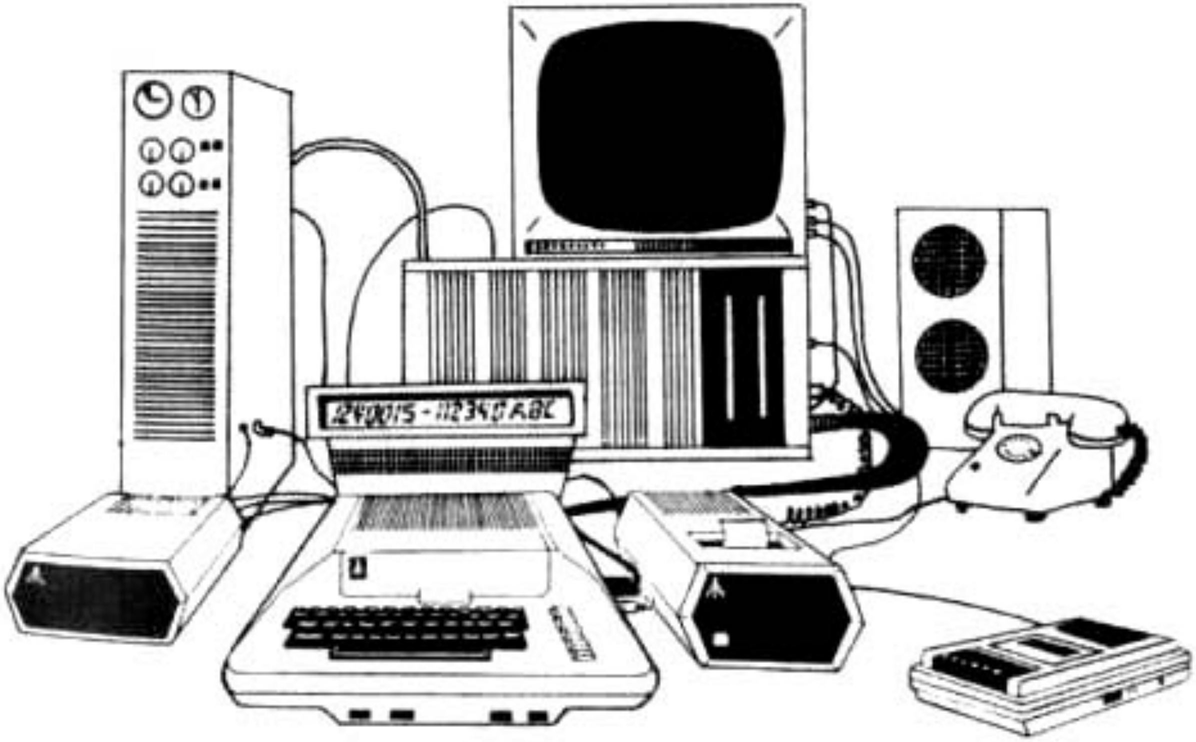
Lecture 26: File Systems

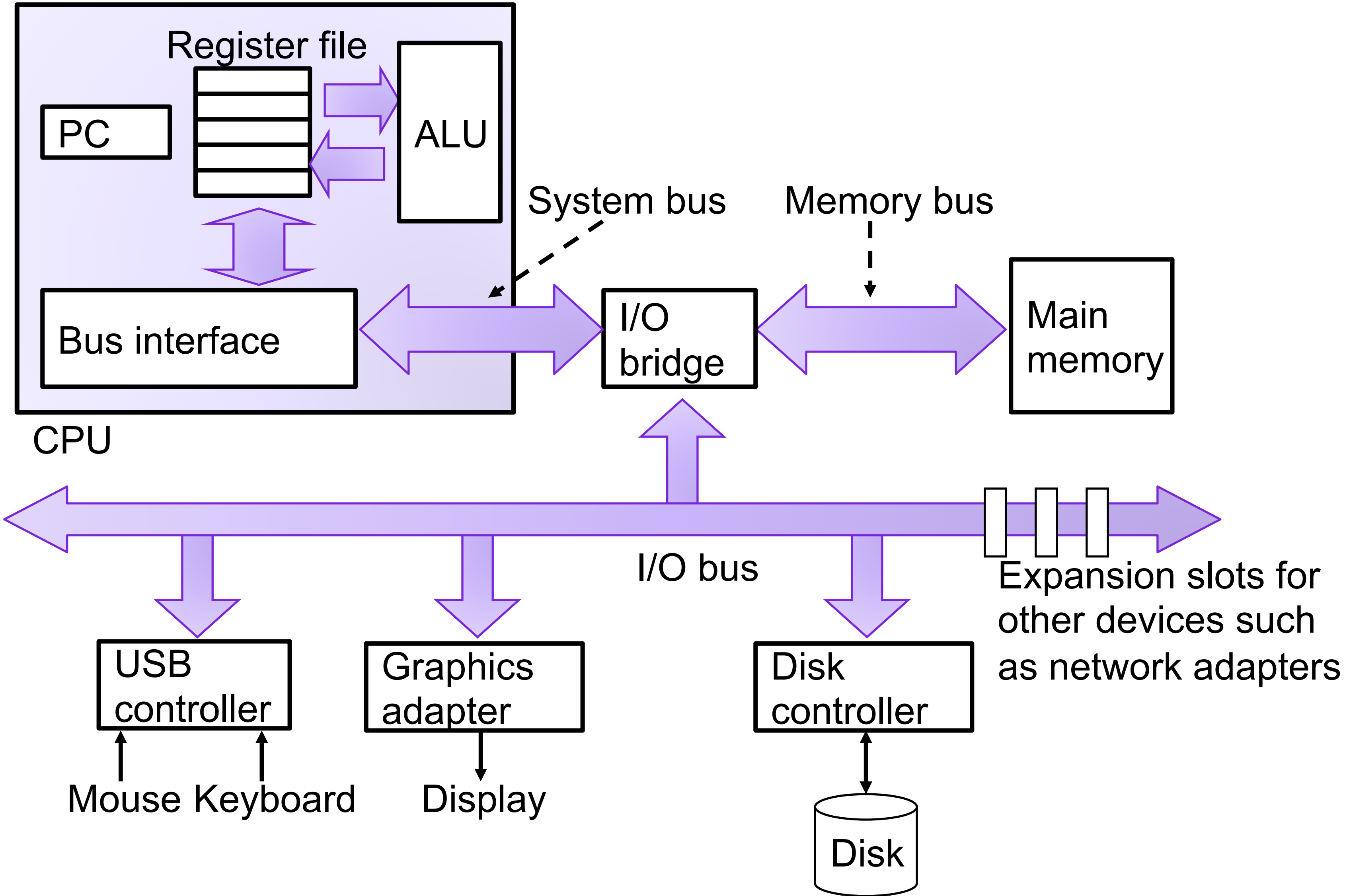
CS 105

April 29, 2019

Input and Output



Input and Output



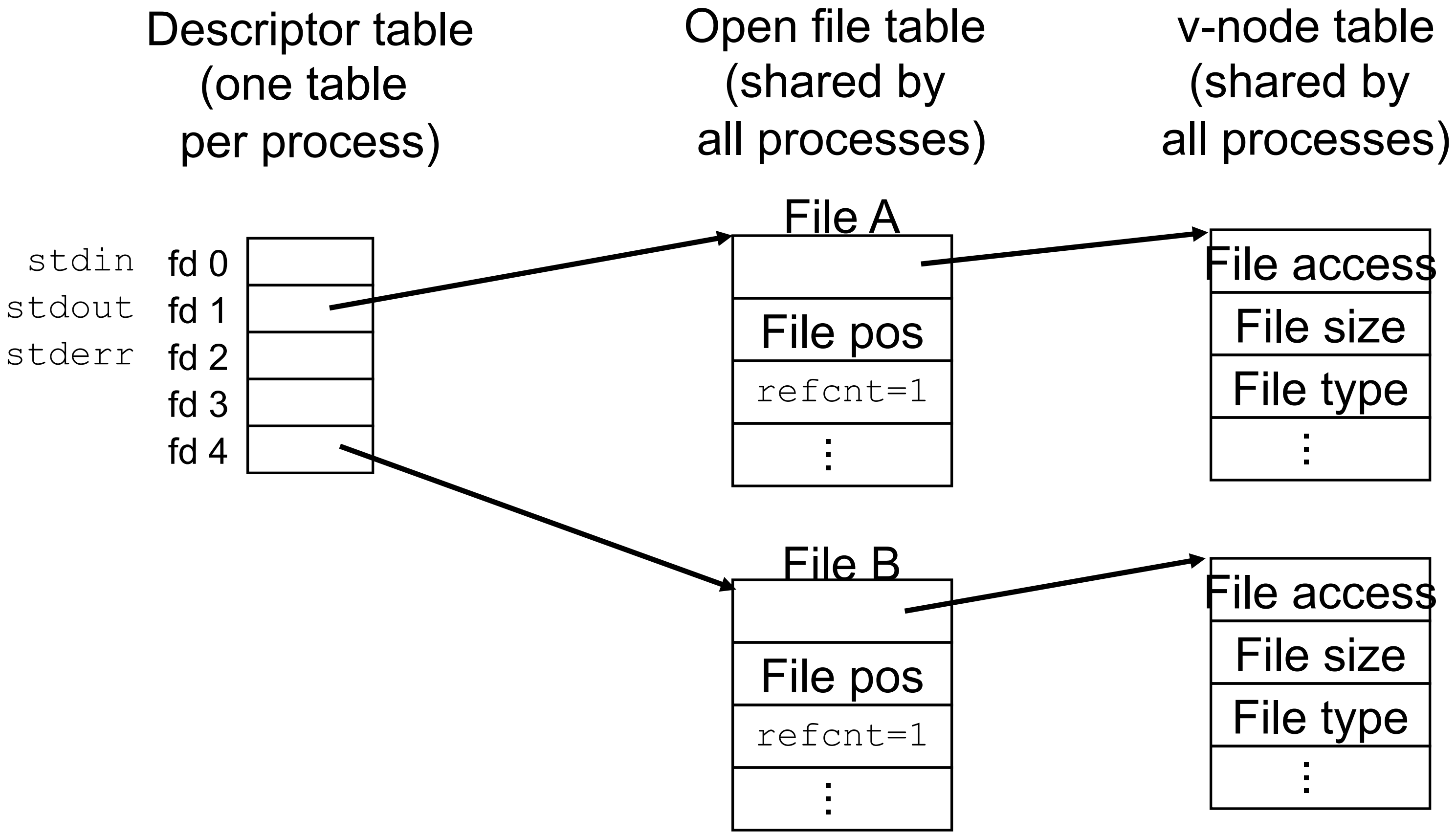
Unix I/O Overview

