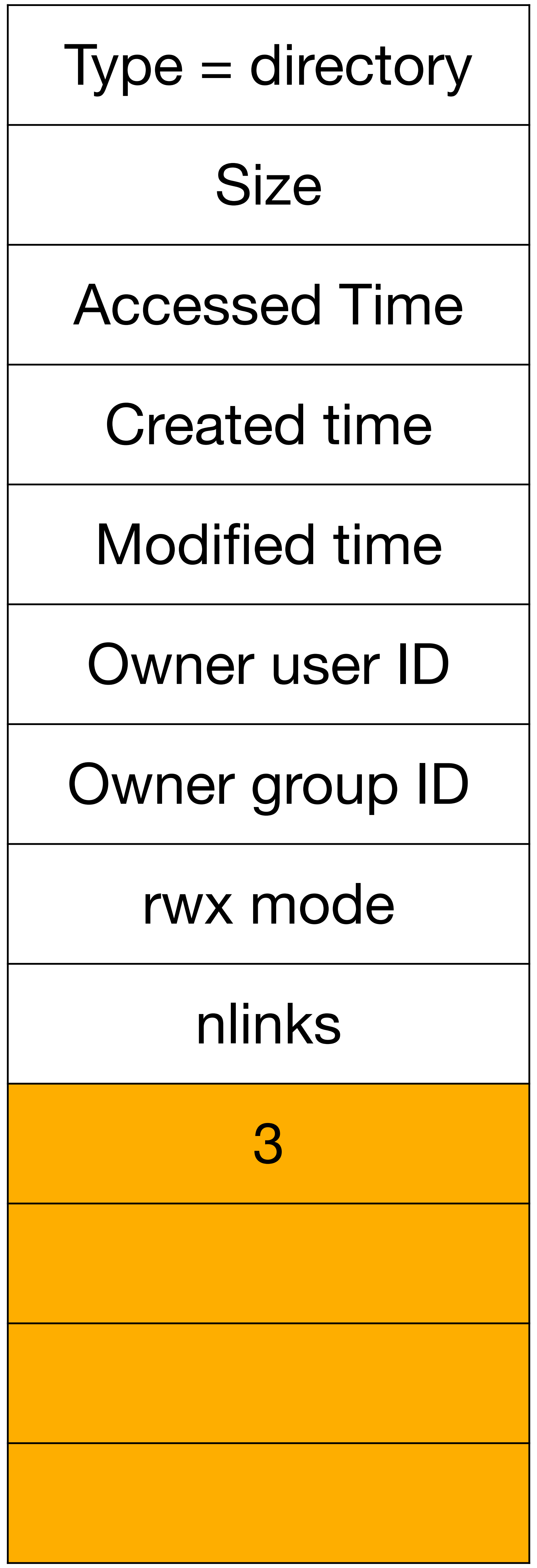
- In xv6, each directory entry is 16 bytes. 32 (=512/16) directory entries in one data block
- Directories have typically < 20 files and directories

Other things in inode

```
stat /tmp/file

  File: /tmp/file
  Size: 14         Blocks: 8          IO Block: 4096   regular file
Device: 803h/2051d  Inode: 22414820    Links: 1
Access: (0600/-rw-------)  Uid: ( 1000/    dell)   Gid: ( 1000/    dell)
Access: 2024-01-24 06:29:51.395609006 +0530
Modify: 2024-01-24 06:29:51.395609006 +0530
Change: 2024-01-24 06:29:51.395609006 +0530
 Birth: -


Modify time: last time when data nodes were changed
Change time: last time when inode was changed
```



File system layout

Example: /foo/bar

Inode = 1 “/“

Type = directory
Size
2

Block 2

File/directory name	Inode number
.	1
foo	8

Inode = 8 “/foo“

Type = directory
Size
16

Block 16

File/directory name	Inode number
.	8
..	1
bar	9

Inode = 9 “/foo/bar”

Type = file
Size
3
31
5
12

Block 3	Block 31	Block 5	Block 12
hel	lo	wor	ld

Reading a file

Example: /foo/bar

```
char buf[10];
fd = open("/foo/bar", O_RDONLY)

while(read(fd, &buf, 10) > 0) {
    // print buf etc.
}
close(fd);
```

