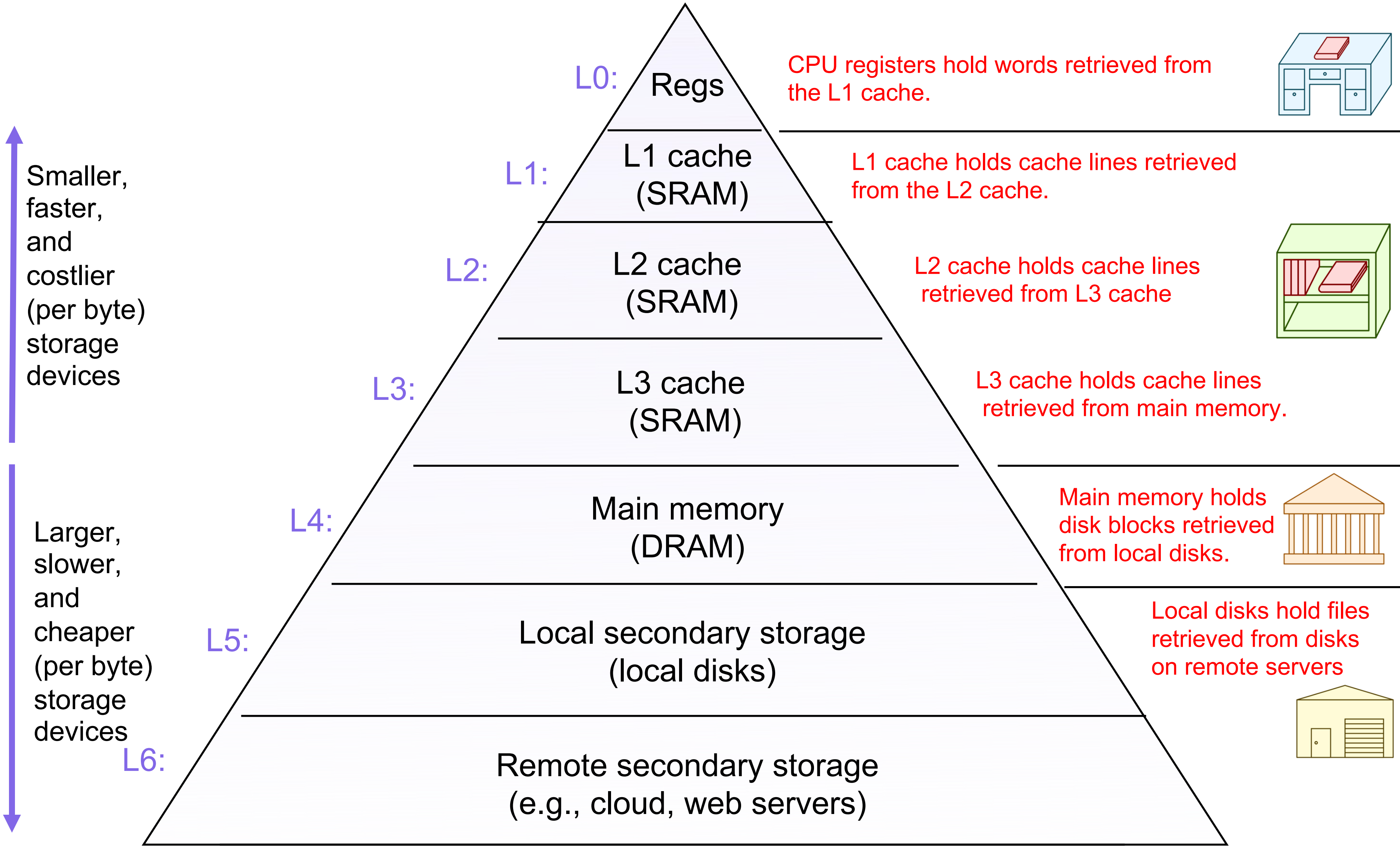
Lecture 22: File Systems

CS 105

Fall 2023

Memory Hierarchy



Smaller, faster, and costlier (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices

L0: Regs

CPU registers hold words retrieved from the L1 cache.

