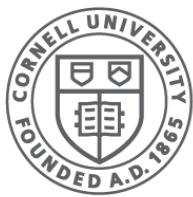


File Systems

CS 4410
Operating Systems



Cornell CIS
COMPUTING AND INFORMATION SCIENCE

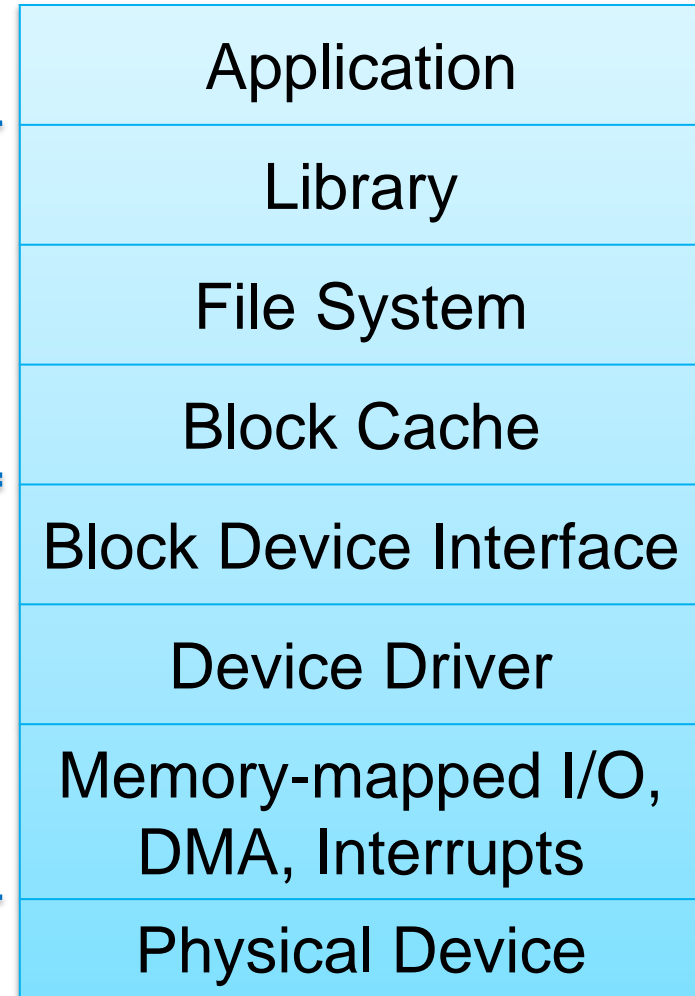
[R. Agarwal, L. Alvisi, A. Bracy, M. George, E. Sirer, R. Van Renesse]

