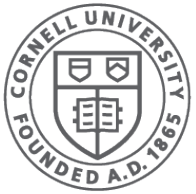
File Systems

(Chapters 39-43,45)

CS 4410

Operating Systems



Cornell CIS

COMPUTING AND INFORMATION SCIENCE

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

Storage Devices: Recap

- Disks
  - RAID-0, 1, 4, 5
- Solid State Drives (Flash memory)

Characteristics: RAM but …

- Access latency
  - seek, rotational delay
- Read / write xfer speeds

Storage Device Use: File System

Goals

- scale
- persistence
- access by multiple processes

File System

Interface provides operations involving:

- Files
- Directories (a special kind of file)

The File Abstraction

A **file** is a named assembly of data.

- Each file comprises:
  - **data** – information a user or application stores
    - array of untyped bytes
    - implemented by an array of fixed-size blocks
  - **metadata** – information added / managed by OS
    - size, owner, security info, modification time, etc.

File Names

Files have names:

- a unique **low-level name**
  - low-level name is distinct from location where file stored
    - ☞ *File system provides mapping from low-level names to storage locations.*
- one or more human-readable names
  - ☞ *File system provides mapping from human-readable names to low-level names.*

File Names (con't)

Naming conventions

- Some aspects of names are OS dependent:

  Windows is not case sensitive, UNIX is.

- Some aspects are not:

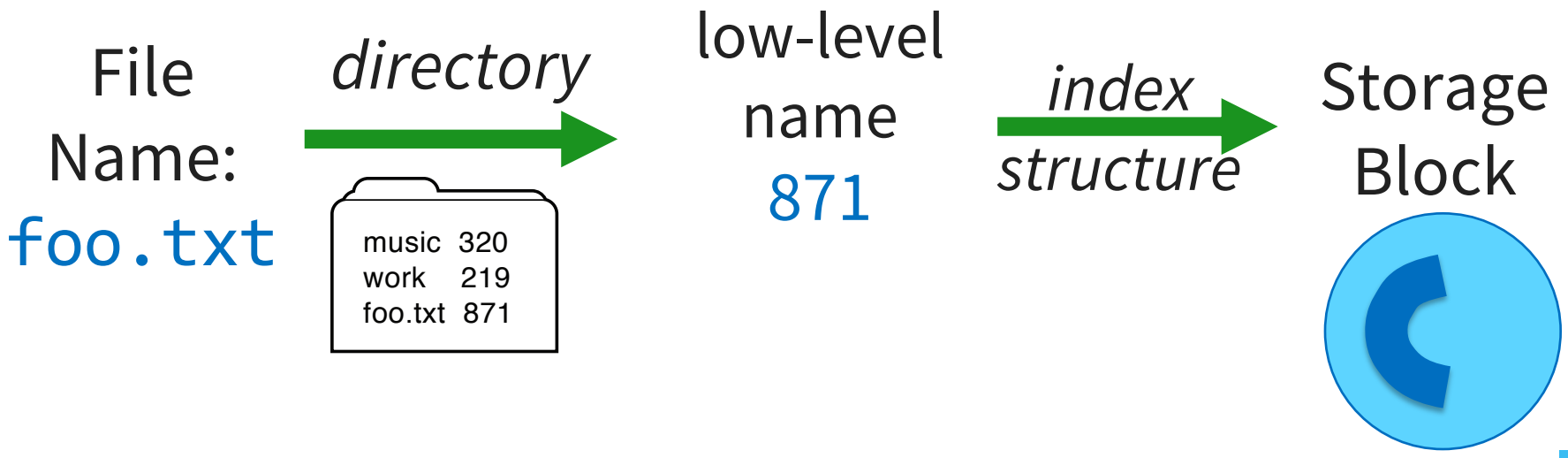
  Names up to 255 characters long

File name extensions are widespread:

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

Directories

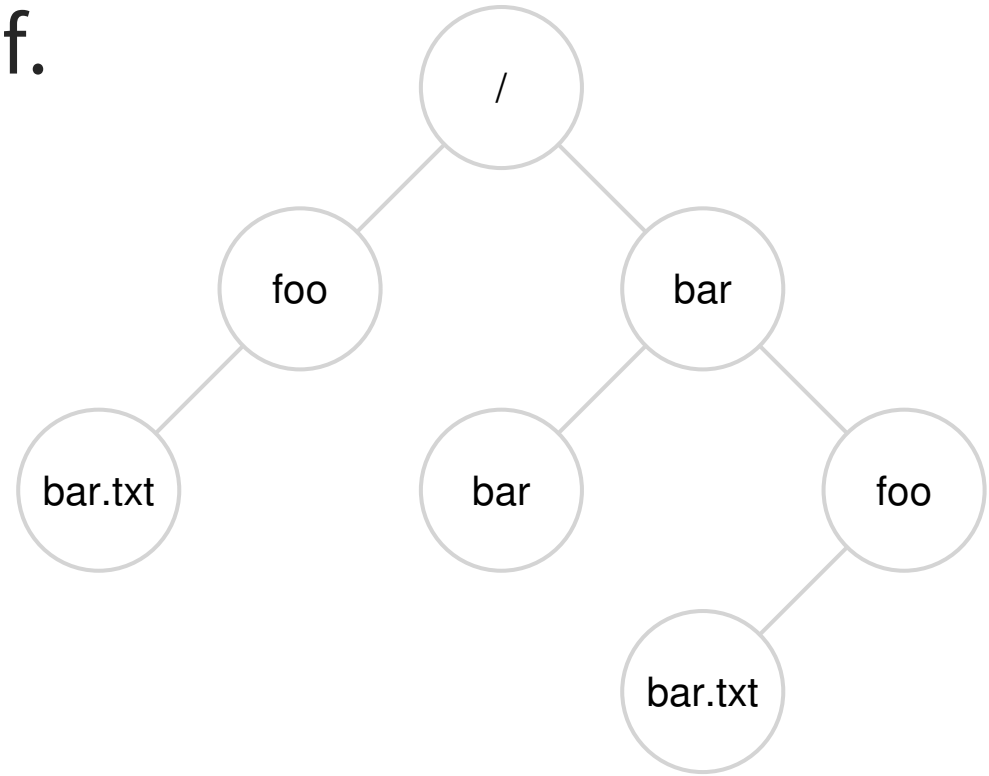
Directory: A file whose interpretation is a mapping from a **character string** to a **low level** name.



Directories Compose into Trees

Each path from root
is a name for a leaf.

/foo/bar.txt
/bar/bar
/bar/foo/bar.txt



Paths as Names

Absolute: path of file from the root directory

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

Relative: path from the current working directory

`projects/babbage.txt`

(N.b. Current working dir stored in process PCB)

2 special entries in each UNIX directory:

"." this dir

".." for parent of this dir (except .. for "/" (root) is "/")