L1: L1 cache (SRAM)

L1 cache holds cache lines retrieved from the L2 cache.

L2: L2 cache (SRAM)

L2 cache holds cache lines retrieved from L3 cache

L3: L3 cache (SRAM)

L3 cache holds cache lines retrieved from main memory.

L4: Main memory (DRAM)

Main memory holds disk blocks retrieved from local disks.

L5: Local secondary storage (local disks)

Local disks hold files retrieved from disks on remote servers

L6: Remote secondary storage (e.g., cloud, web servers)

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

The File System Abstraction

- interface that provides operations on data stored long-term on disk

- a **file** is a named sequence of stored bytes
  - 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
  - **metadata**: information added and managed by the OS
    - e.g., size, owner, security info, modification time

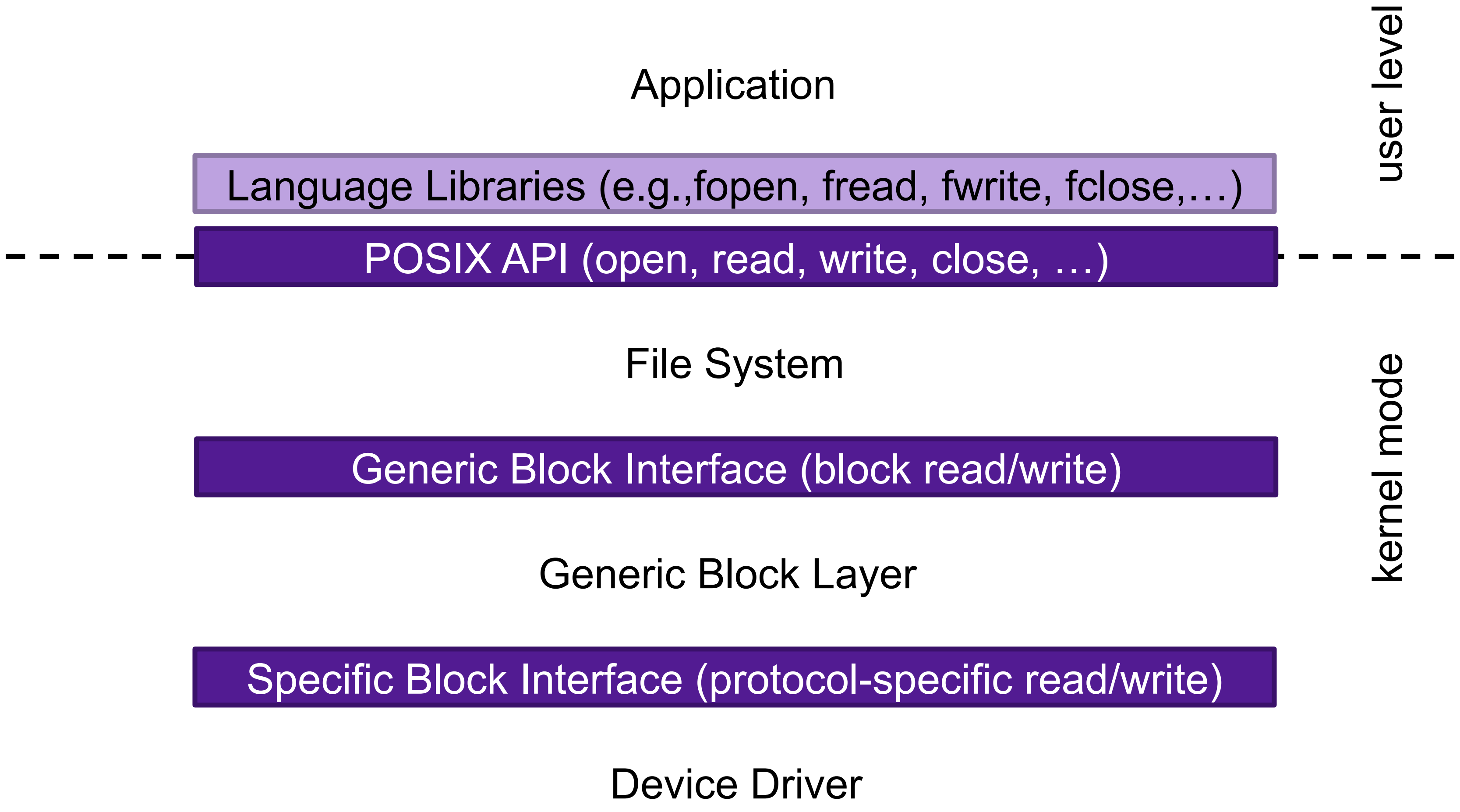
System I/O as a Uniform Interface

- Operating systems use the System I/O commands as an interface for all I/O devices