- All I/O devices are represented as files:
  - **`/dev/sda2`** (**`/usr`** disk partition)
  - **`/dev/tty2`** (terminal)

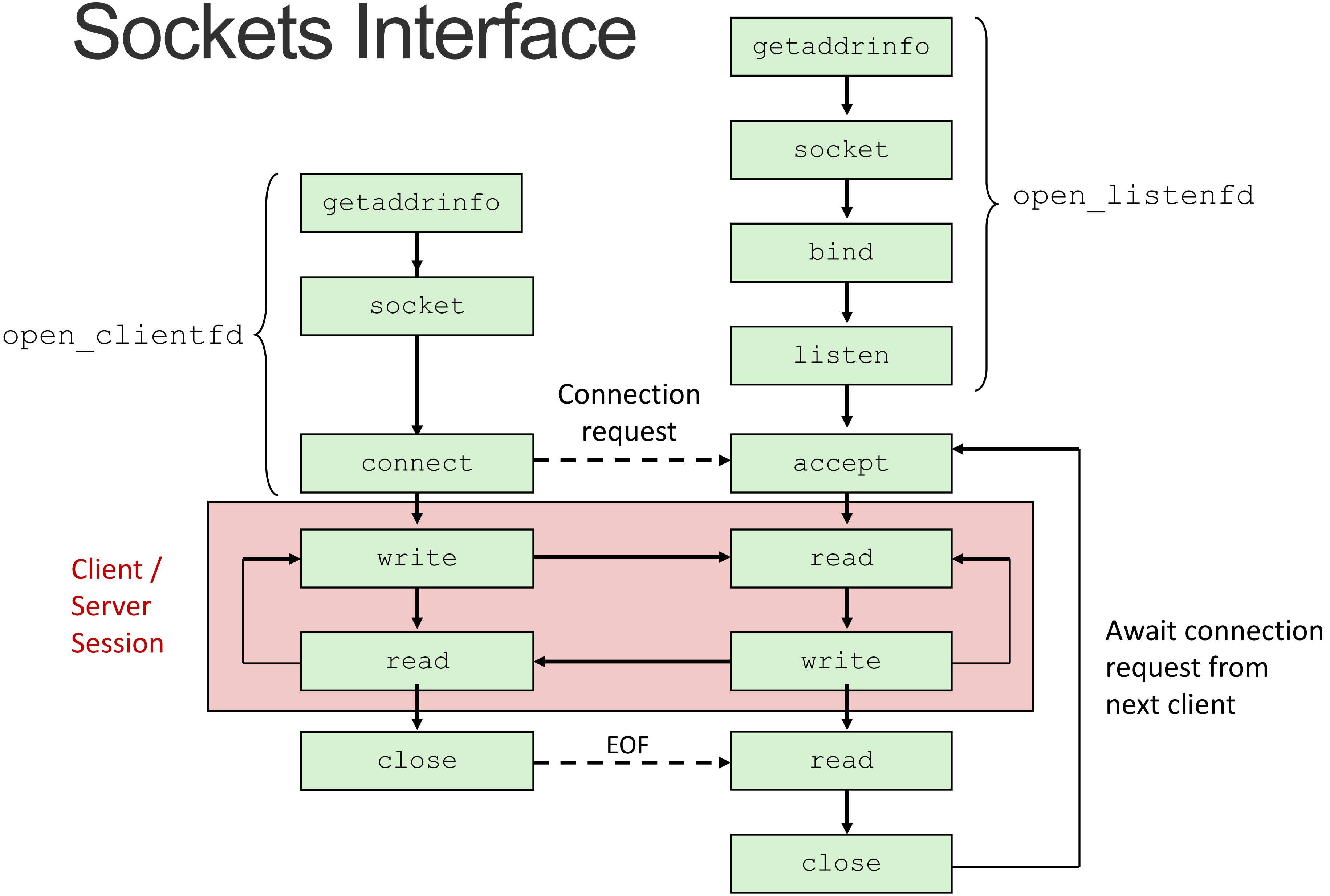
- A Linux *file* is a sequence of *m* bytes:
  - $B_0, B_1, \ldots\ldots, B_k, \ldots\ldots, B_{m-1}$