Inode = 1 "/"

	Type = directory
	Size
	Access time
1	2

Block 2

	File/directory name	Inode number
	.	1
2	foo	8

Inode = 8 "/foo"

	Type = directory
	Size
	Access time
3	16

Block 16

	File/directory name	Inode number
	.	8
	..	1
4	bar	9

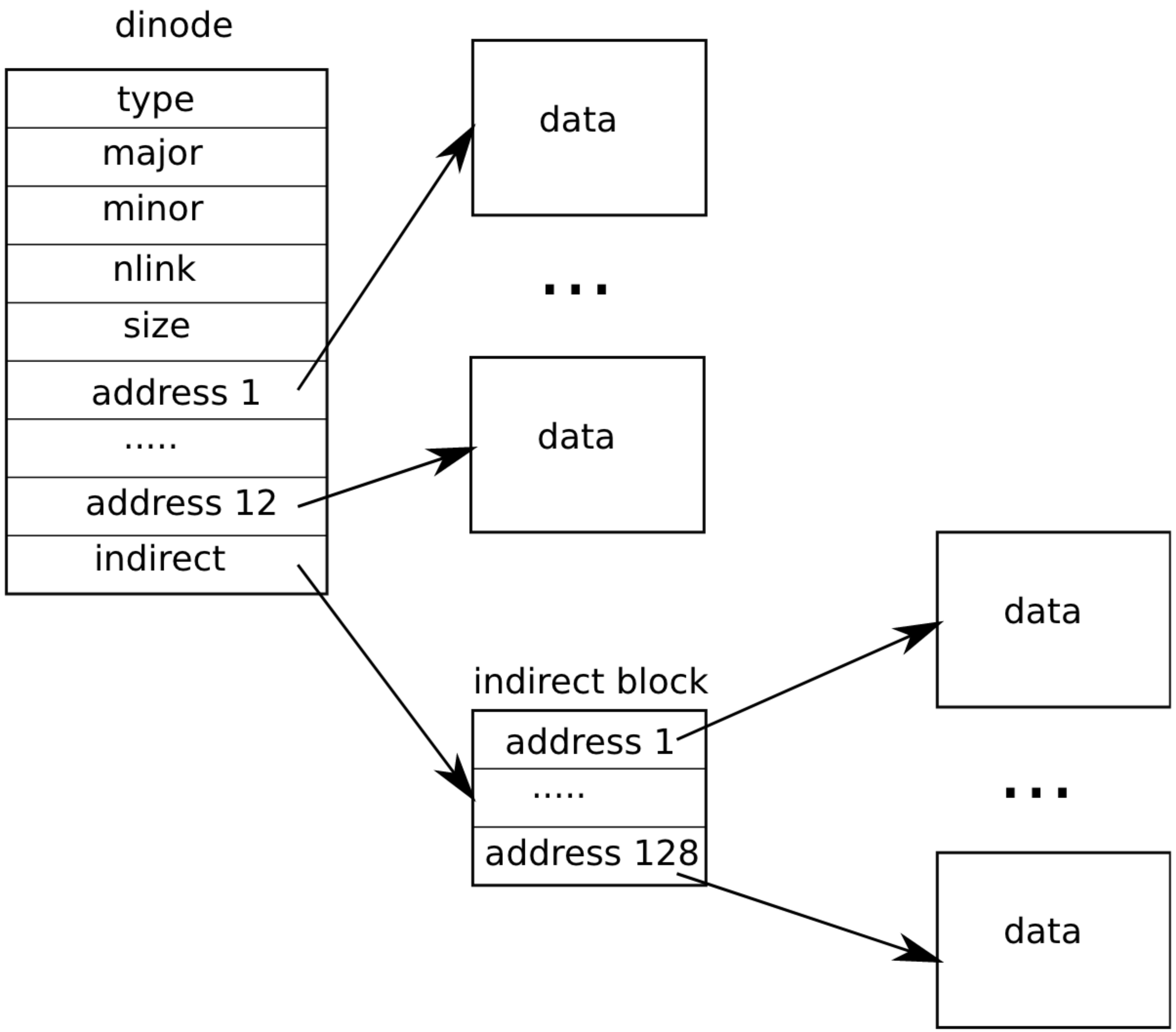
Inode = 9 "/foo/bar"