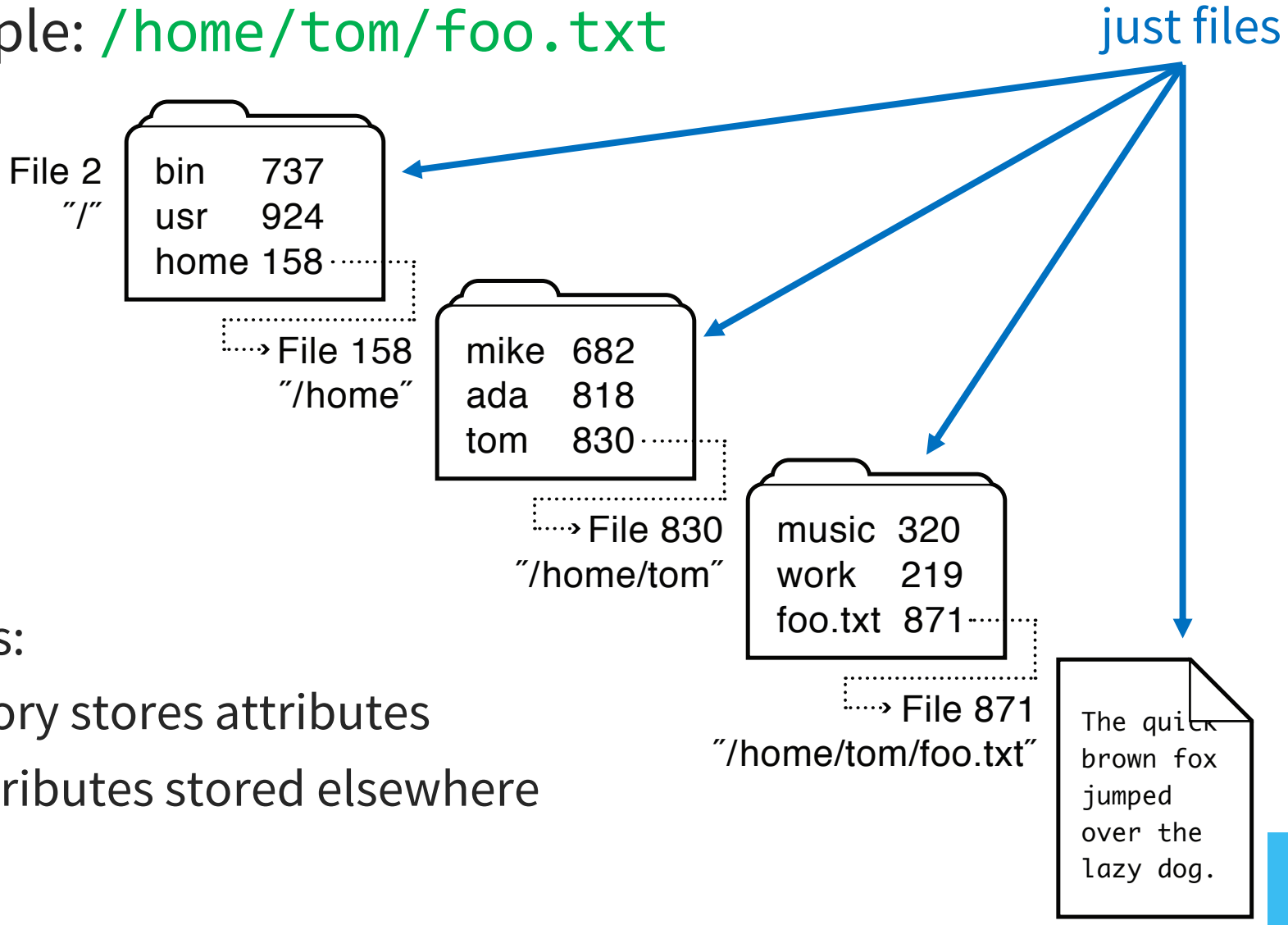
To access a file:

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

Paths as Names (con't)

OS uses path name to identify a file

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



2 options:

- directory stores attributes
- file attributes stored elsewhere

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

File System Design Challenges

Performance: Overcome limitations of disks

- leverage spatial locality to avoid seeks and to transfer block sequences.

Flexibility: Handle diverse application workloads

Persistence: Storage for long term.

Reliability: Resilient to OS crashes and HW failure

Implementation Basics: Mappings

Mappings:

- Directories: file name → low-level name
- Index structures: low-level name→ block
- Free space maps: locate free blocks (near each other)

To exploit locality of file references:

- Group directories together on disk
- Prefer (large) sequential writes/reads
- Defragmentation: Relocation of blocks:
  - Blocks for a file appear on disk in sequence
  - Files for directories appear near each other

Workload Overview (circa 2002-7)

File size is bimodal:

- Most files are small (2K is most common size).
  - to support small files: use small block size or pack multiple file blocks (.5K) within a single disk block (4K).
- Some files are very large.
  - to support large files: prefer trees to lists

Files systems are roughly ½ full.

- …even as disks get larger.

Directories are typically small (20 or fewer entries).

