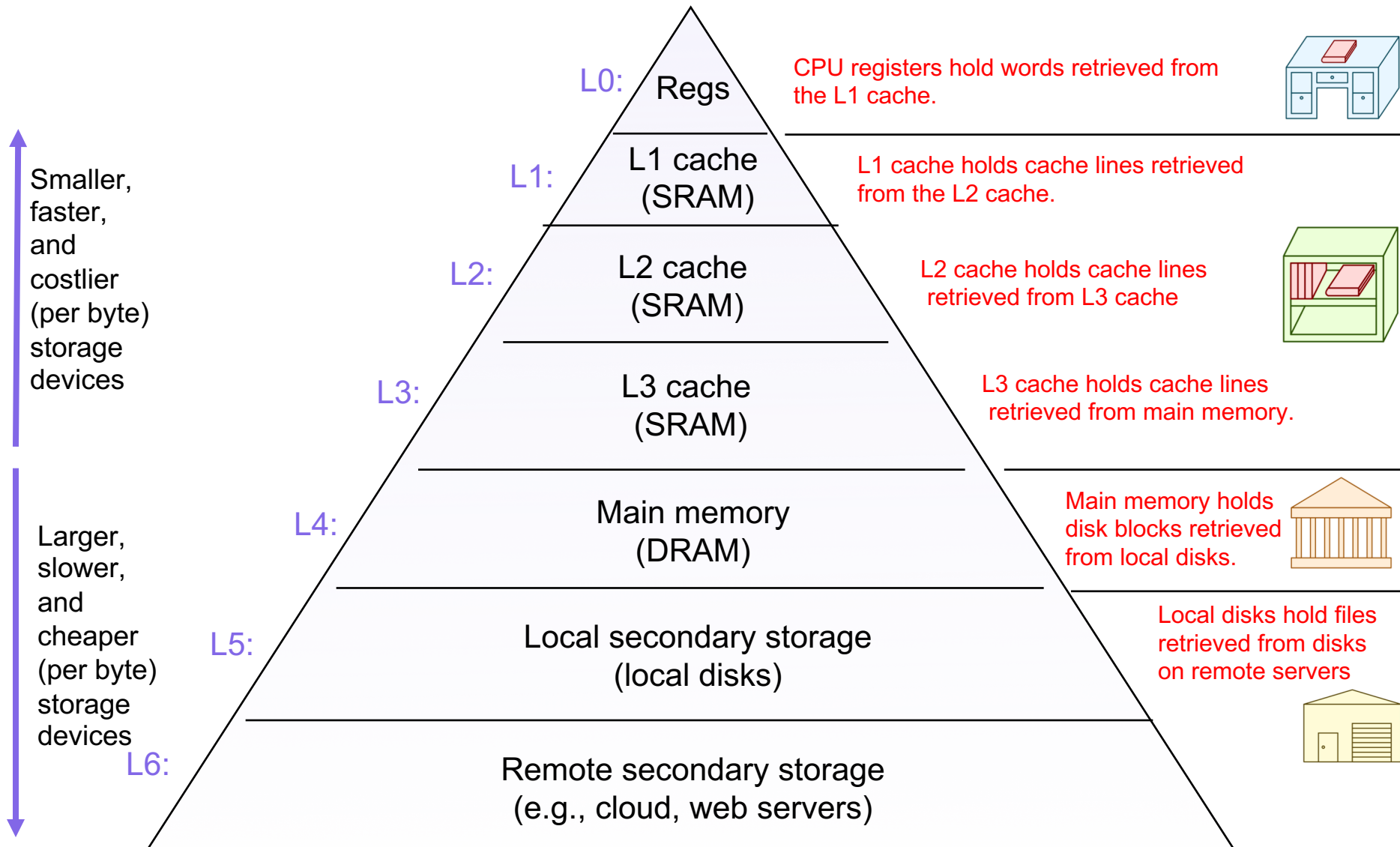


Lecture 21: File Systems

CS 105

November 21, 2019

Memory Hierarchy



Storage Devices

- Magnetic Disks

- Storage that rarely becomes corrupted
- Large capacity at low cost
- Block-level random access
- Slow performance for random access
- Better performance for streaming access

- Solid State Disks (Flash Memory)