	Type = file	
	Size	
	Access time	13
5	3	
	31	7
9	5	
	12	11

Block 3	Block 31	Block 5	Block 12
6	8	10	12
hel	lo	wor	ld

Storing large files

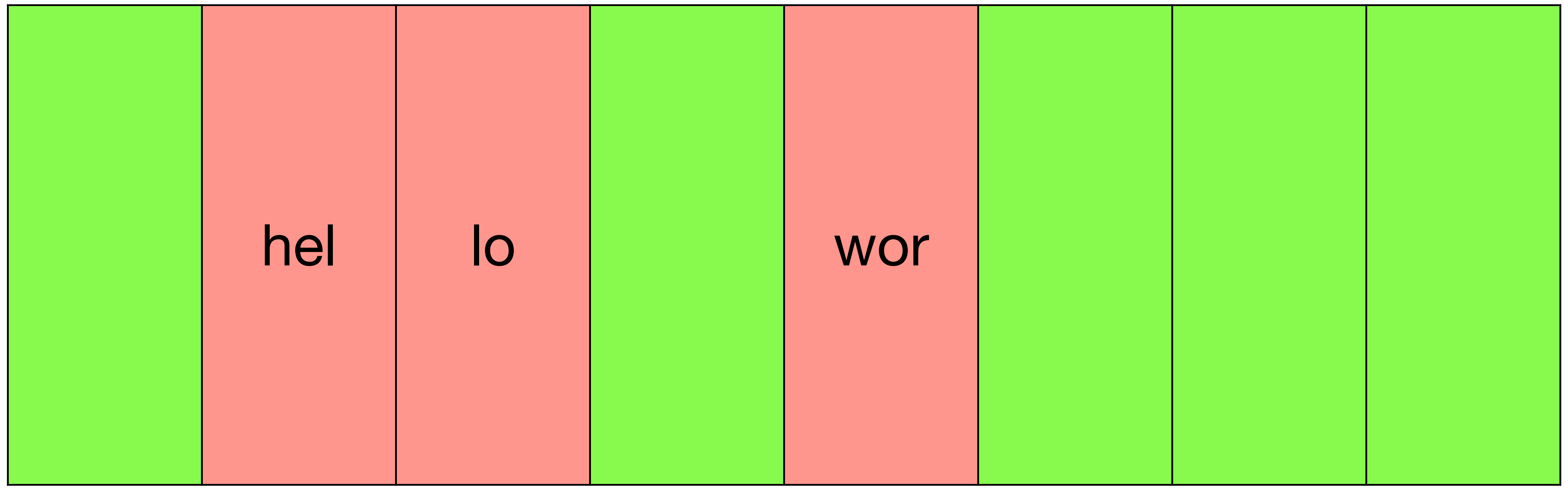
- Keep inodes of fixed size (64 bytes) for simplicity
- 8 inodes in a 512 byte block
- Most files are < 2KB
- 12*512 bytes = 6KB
- Most files do not need indirect block



How to track free blocks?