- Each process created by a Linux shell begins life with three open files associated with a terminal:
  - 0: standard input (stdin)
  - 1: standard output (stdout)
  - 2: standard error (stderr)
- Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O:*
  - *open, close, read, write, seek*

Kernel Data Structures

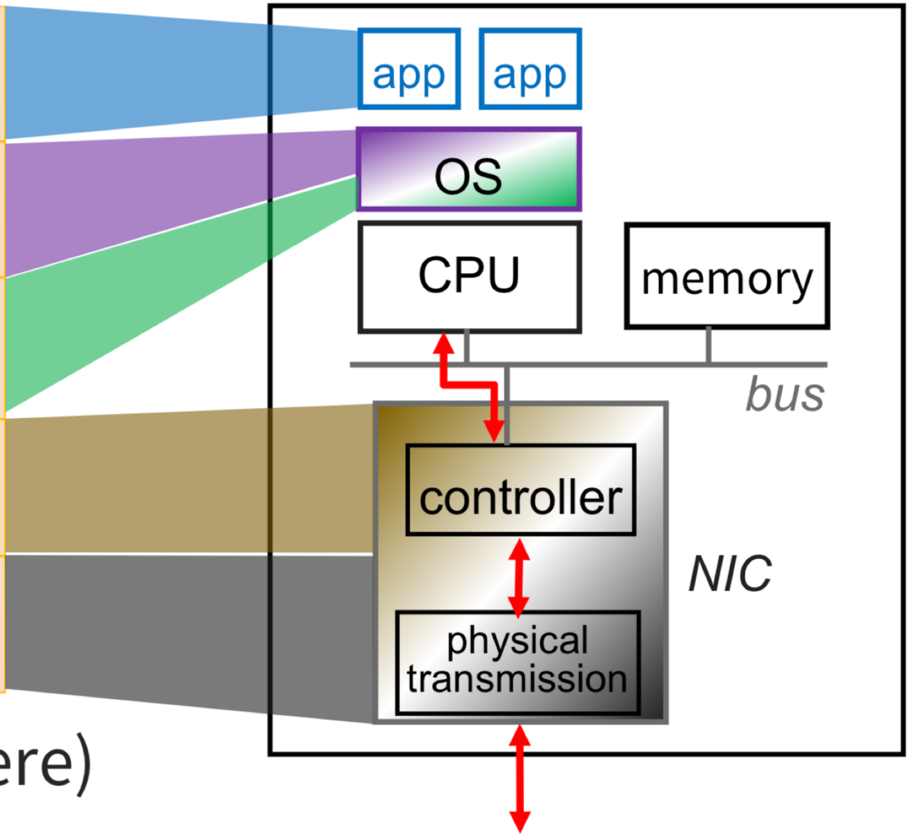


Sockets Interface



Hardware and Software Interfaces

Application	HTTP, FTP, DNS (*these*^ are usually in libraries)
Transport	TCP, UDP
Network	IP, ICMP (ping)
Link	Ethernet, WiFi
Physical	wires, signal encoding



(Hard to draw firm lines here)

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 μs
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
  - presents processes with persistent, named data
  - two main components: files and directories

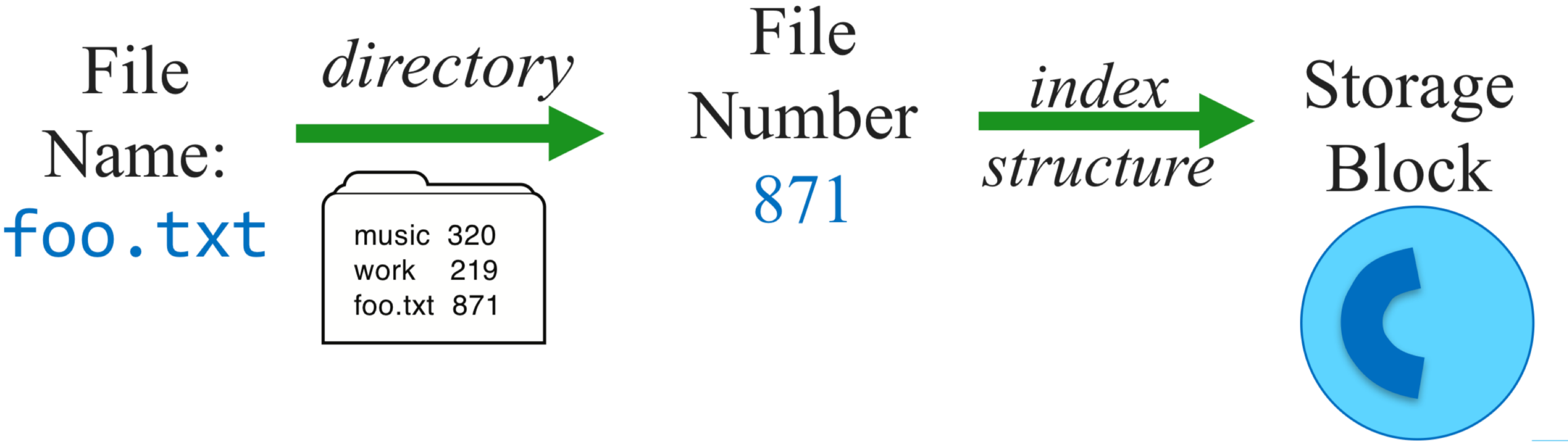
The File Abstraction

- a **file** is a named collection of data
  - name is defined on creation
  - processes use name to subsequently access that file
  - processes don't care where on disk a file is stored