- Storage that rarely becomes corrupted
- Capacity at moderate cost (50x magnetic disk)
- Block-level random access
- Good performance for random reads
- Not-as-good performance for random writes



1950s
IBM 350
5 MB



2019
WD Red
10 TB



2019
Samsung 840
250 GB

Comparing Storage Media

	RAM	HDD	SSD
Typical Size	8 GB	1 TB	256 GB
Cost	\$10 per GB	\$0.05 per GB	\$0.32 per GB
Power	3 W	2.5 W	1.5 W
Read Latency	15 ns	15 ms	30 μ s
Read Speed (Seq.)	8000 MB/s	175 MB/s	550 MB/s
Read/Write Granularity	word	sector	page*
Power Reliance	volatile	non-volatile	non-volatile

File Systems 101

- Long-term information storage goals
 - should be able to store large amounts of information
 - information must survive processes, power failures, etc.
 - processes must be able to find information
 - needs to support concurrent accesses by multiple processes
- Solution: the File System Abstraction
 - interface that provides operations involving
 - files
 - directories (a special kind of file)

The File Abstraction

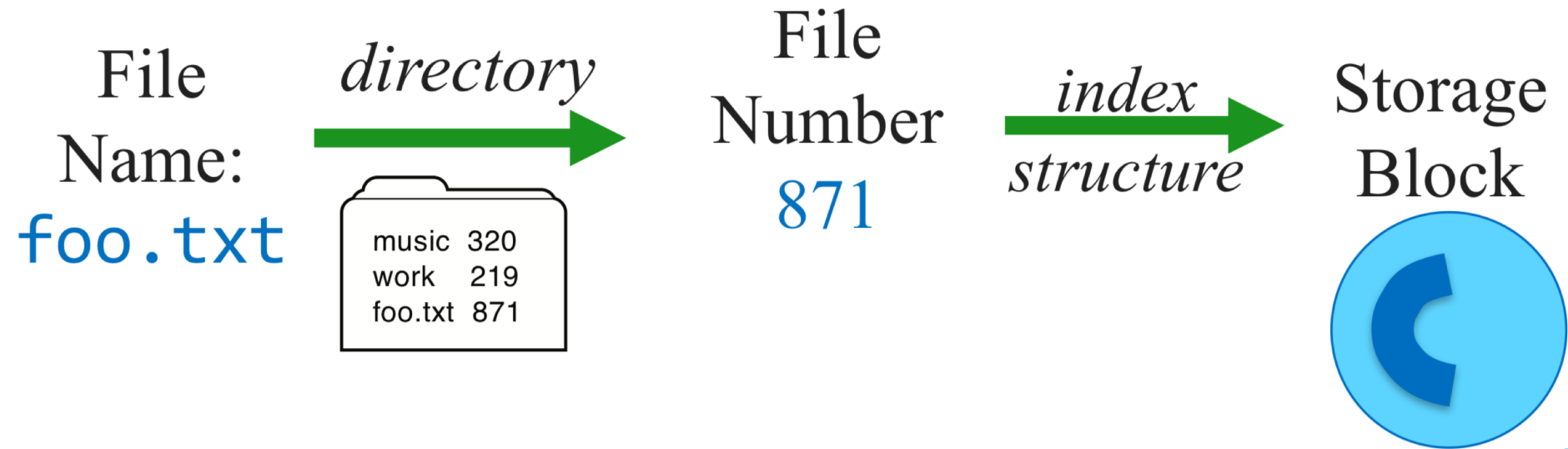
- a **file** is a named collection of data
 - name is defined on creation
 - processes use name to subsequently access that file
- a file is comprised of two parts:
 - **data**: information a user or application puts in a file
 - an array of untyped bytes
 - implemented as an array of fixed-size blocks
 - **metadata**: information added and managed by the OS
 - e.g., size, owner, security info, modification time

File Names

- Each file has a unique low-level name
 - distinct from location; processes don't care where on disk a file is stored
 - file system provides mapping from low-level names to storage location
- Each file has one or more human-readable names
 - file system provides mapping from human-readable names to low-level names
- Naming conventions
 - up to 255 characters long
 - case sensitive (UNIX) or not case sensitive (Windows)
 - extensions not enforced (UNIX) or associated with meaning (Windows)