The abstraction stack

I/O systems are accessed through a series of layered abstractions

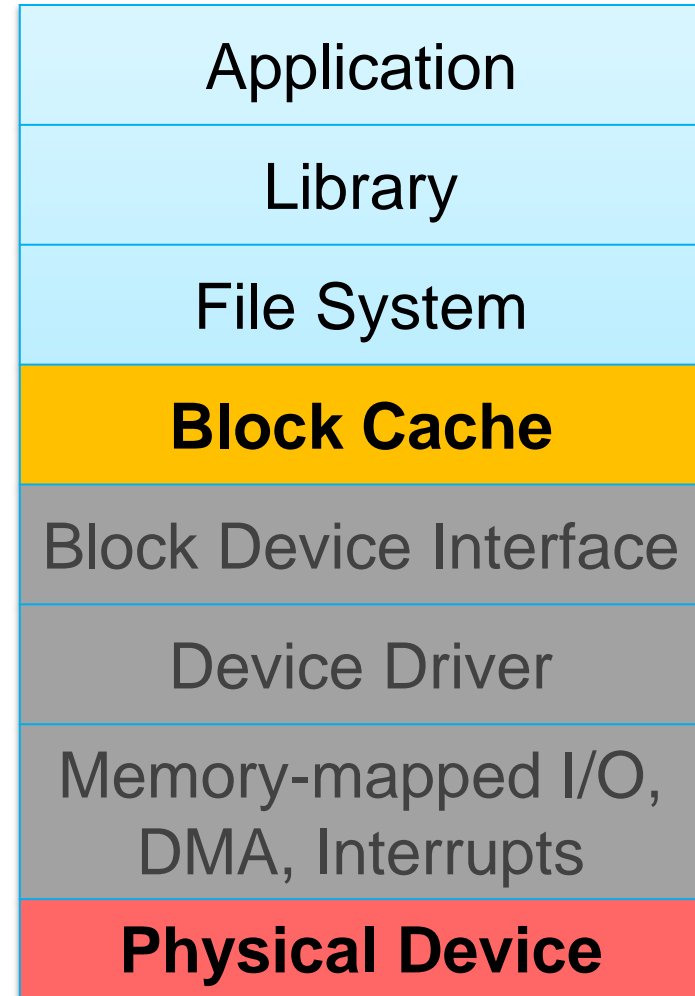
File System API
& Performance

Device
Access



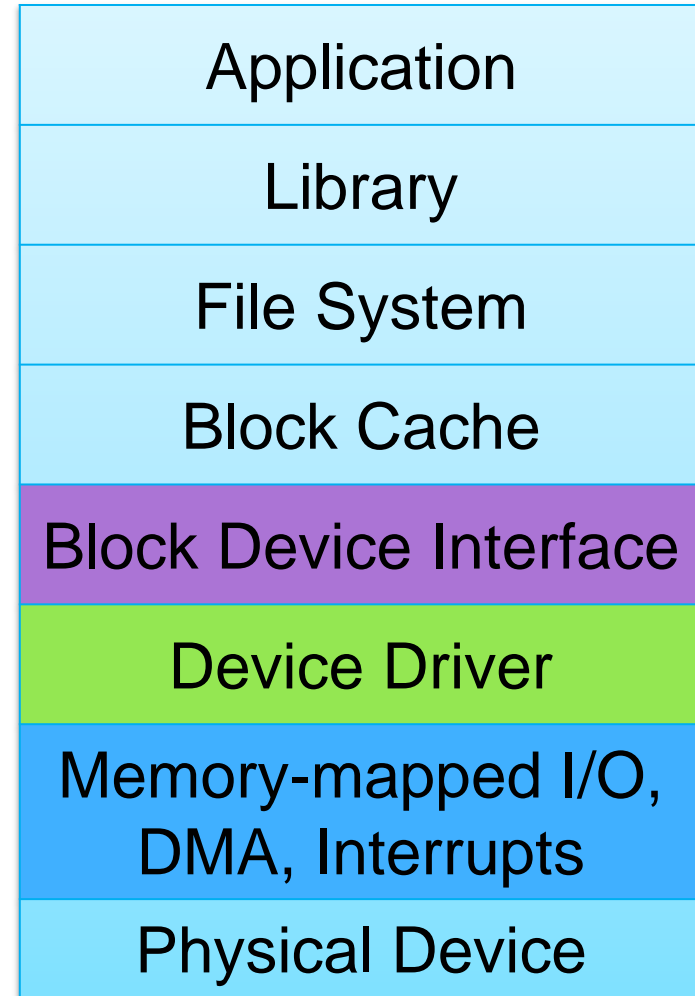
The Block Cache

- a **cache** for the **disk**
- caches recently read blocks
- buffers recently written blocks
- serves as synchronization point (ensures a block is only fetched once)



More Layers *(not a 4410 focus)*

- allows data to be read or written in fixed-sized blocks
- uniform interface to disparate devices
- translate between OS abstractions and hw-specific details of I/O devices
- Control registers, bulk data transfer, OS notifications



Where shall we store our data?

Process Memory? (*why is this a bad idea?*)

- Size is limited to size of virtual address space
- Data lost when the application terminates
- Even if the computer doesn't crash!
- Multiple processes might want to access the same data

File Systems 101

Long-term Information Storage Needs

- large amounts of information
- information must survive processes
- need concurrent access by multiple processes

Solution: the File System Abstraction

- Presents applications w/ **persistent, named** data
- Two main components:
 - Files
 - Directories

The File Abstraction

- **File:** a named collection of data
- has two parts
 - **data** – what a user or application puts in it
 - array of untyped bytes
 - **metadata** – information added and managed by the OS
 - size, owner, security info, modification time

First things first: Name the File!