- a file is comprised of two parts:
  - **data**: information a user or application puts in a file, stored as an array of untyped bytes
  - **metadata**: information added and managed by the OS (e.g., size, owner, security info, modification time)

Directories

- a **directory** provides names for files:
  - a list of human-readable names
  - a mapping from each name to a specific underlying file or directory



File Name:
foo.txt

directory

music	320
work	219
foo.txt	871

File Number
871

index structure

Storage Block

Path Names

- Absolute: path of file from the root directory

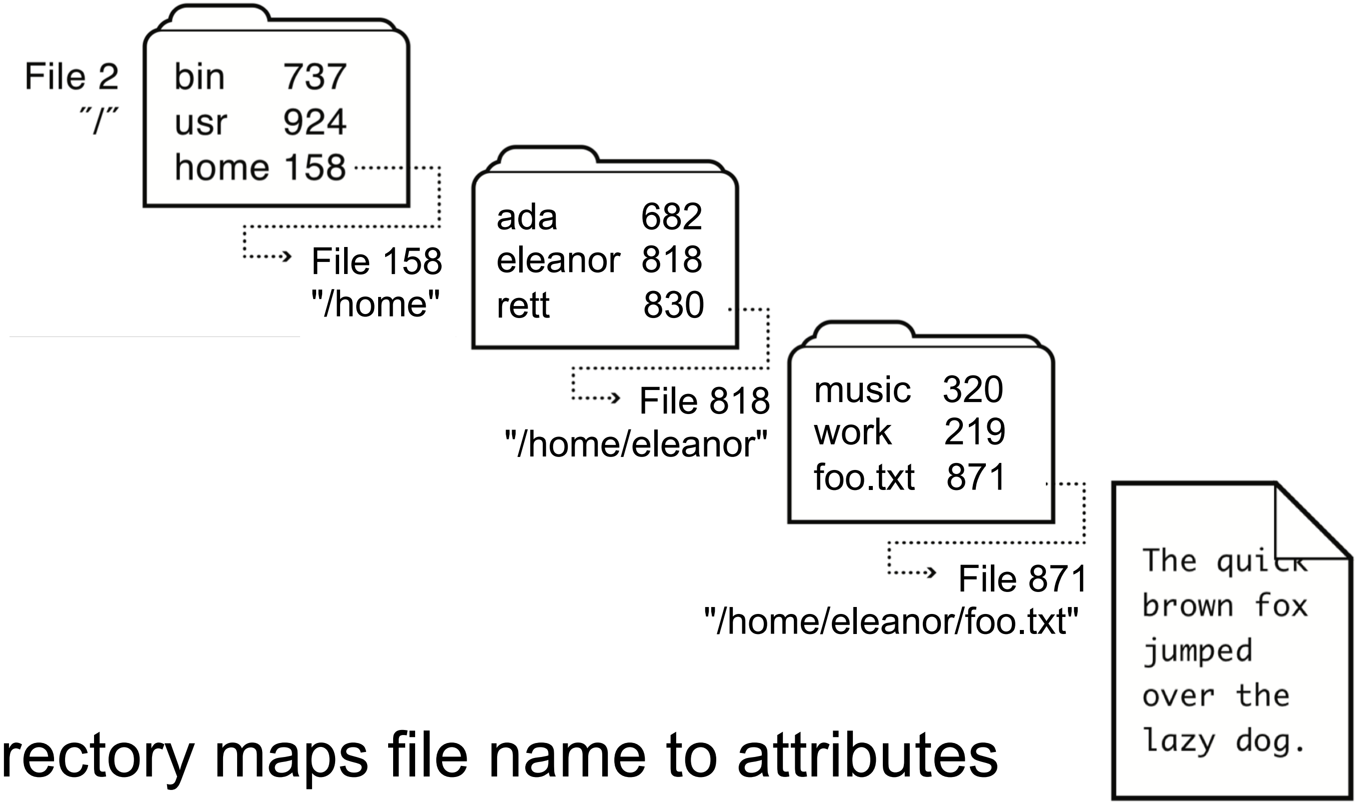
  /home/ada/projects/babbage.txt
- Relative: path of file from the current working directory

  projects/babbage.txt

- Two special entries in each Unix directory:
  - . = current directory
  - .. = parent directory

Directories

- OS uses path name to find directories and files



File 2
"/"

bin 737
usr 924
home 158

File 158
"/home"

ada 682
eleanor 818
rett 830

File 818
"/home/eleanor"

music 320
work 219
foo.txt 871

File 871
"/home/eleanor/foo.txt"

The quick
brown fox
jumped
over the
lazy dog.

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

- **Reliability:** must store data for long periods of time despite OS crashes or hardware malfunctions

Implementation Basics

- Directories: file name -> file number
- Index structures: file number -> block
- Free space maps: find a free block (ideally nearby)
- Locality heuristics:
  - group directories
  - make writes sequential
  - defragment

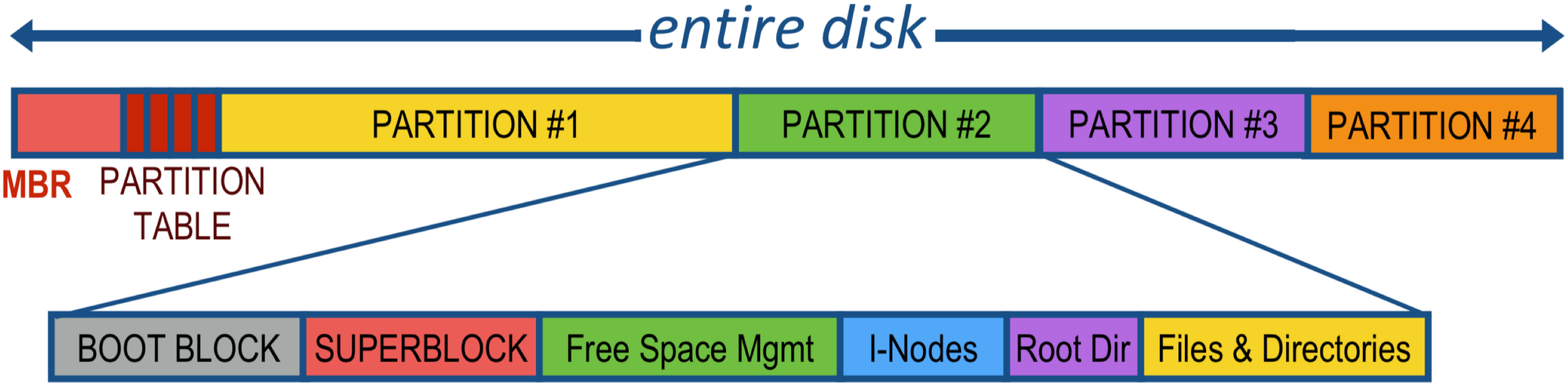
File System Properties

- Most files are small
  - need strong support for small files (optimize the common case)
  - block size can't be too big

- Some files are very large
  - must handle large files
  - large file access should be reasonably efficient

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
  - end of MBR: partition table (contains partitions' start & end addr.)
- First block of each partition has boot block
  - loaded by MBR and executed on boot



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

Which is the best?