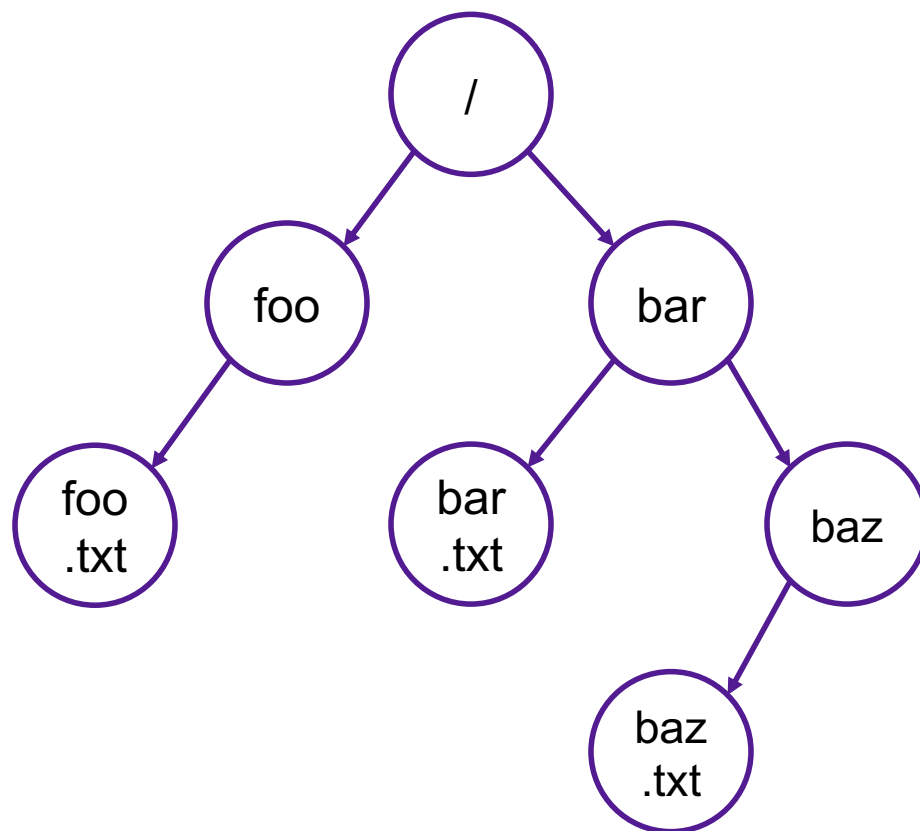
Directories

- a **directory** is a file that provides mappings from human-readable names to low-level names (i.e., file numbers):
 - a list of human-readable names
 - a mapping from each name to a specific underlying file or directory