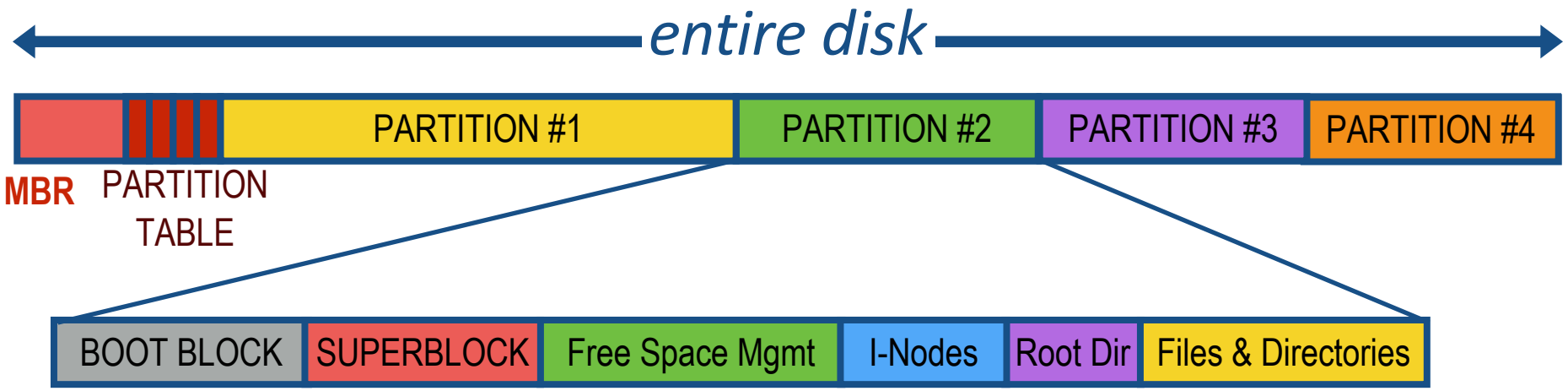
Average file size is growing (200K in 2007).

Agrawal, Bolosky, Douceur, Lorch. A Five Year Study of File-System Metadata. FAST'07, San Jose CA.

Disk Layout

File System is stored on *disks*

- sector 0 of disk called Master Boot Record (MBR)
- end of MBR: partition table (partitions' start & end addrs)
- Remainder of disk divided into *partitions*.
  - Each partition starts with a *boot block*
  - Boot block loaded by MBR and executed on boot
  - Remainder of partition stores file system.



File Storage Layout Options

- Contiguous allocation

  All bytes together, in order

- Linked-list

  Each block points to the next block

- Indexed structure

  Index block points to many other blocks

- Log structure

  Sequence of segments, each containing updated blocks

Which is best? It depends…

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

Contiguous Allocation

All bytes of file are stored together, in order.

+ **Simple:** state required per file: start block & size
+ **Efficient:** entire file can be read with one seek
– **Fragmentation:** external fragmentation is bigger problem
– **Usability:** user needs to know size of file at time of creation

Used in CD-ROMs, DVDs

Linked-List File Storage

Each file is stored as linked list of blocks

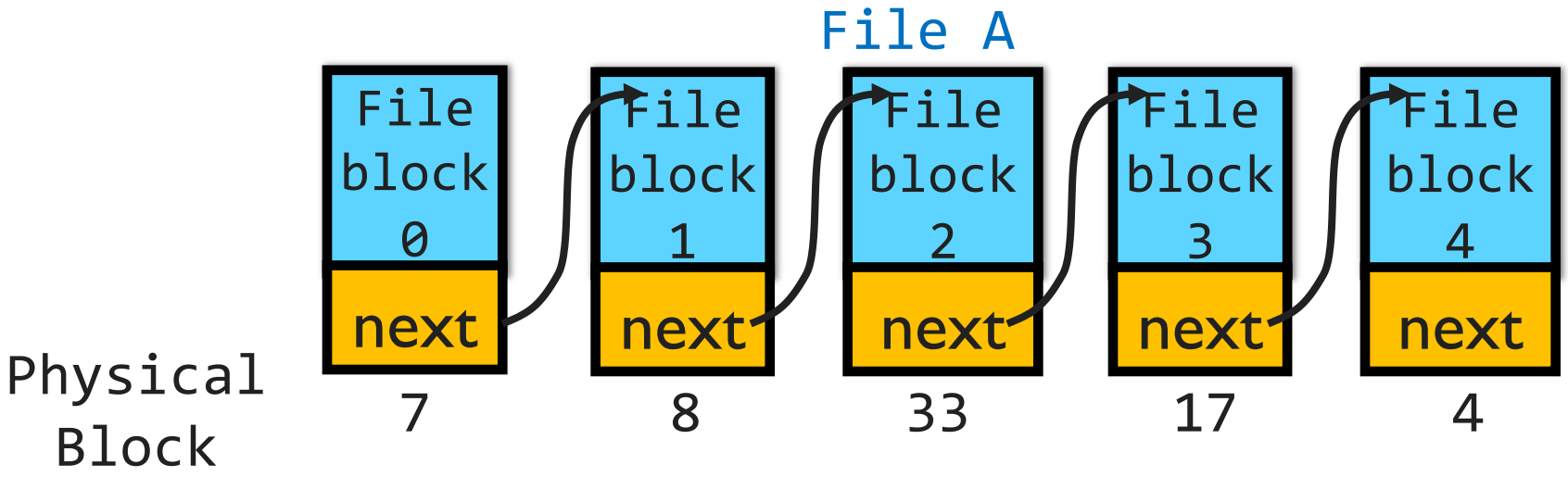
- First word of each block points to next block
- Rest of disk block is file data