- Examples of files include
  - file
  - keyboard
  - screen
  - pipe
  - device
  - network

- The commands to read and write to an open file descriptor are the same no matter what kind of "file" it is

The File System Stack



Application

Language Libraries (e.g.,fopen, fread, fwrite, fclose,…)

POSIX API (open, read, write, close, …)

File System

Generic Block Interface (block read/write)

Generic Block Layer

Specific Block Interface (protocol-specific read/write)

Device Driver

user level

kernel mode

File Descriptors

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

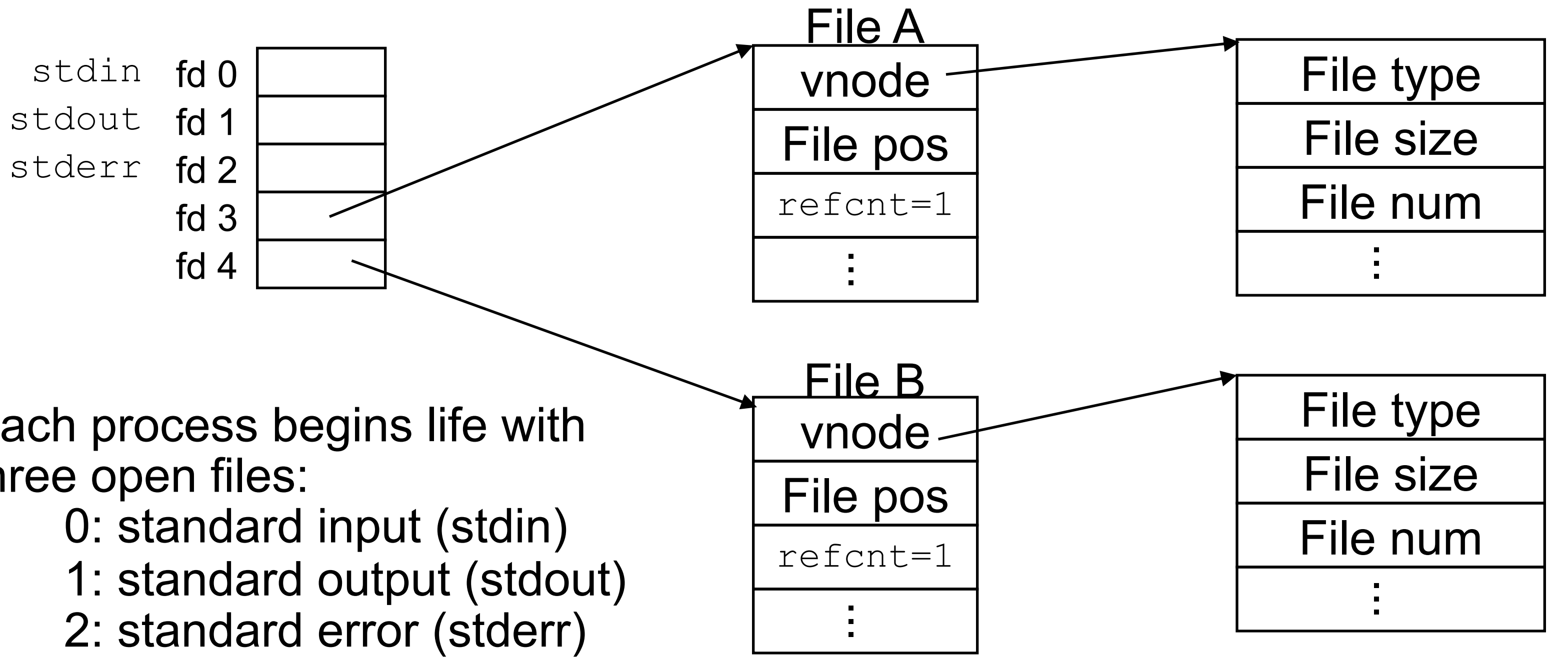
- Returns a small integer **file descriptor**
  - `fd == -1` indicates that an error occurred