For sequential access?
For random access?
For small files?
For large files?

Continuous Allocation

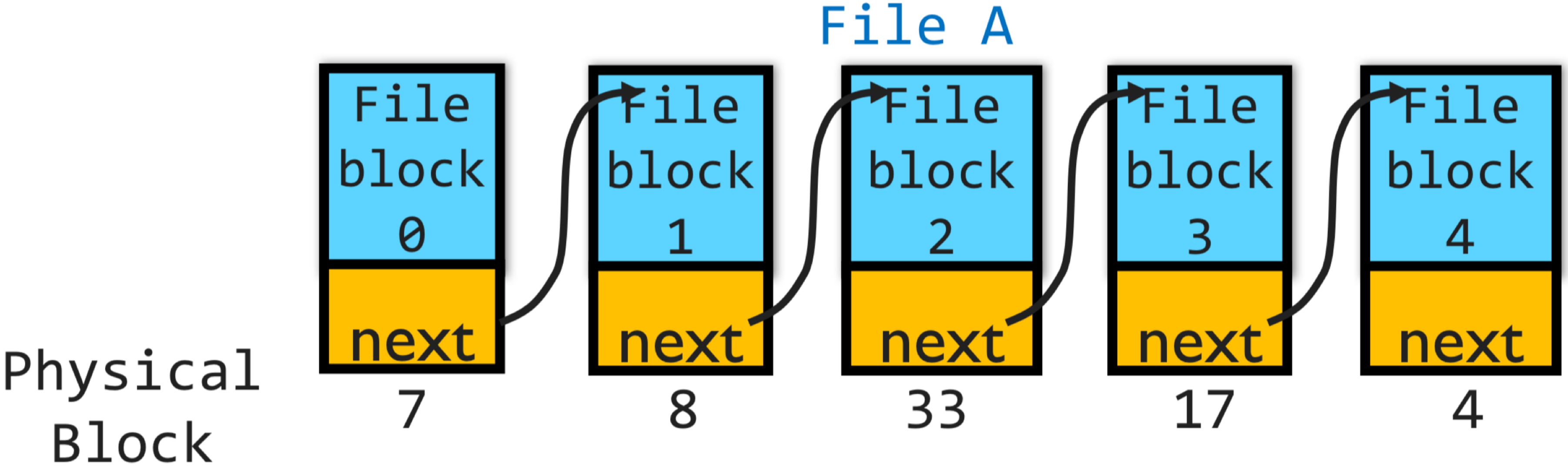
All bytes together, in order

- **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

Linked Allocation

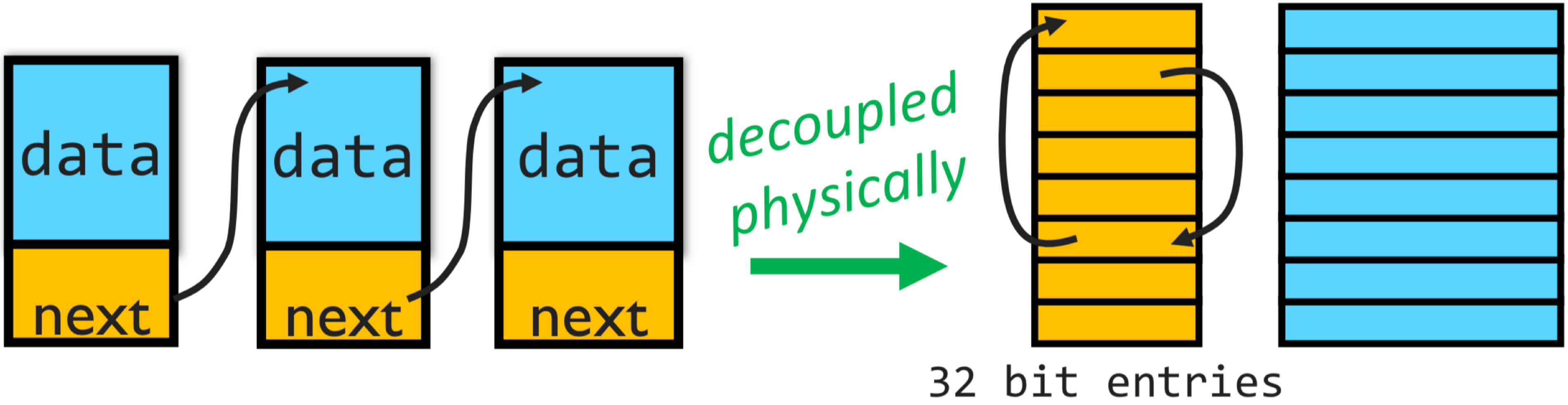
Each file is stored as linked list of blocks: First word of each block points to next block, rest of disk block is file data