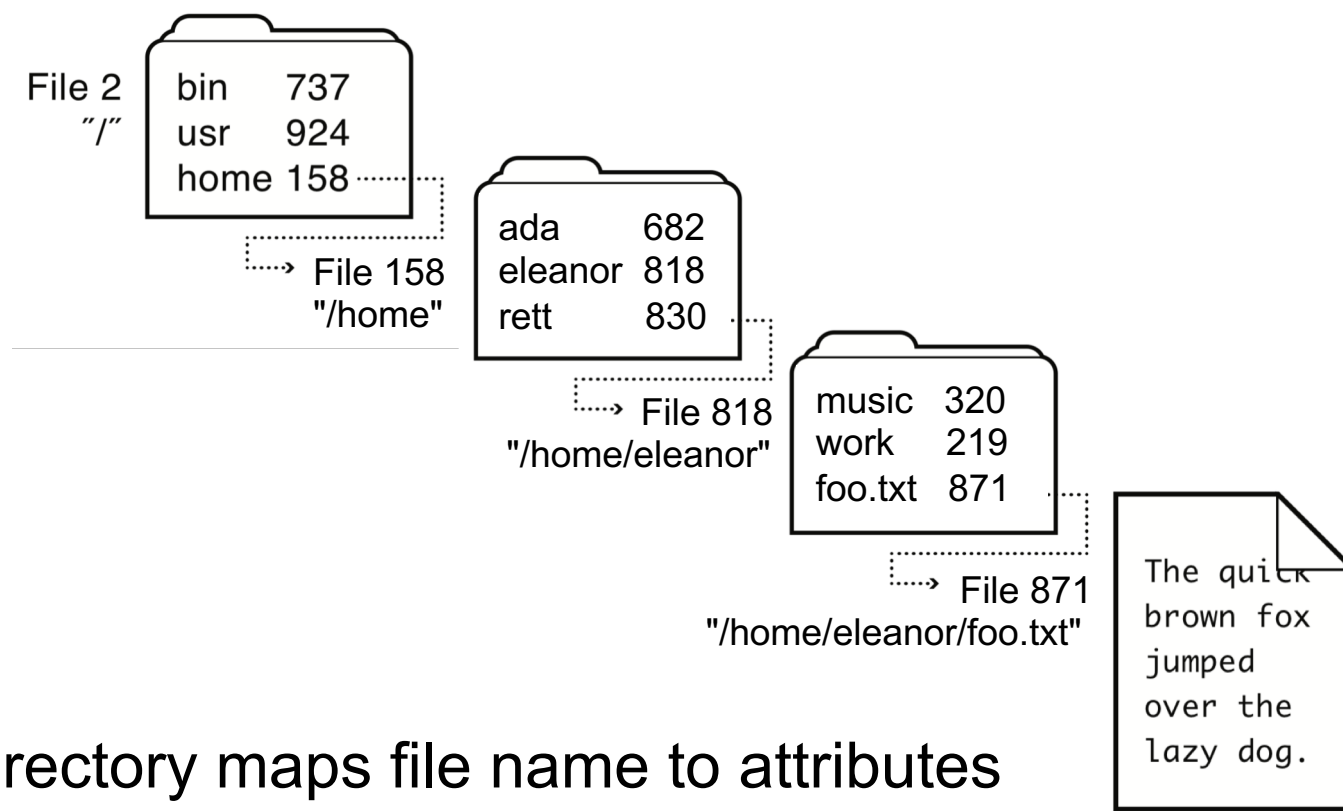
Path Names

- Each path from root is a name for a leaf
 - /foo/foo.txt
 - /bar/baz/baz.txt
- Each UNIX directory contains 2 special entries
 - "." = this directory
 - ".." = parent directory
- **Absolute paths:** path of file from the root directory
- **Relative paths:** path from current working directory



Directories

- OS uses path name to find directories and files



- Directory maps file name to attributes and locations

Basic File System Operations

- Create a file
- Write to a file
- Read from a file
- Seek to somewhere in a file
- Delete a file

How should we implement this?

File System Challenges

- **Performance:** despite limitations of disks
- **Flexibility:** need to support diverse file types and workloads
- **Persistence:** store data long term
- **Reliability:** resilient to OS crashes and hardware failures

File System Properties

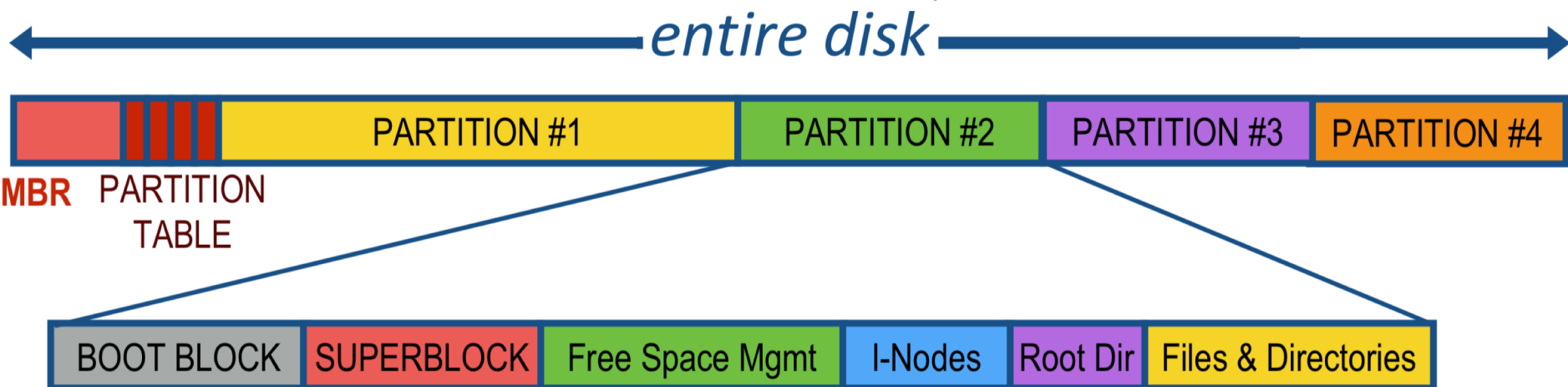
- Most files are small
 - need strong support for small files (optimize the common case)
 - block size can't be too big
- Directories are typically small
 - usually 20 or fewer entries
- Some files are very large
 - must handle large files
 - large file access should be reasonably efficient
- File systems are usually about half full