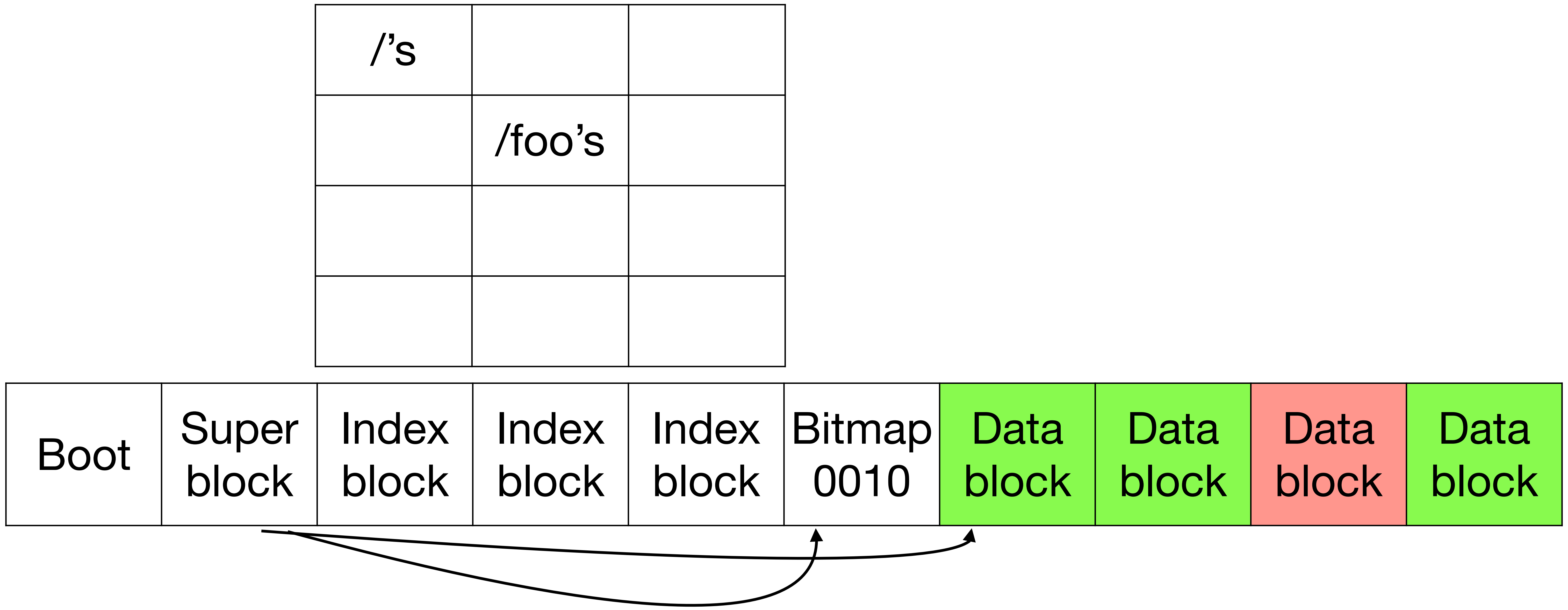
- Keep bitmap in another block

10010111



Putting it all together: xv6 FS organisation

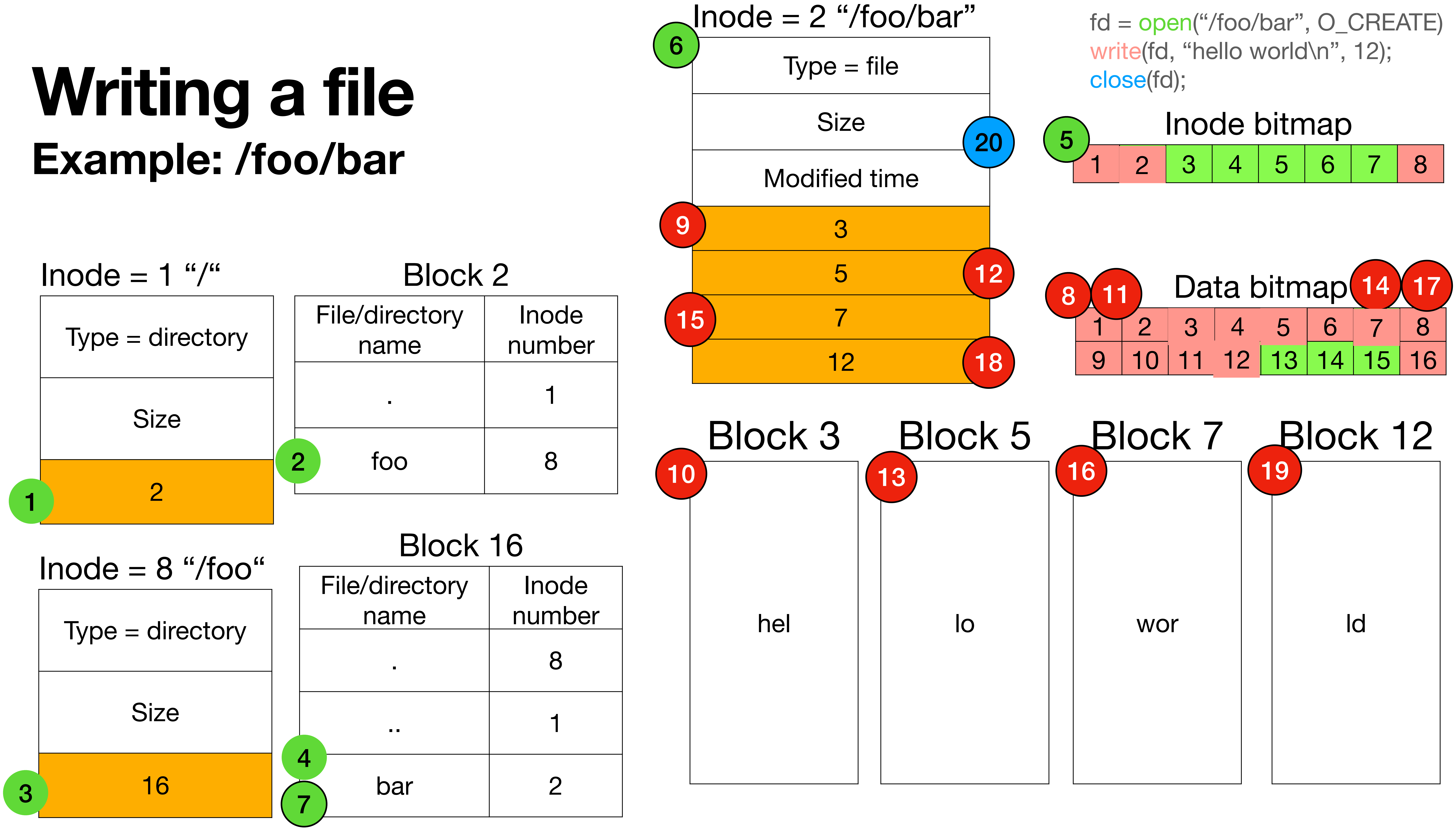
- Data region contains actual file and directory data
- File system structure is maintained via nodes stored in index blocks
- Superblock contains file system metadata:
  - how many inodes are in system, etc



Writing a file

Example: /foo/bar

```
fd = open("/foo/bar", O_CREATE)
write(fd, "hello world\n", 12);
close(fd);
```



Xv6 code walkthrough

p8-read-fs

- fs.h
  - ROOTINO=1: root folder is at the first inode
  - struct superblock
  - NDIRECT: 12 direct pointers. NINDIRECT: Number of pointers that can fit in the second-level pointer node (128).
  - MAXFILE: maximum number of data blocks (140). Max file size is 70 KB
  - struct dinode (16 bytes). IPB = 32.
  - struct dirent. 16 bytes. 32 directory entries in one data block of directory
- Makefile, mkfs.c creates a disk image with the file system containing one “/welcome.txt” file
- main.c reads and prints contents from welcome.txt