- **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

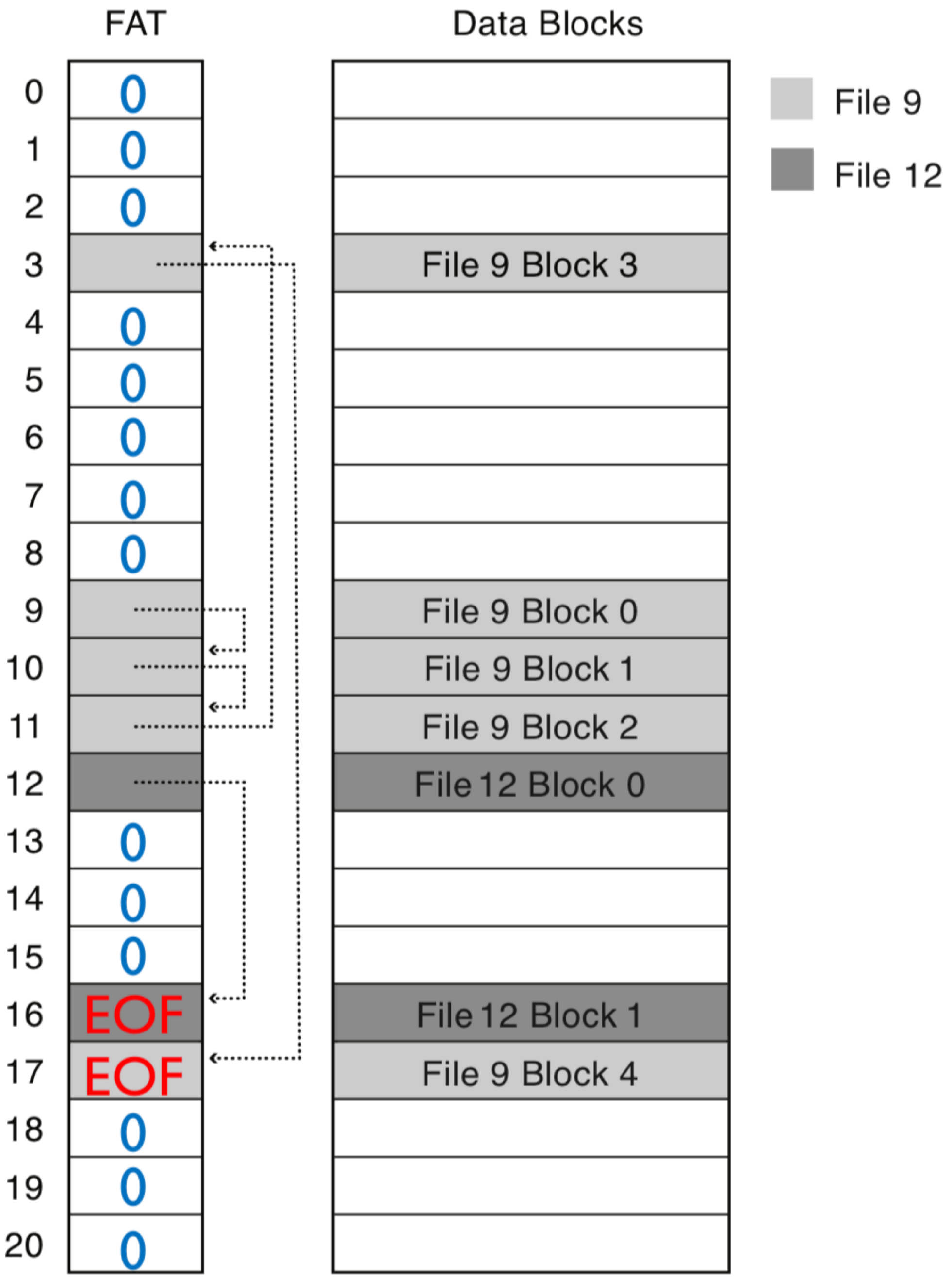
- Developed by Microsoft for MS-DOS
- Still widely used for flash drives, camera cards, etc.
- Fat-32 supports 2^{28} blocks and files of $2^{32} - 1$ bytes
- File table:
  - Linear map of all blocks on disk
  - Each file a linked list of blocks



FAT File System

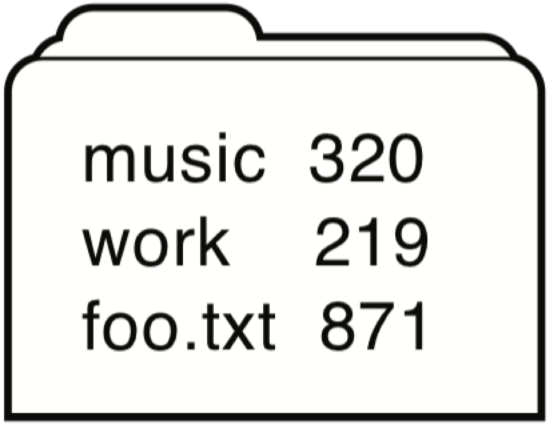
- 1 entry per block
- EOF for last block
- 0 indicates free block
- directory entry maps name to FAT index

Directory	
bart.txt	9
maggie.txt	12



	FAT	Data Blocks
0	0	
1	0	
2	0	
3		File 9 Block 3
4	0	
5	0	
6	0	
7	0	
8	0	
9		File 9 Block 0
10		File 9 Block 1
11		File 9 Block 2
12		File 12 Block 0
13	0	
14	0	
15	0	
16	EOF	File 12 Block 1
17	EOF	File 9 Block 4
18	0	
19	0	
20	0	

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

Evaluating Fat

How is FAT good?

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

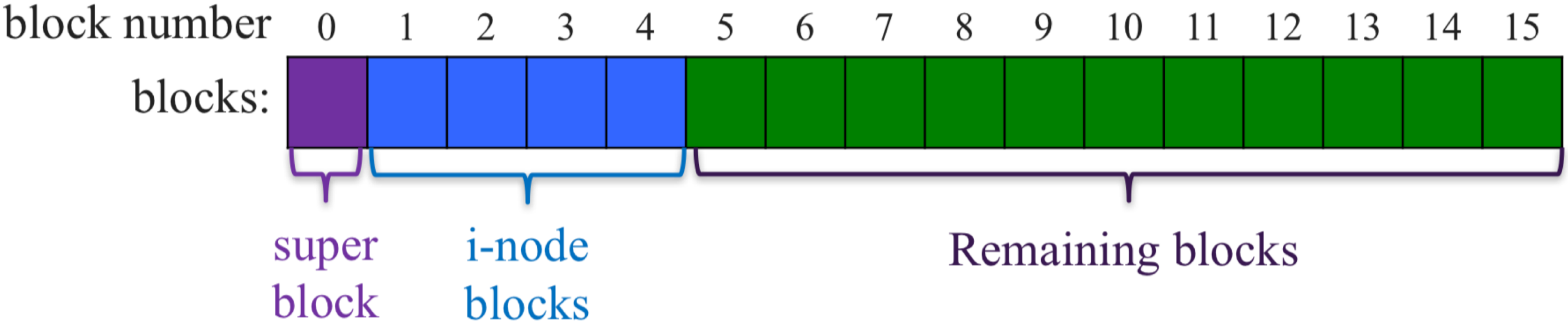
How is FAT bad?

- 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

Indexed Allocation: Fast File System (FFS)

- tree-based, multi-level index

- **superblock** identifies file system's key parameters
- **inodes** store metadata and pointers
- **datablocks** store data