1. Files are abstracted unit of information
 2. Don't care exactly where *on disk* the file is
- Files have human readable names
- file given name upon creation
 - use the name to access the file

Name + Extension

Naming Conventions

- Some things OS dependent:
 - Windows not case sensitive, UNIX is
- Some things common:
 - Usually ok up to 255 characters

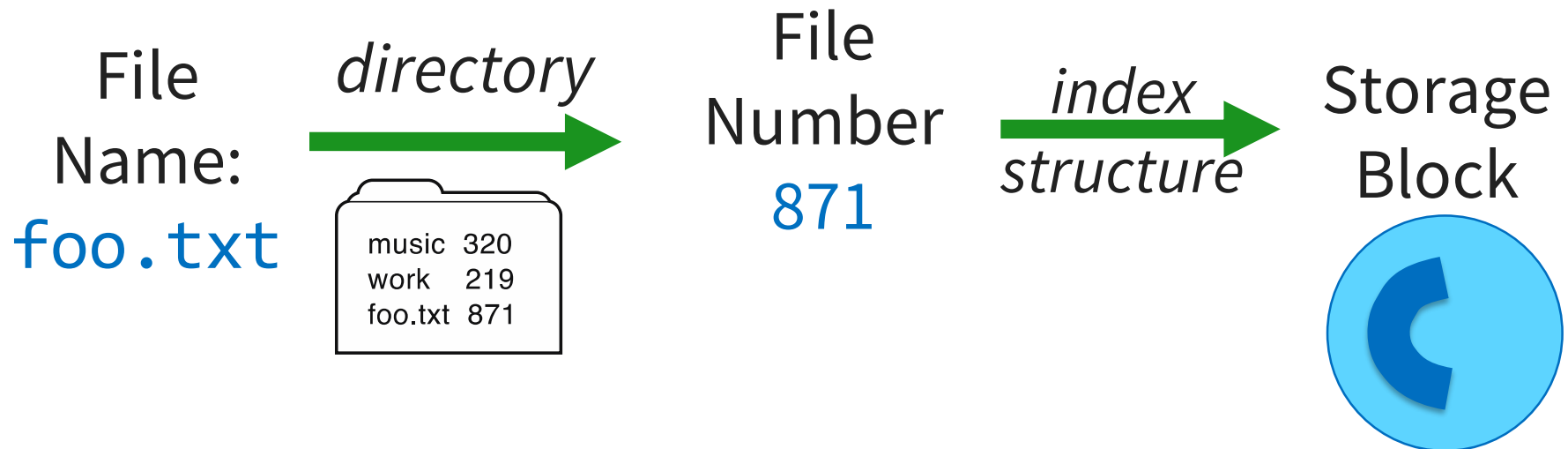
File Extensions, OS dependent:

- Windows:
 - attaches meaning to extensions
 - associates applications to extensions
- UNIX:
 - extensions not enforced by OS
 - Some apps might insist upon them (.c, .h, .o, .s, for C compiler)

Directory

Directory: provides names for files

- a list of human readable names
- a mapping from each name to a specific underlying file or directory



Path Names

Absolute: path of file from the root directory

`/home/ada/projects/babbage.txt`

Relative: path from the current working directory
(current working dir stored in process' PCB)

2 special entries in each UNIX directory:

“.” current dir

“..” for parent

To access a file:

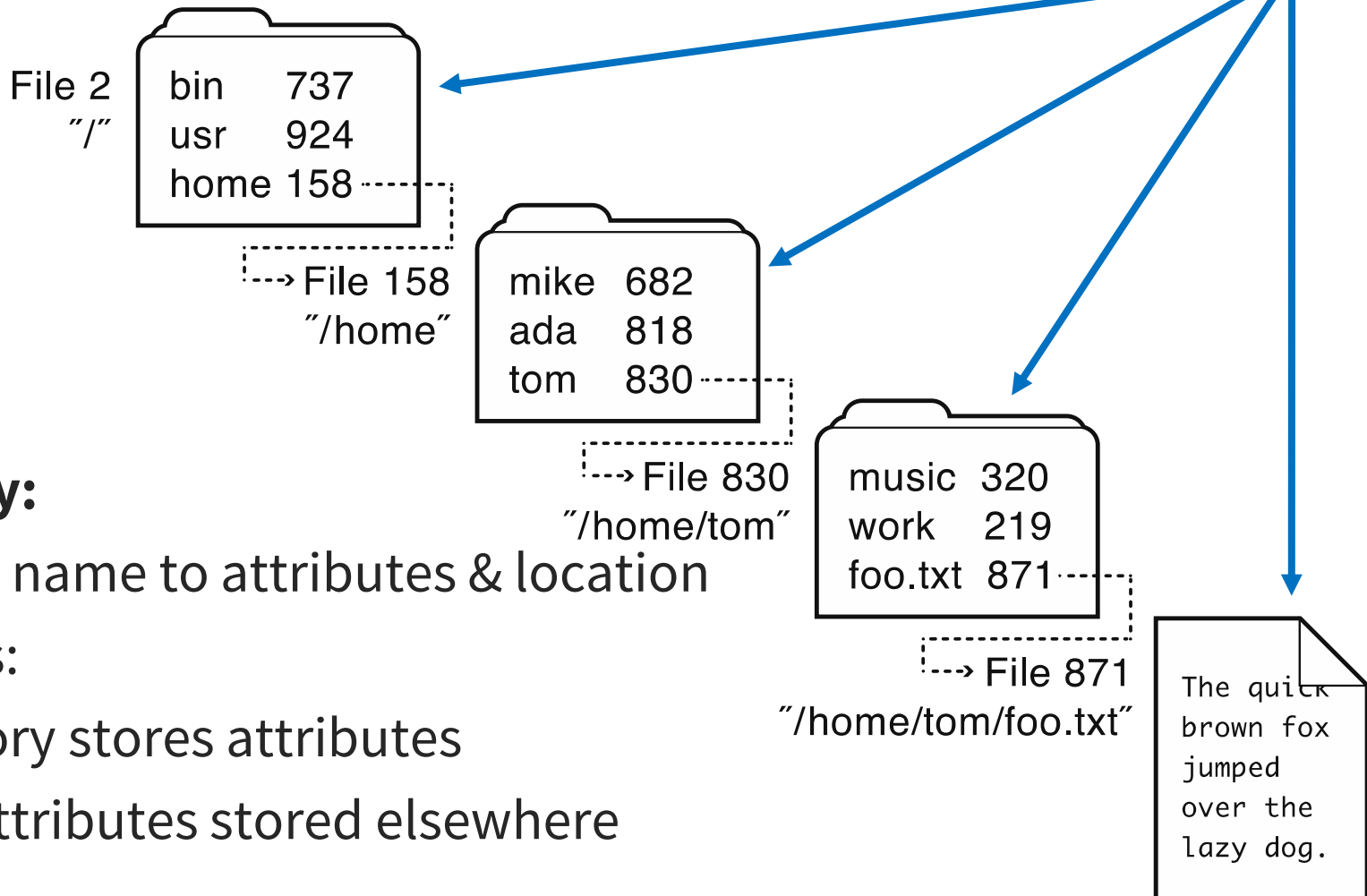
- Go to the folder where file resides —OR—
- Specify the path where the file is

Directories

OS uses path name to find directory

Example: `/home/tom/foo.txt`

all files



Directory:

maps file name to attributes & location

2 options:

- directory stores attributes
- files' attributes stored elsewhere

Basic 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

How shall we implement this?

Just map **keys** (file names) to **values** (block numbers on disk)?