Implementation Basics

- Directories: file name -> low-level names (i.e., file numbers)
- Index structures: file number -> block
- Free space maps: find a free block (ideally nearby)
- Locality heuristics:
 - group directories
 - make writes sequential
 - defragment

File System Layout

- File systems are stored on disks
 - disks can be divided into one or more partitions
- Sector 0 of disk called Master Boot Record
 - executable boot loader
 - end of MBR: partition table (contains partitions' start & end addr.)
- Remainder of disk divided into partitions
 - First block of each partition is boot block (loaded by MBR on boot)
 - The rest of the partition stores the file system



Storing Files

Possible ways to allocate files:

- **Continuous allocation:** all bytes together, in order
- **Linked structure:** each block points to the next block
- **Indexed structure:** index block points to many other blocks
- **Log structure:** sequence of segments, each containing updates

Which is the best?

For sequential access?

For random access?

For small files?

For large files?

Continuous 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



Storing Files

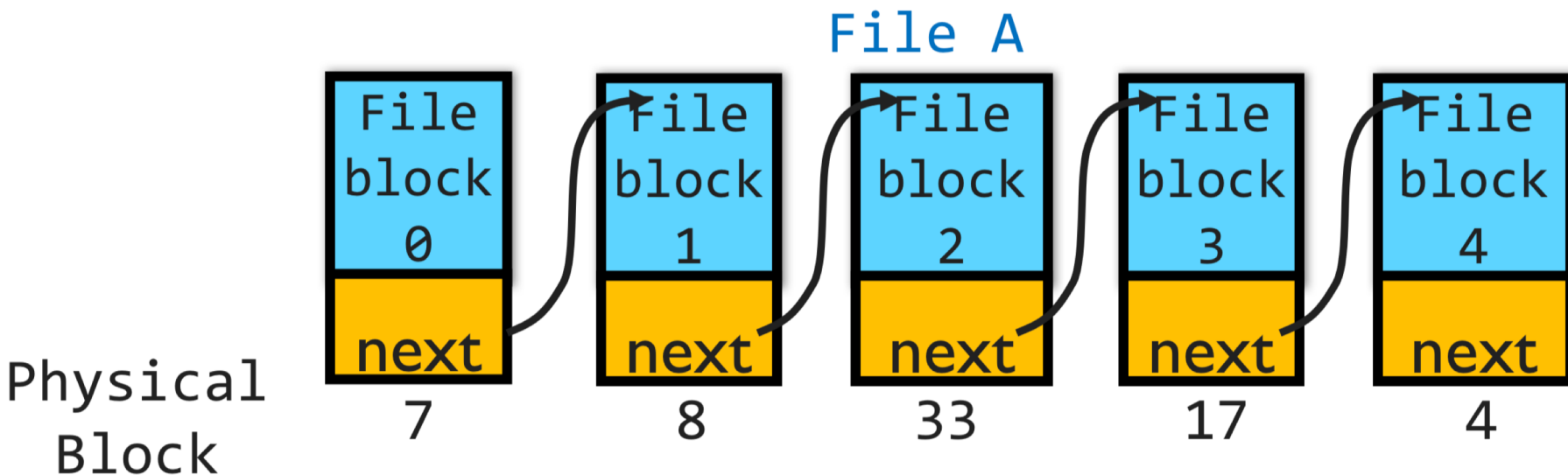
Possible ways to allocate files:

- **Continuous allocation:** all bytes together, in order
- **Linked structure:** each block points to the next block
- **Indexed structure:** index block points to many other blocks
- **Log structure:** sequence of segments, each containing updates

Linked Allocation

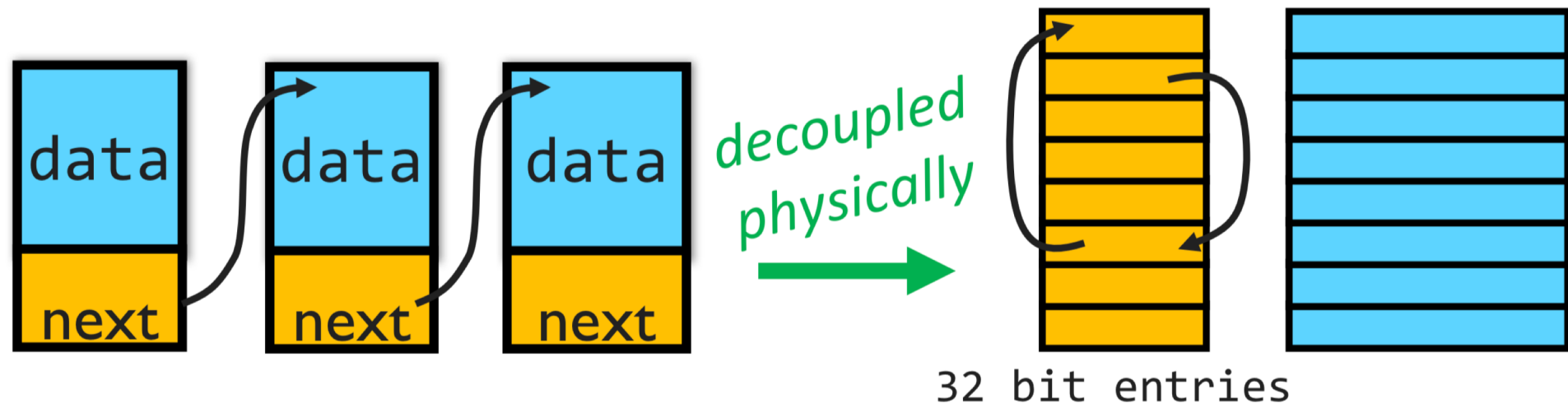
Each file is stored as linked list of blocks: First word of each block points to next block, rest of disk block is file data