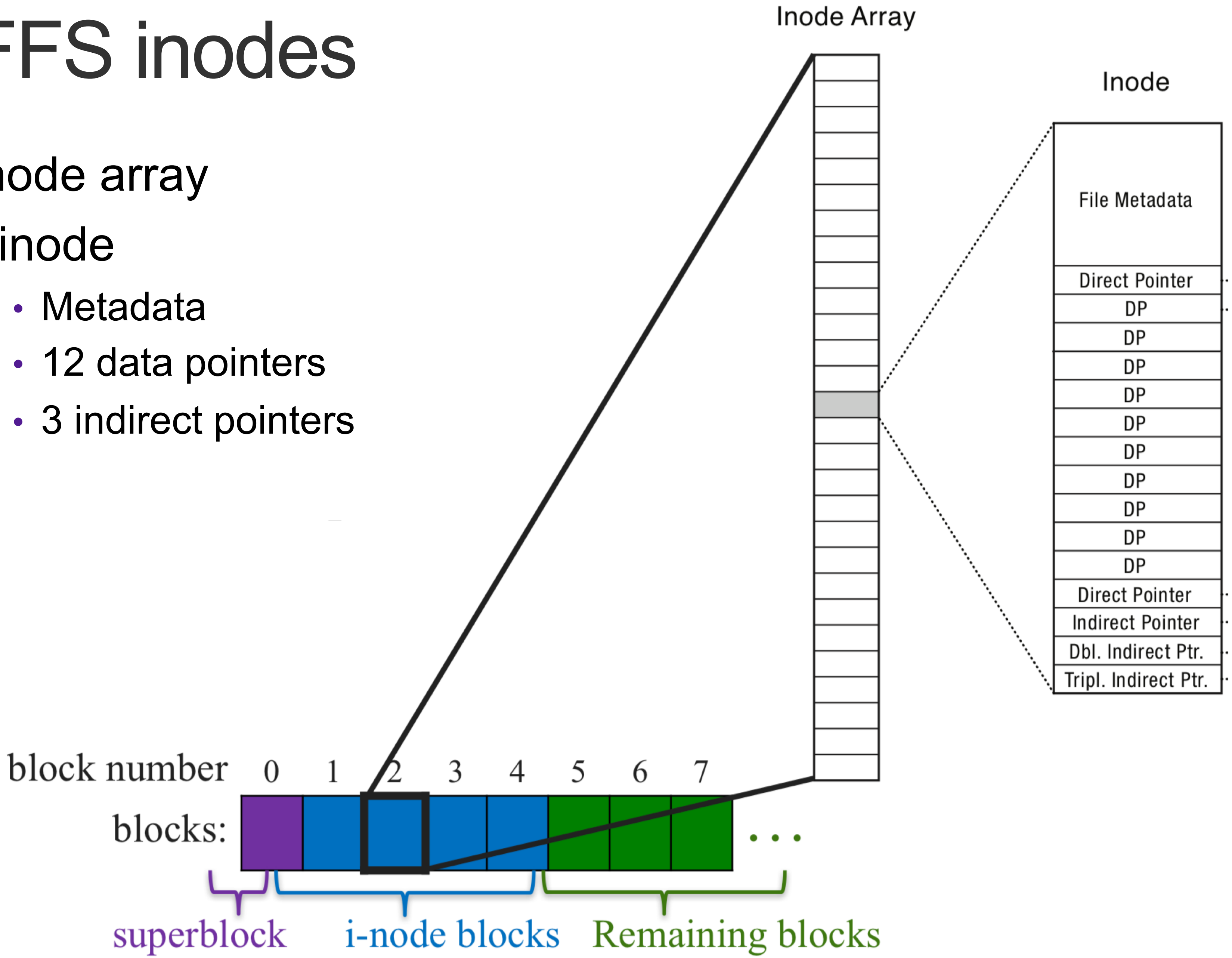
FFS Superblock

- Identifies file system’s key parameters:
  - type
  - block size
  - inode array location and size
  - location of free list

FFS inodes

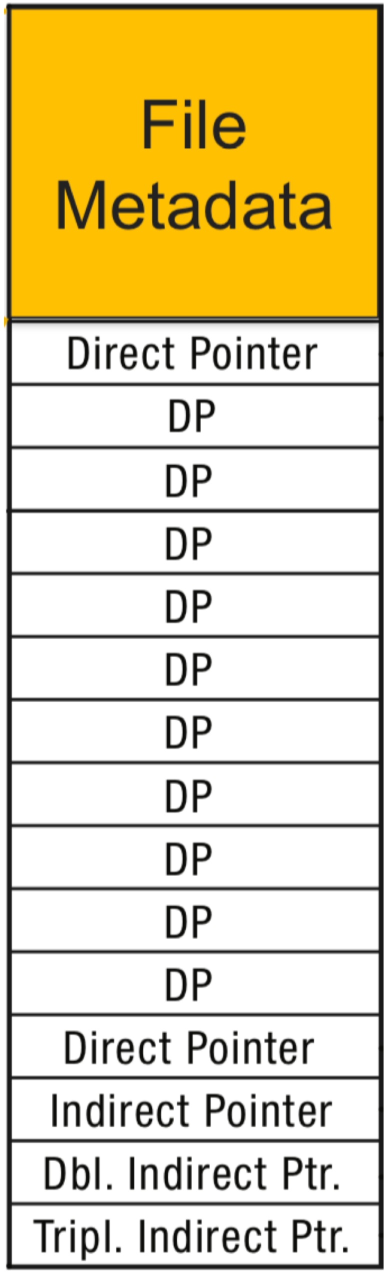
inode array

- inode
  - Metadata
  - 12 data pointers
  - 3 indirect pointers



inode Metadata

- Type
  - ordinary file
  - directory
  - symbolic link
  - special device
- Size of the file (in #bytes)
- # links to the i-node
- Owner (user id and group id)
- Protection bits
- Times: creation, last accessed, last modified