+ 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



Linked List File System

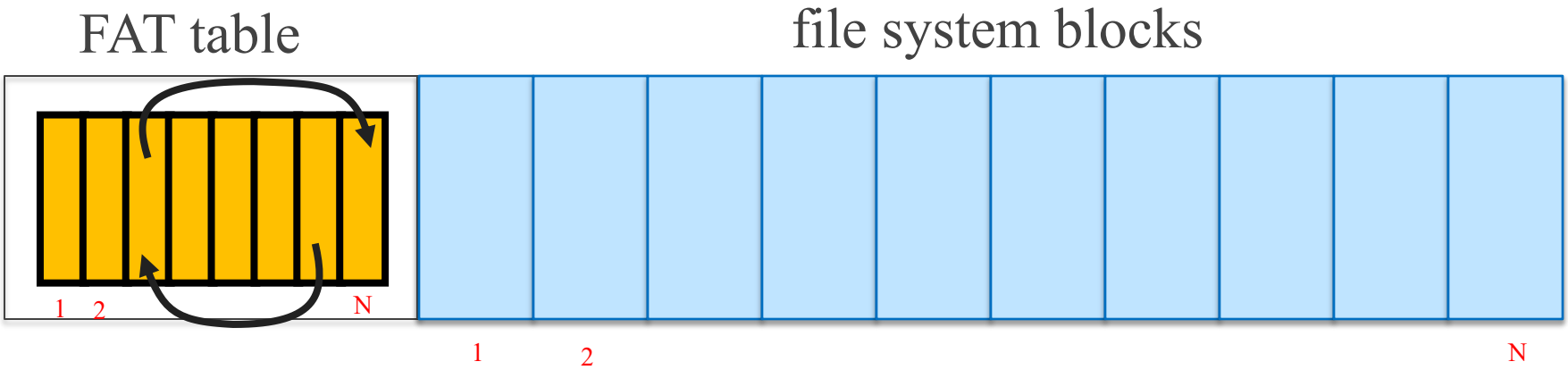
File Allocation Table (FAT)

- Used in MS-DOS, precursor of Windows
- Still used (e.g., CD-ROMs, thumb drives, camera cards)
- FAT-32, supports 2^{28} blocks and files of 2^{32}-1 bytes

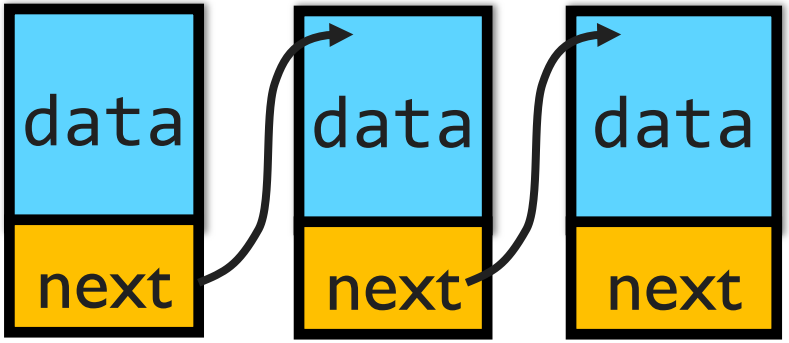
FAT (is stored on disk):