- + **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) File System

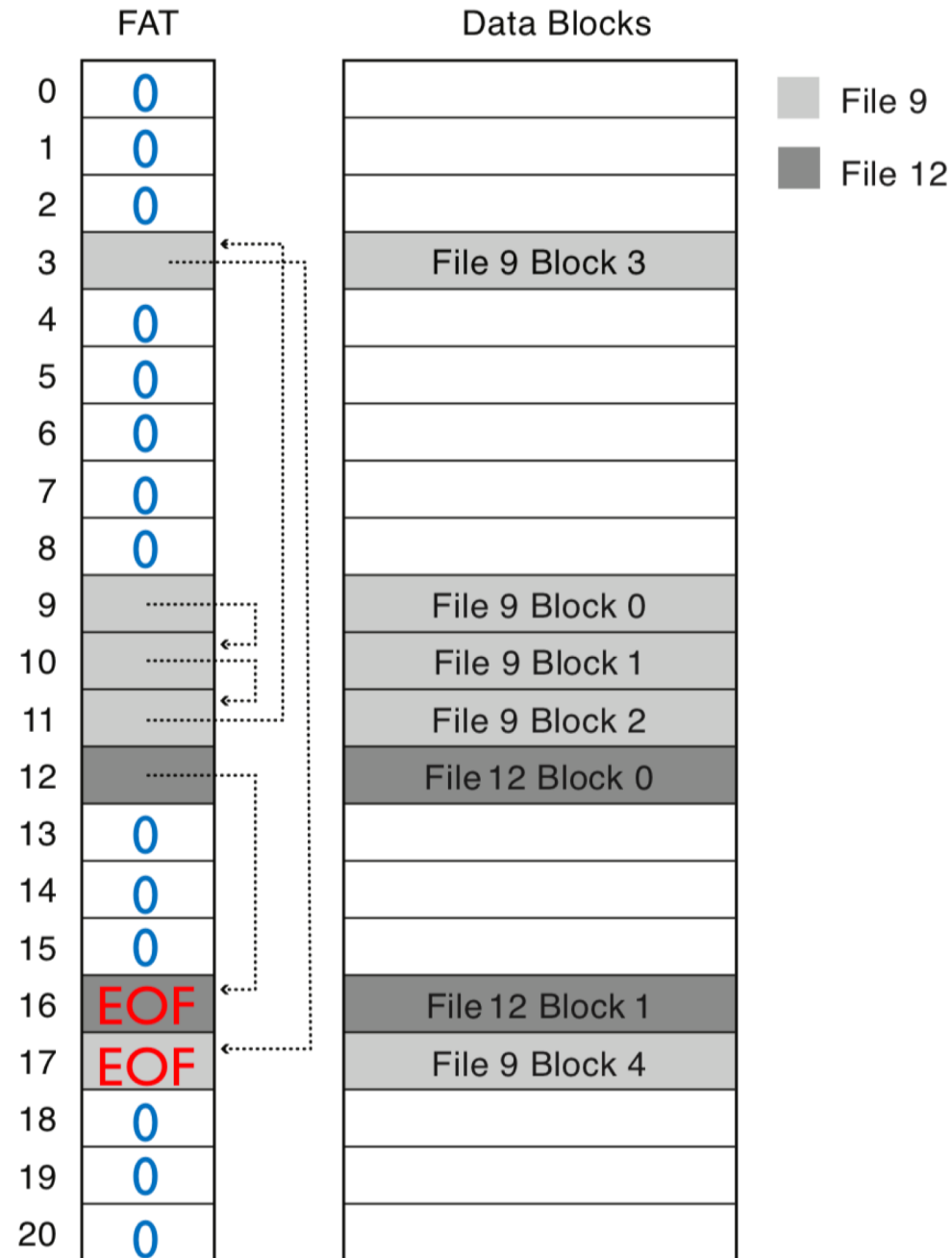
- Developed by Microsoft for MS-DOS
- Still widely used for flash drives, camera cards, etc.
- Fat-32 supports 2^{28} blocks and files of $2^{32} - 1$ bytes
- File table:
 - Linear map of all blocks on disk
 - Each file a linked list of blocks



FAT File System

- 1 entry per block
- EOF for last block
- 0 indicates free block
- low-level name = FAT index of first block in file

Directory	
bart.txt	9
maggie.txt	12

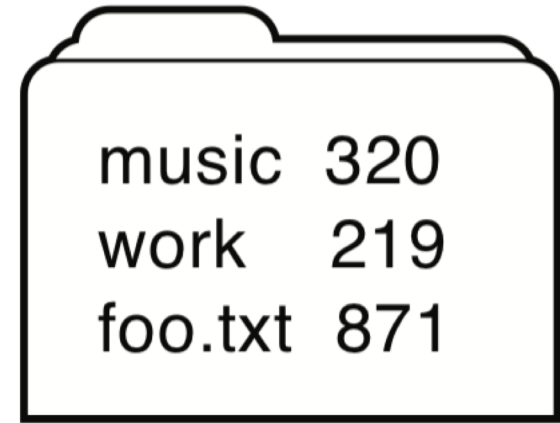


FAT Directory Structure

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



music	320
work	219
foo.txt	871

Exercise

- How many disk reads would be required to read (all of) a 2^{15} byte file named `/foo/bar/baz.txt`
 - assume 4096 byte blocks
 - assume that all directories are small enough to fit in one block

Multiple human-readable names

- Many file systems allow a given file to have multiple names
- Hard links are multiple file directory entries that map different path names to the same file number
- Symbolic Links or soft links are directory entries that map one name to another (effectively a redirect)
- Exercise: how could we implement symbolic links in the FAT file system?

Evaluating Fat

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)
- Poor random access
- Limited metadata
- Limited access control
- Limitations on volume and file size
- No support for reliability techniques

Storing Files

Possible ways to allocate files:

- **Continuous allocation:** all bytes together, in order
- **Linked structure:** each block points to the next block
- **Indexed structure:** index block points to many other blocks
- **Log structure:** sequence of segments, each containing updates