File Metadata

Direct Pointer

DP

DP

DP

DP

DP

DP

DP

DP

DP

DP

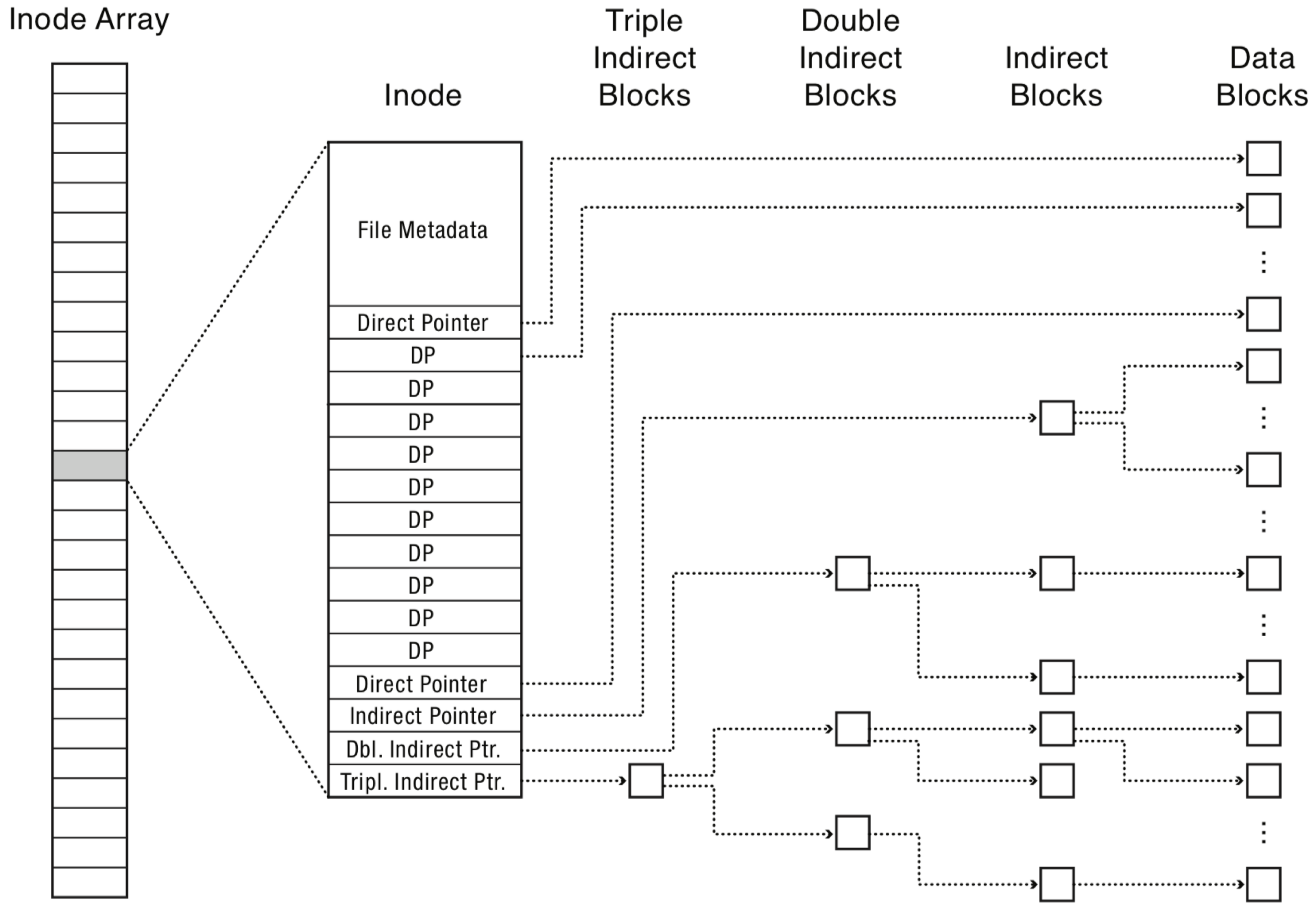
Direct Pointer

Indirect Pointer

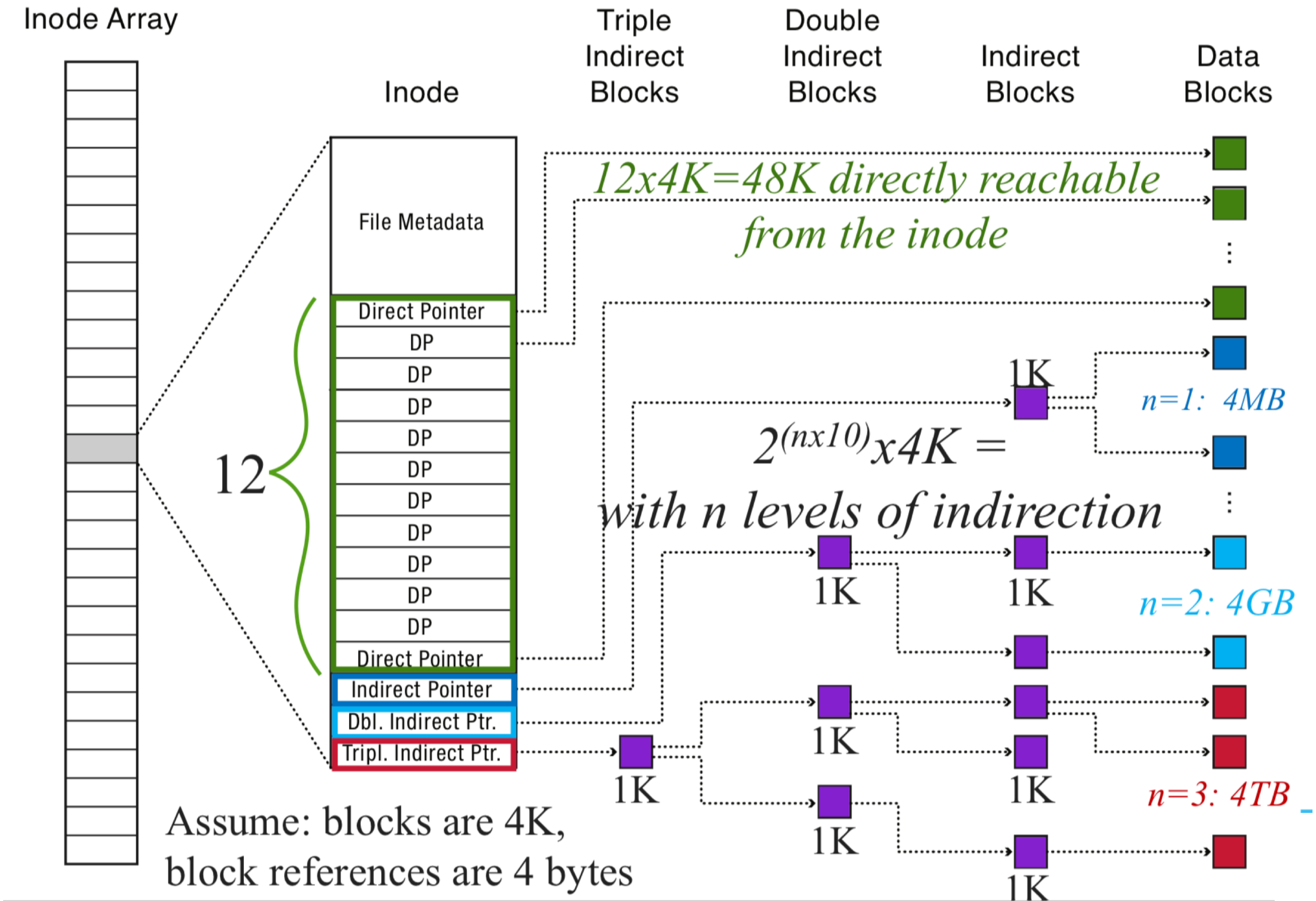
Dbl. Indirect Ptr.

Tripl. Indirect Ptr.

FFS Index Structures



FFS Index Structures



12x4K=48K directly reachable from the inode

$2^{(n\times10)}\times 4K =$ with n levels of indirection

n=1: 4MB

n=2: 4GB

n=3: 4TB

Assume: blocks are 4K, block references are 4 bytes

Key Characteristics of FFS

- Tree Structure
  - efficiently find any block of a file
- High Degree (or fan out)
  - minimizes number of seeks
  - supports sequential reads & writes
- Fixed Structure
  - implementation simplicity
- Asymmetric
  - not all data blocks are at the same level
  - supports large files
  - small files don’t pay large overheads