File System Optimizations

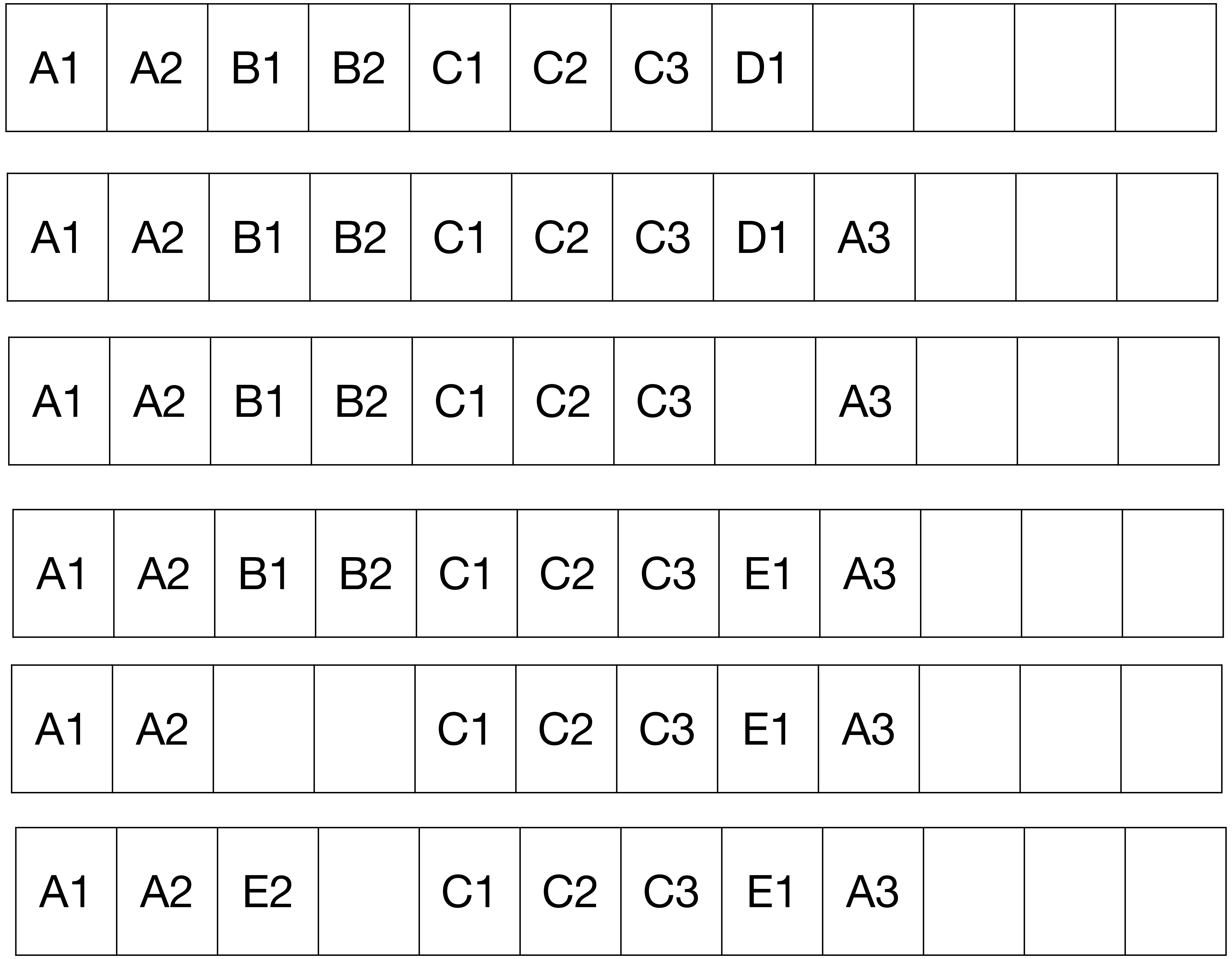
OSTEP Ch.41

Performance problems

- Fragmentation
- Poor locality
- Poor use of the buffer cache
- Minimal disk scheduling opportunities

Fragmentation problem

- Over time, a file’s data blocks get spread all over the disk
    - Disk head(s) need to go back and forth to read files sequentially



Fragmentation problem

Defragmentation

- Defragmenter rearranges data blocks
  - Also updates data block pointers in file’s inode
- Modern FS such as ext4 do defragmentation in background: without making FS unavailable

A1	A2	E2		C1	C2	C3	E1	A3			

A1	A2	A3	C1	C2	C3	E1	E2				

Fragmentation problem

Pre-allocate blocks

- Disks have grown bigger
    - Ext3 pre-allocates 8 blocks at file creation
- Reduce metadata lookup overhead by keeping extents

A1	A2		B1	B2		C1	C2		D1		
A1	A2	A3	B1	B2		C1	C2		D1		
A1	A2	A3	B1	B2		C1	C2				
A1	A2	A3	B1	B2		C1	C2		E1		
A1	A2	A3				C1	C2		E1		
A1	A2	A3				C1	C2		E1	E2	
A1	A2	A3	A4			C1	C2		E1	E2	

A
1
2
3
4

A
1, 4