Challenges for File System Designers

Performance: despite limitations of disks

- leverage spatial locality

Flexibility: need jacks-of-all-trades, diverse workloads, not just FS for X

Persistence: maintain/update user data + internal data structures on persistent storage devices

Reliability: must store data for long periods of time, despite OS crashes or HW malfunctions

Implementation Basics

Directories

- file name → file number

Index structures

- file number → block

Free space maps

- find a free block; better: find a free block *nearby*

Locality heuristics

- policies enabled by above mechanisms
 - group directories
 - make writes sequential
 - defragment

File System Properties

Most files are small

- need strong support for small files
- block size can't be too big

Some files are very large

- must allow large files
- large file access should be reasonably efficient

Storing Files

Files can be allocated in different ways:

- **Contiguous allocation**
All bytes together, in order
- **Linked Structure**
Each block points to the next block
- **Indexed Structure**
Index block points to many other blocks

Which is best?

- For sequential access? Random access?
- Large files? Small files? Mixed?



Contiguous Allocation

All bytes together, in order

+ **Simple:** state required per file: start block & size

+ **Efficient:** entire file can be read with one seek

- **Fragmentation:** external is bigger problem

- **Usability:** user needs to know size of file at time of creation



Used in CD-ROMs, DVDs

Linked List Allocation

Each file is stored as linked list of blocks

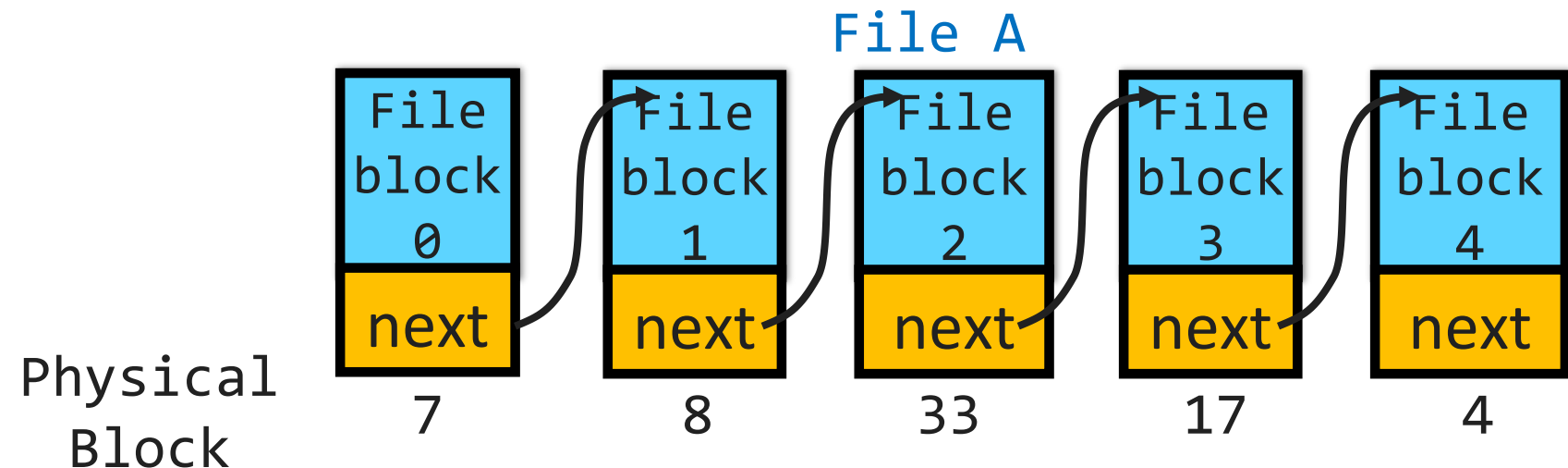
- First word of each block points to next block
- Rest of disk block is file data