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

FAT File System



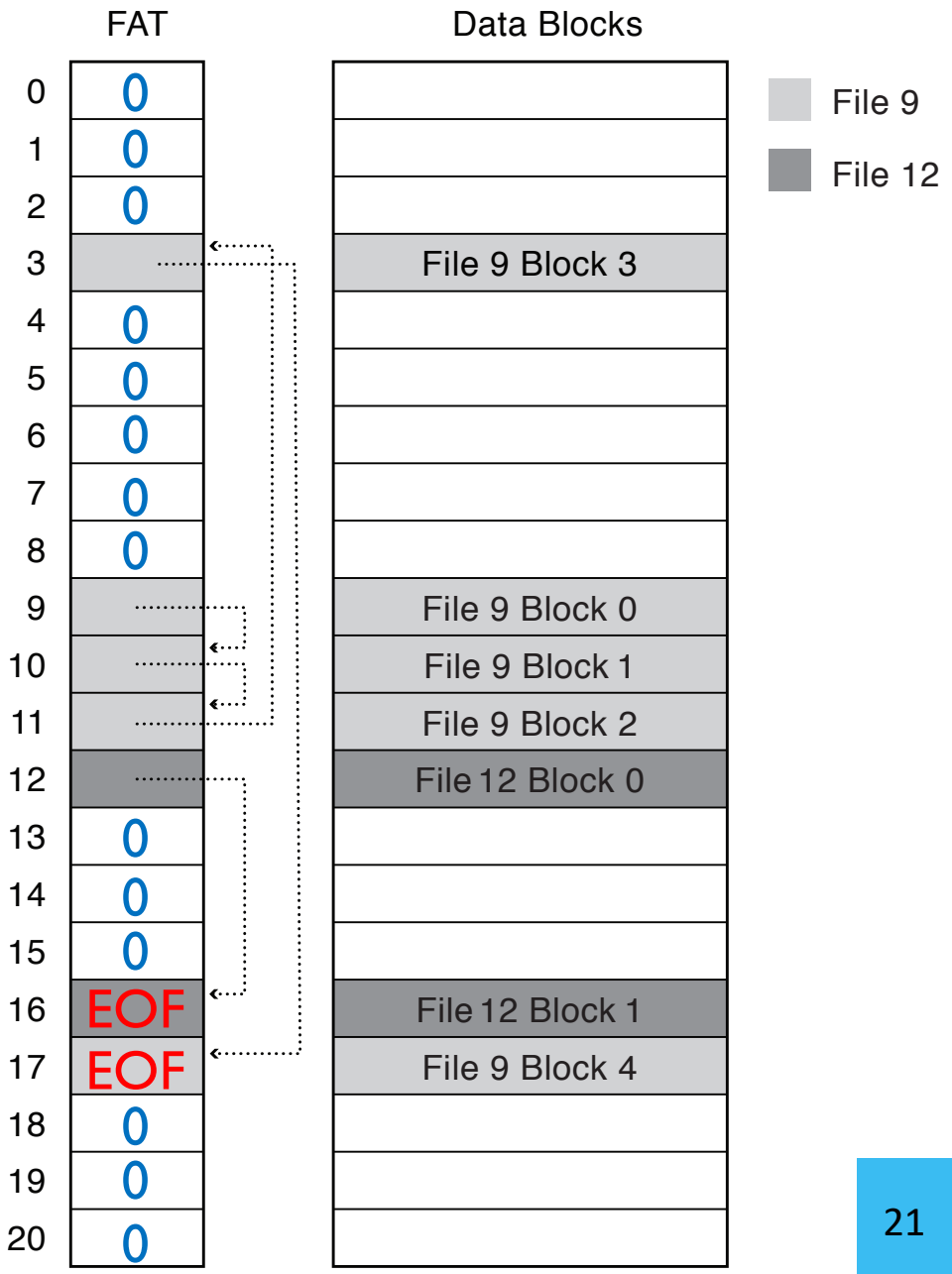
implements



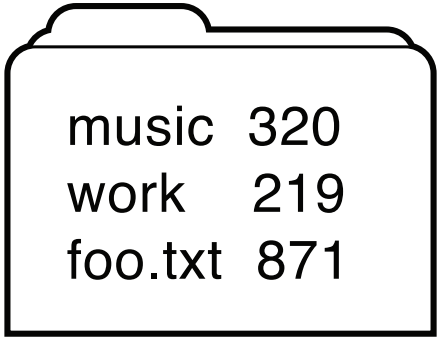
FAT File System

- 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



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 → 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