+ **Space Utilization:** no space lost to external fragmentation

+ **Simple:** only need to store 1st block of each file

- **Performance:** random access is slow

- **Space Utilization:** overhead of pointers



File Allocation Table (FAT) FS

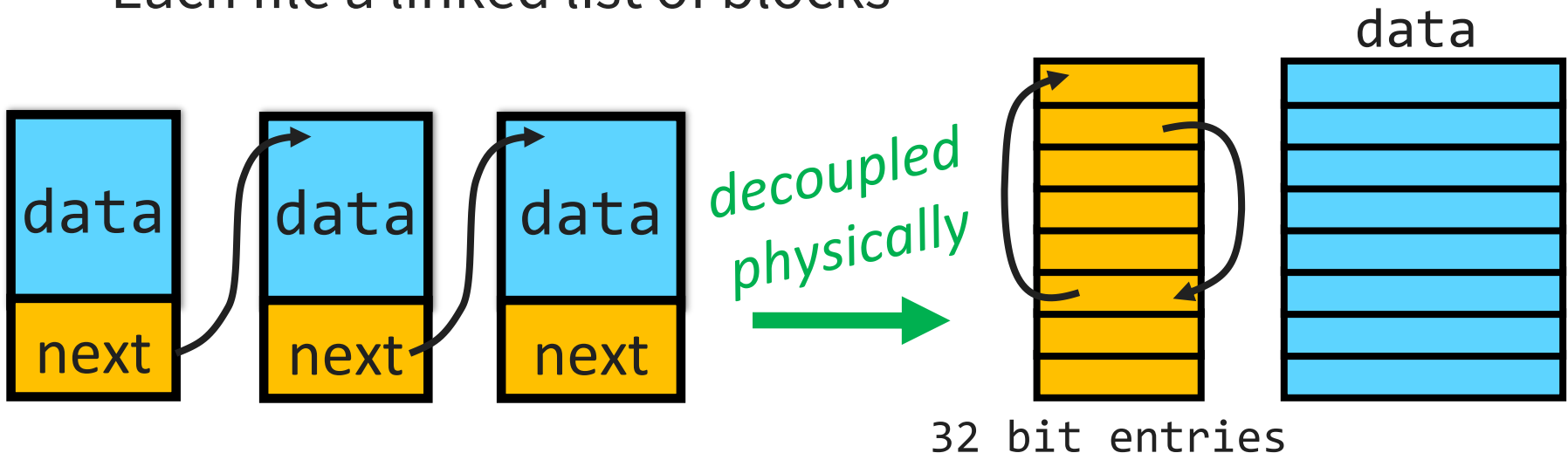
[late 70's]

Microsoft File Allocation Table

- originally: MS-DOS, early version of Windows
- today: still widely used (e.g., CD-ROMs, thumb drives, camera cards)
- FAT-32, supports 2^{28} blocks and files of $2^{32}-1$ bytes

File table:

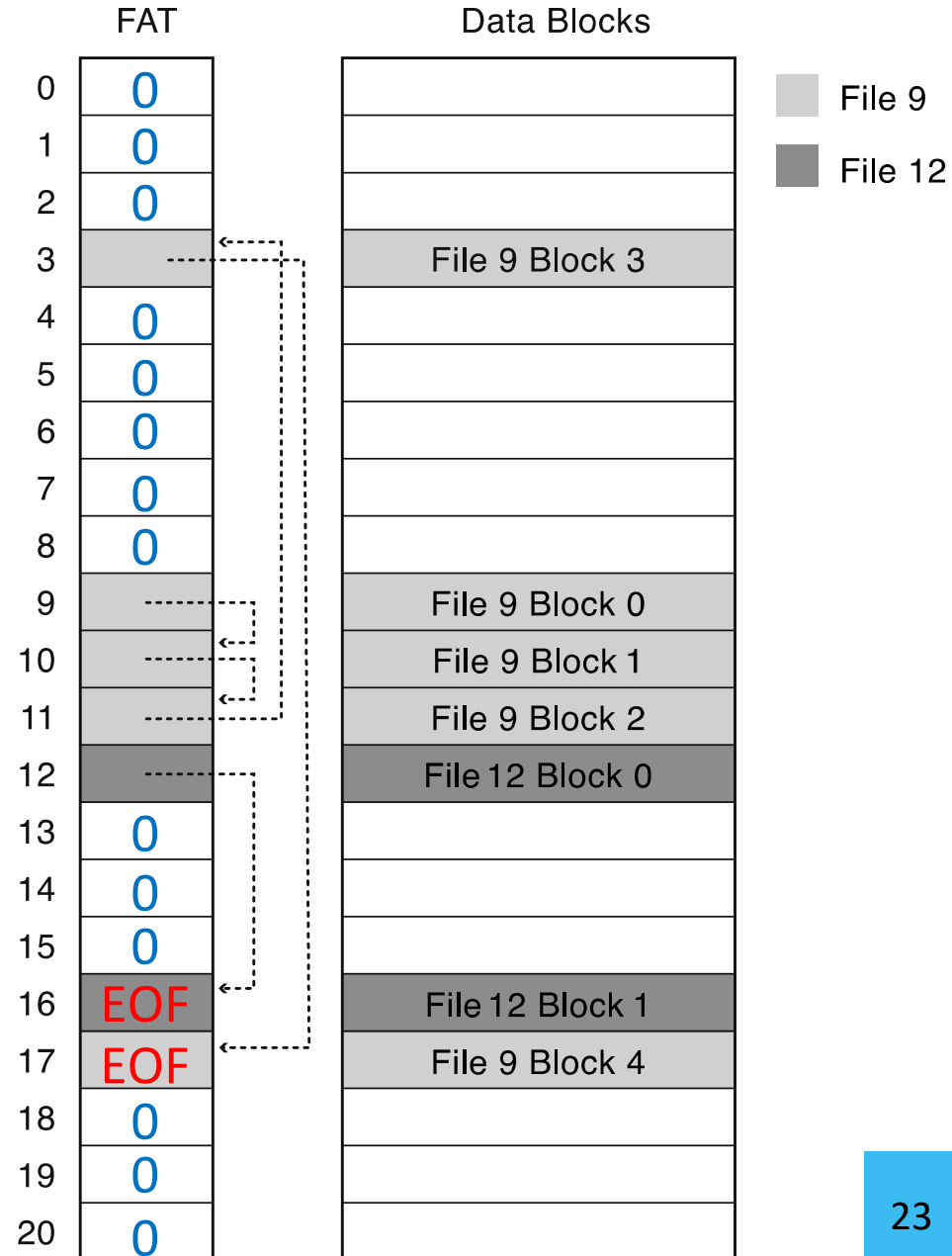
- Linear map of all blocks on disk
- Each file a linked list of blocks



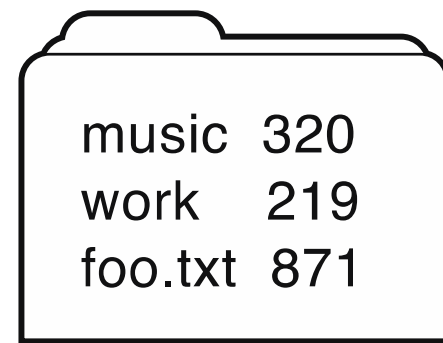
FAT File System

- 1 entry per block
- **EOF** for last block
- **0** indicates free block
- directory entry maps name to FAT index

Directory	
bart.txt	9
maggie.txt	12



FAT Directory Structure



music	320
work	219
foo.txt	871

Folder: a file with 32-byte entries

Each Entry:

- 8 byte name + 3 byte extension (ASCII)
- creation date and time
- last modification date and time
- first block in the file (index into FAT)
- size of the file
- Long and Unicode file names take up multiple entries

How is FAT Good?

- + Simple: state required per file: start block only
- + Widely supported
- + No external fragmentation
- + block used only for data

How is FAT Bad?

- Poor locality
- Many file seeks unless entire FAT in memory:
Example: 1TB (2^{40} bytes) disk, 4KB (2^{12}) block size, FAT has 256 million (2^{28}) entries (!)
4 bytes per entry \rightarrow 1GB (2^{30}) of main memory required for FS (a sizeable overhead)
- Poor random access
- Limited metadata
- Limited access control
- Limitations on volume and file size
- No support for reliability techniques