Kernel Data Structures

Descriptor table
(table created on fork(),
one table
per process)

Open file table
(entry created on open(),
shared by
all processes)

vnode table
(one per open file,
shared by
all processes)



Each process begins life with
three open files:
- 0: standard input (stdin)
- 1: standard output (stdout)
- 2: standard error (stderr)

The File System Abstraction

- interface that provides operations on data stored long-term on disk

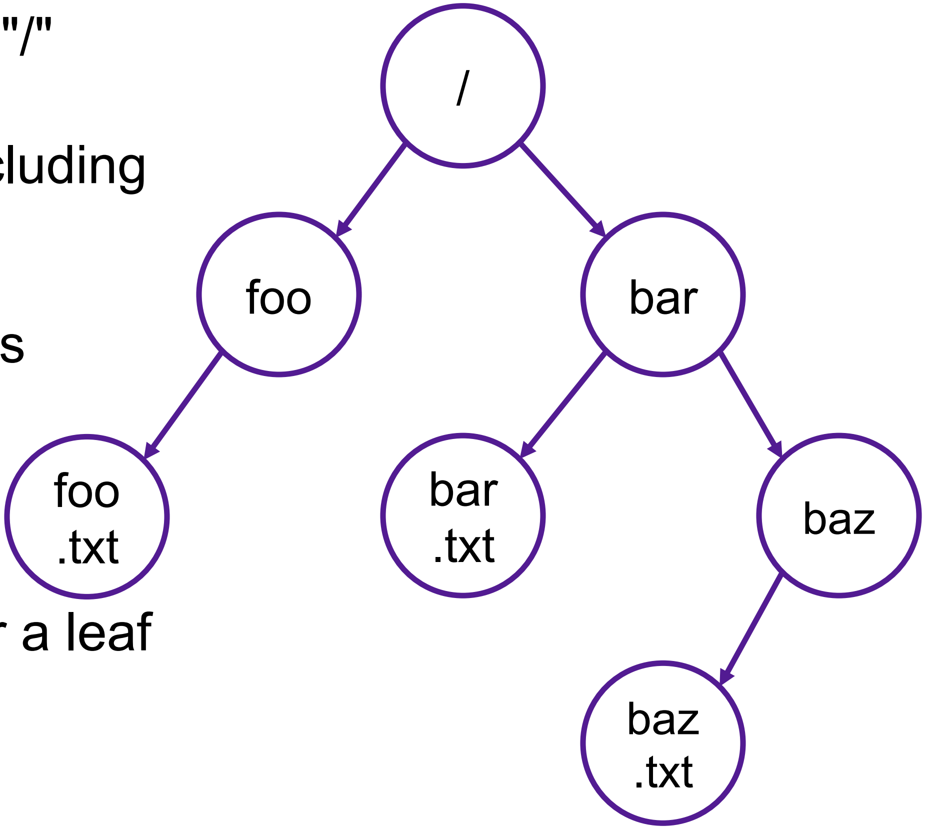
- a **file** is a named sequence of stored bytes
  - name is defined on creation
  - processes use name to subsequently access that file

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

- two types of files
  - **normal files**: data is an arbitrary sequence of bytes
  - **directories**: a special type of file that provides mappings from human-readable names to low-level names (i.e., file numbers)

Path Names

- A file system has a root directory "/"
- Directories contain other files (including subdirectories)
- Each UNIX directory also contains 2 special entries
  - "." = this directory
  - ".." = parent directory
- Each path from root is a name for a leaf
  - /foo/foo.txt
  - /bar/baz/baz.txt
- **Absolute paths:** path of file from the root directory
- **Relative paths:** path from current working directory



Exercise 1: Path Names

I’ve created a file named `example1.txt` in the directory `cs105`, which is located in the root directory.

1. Specify an absolute path to the file `example1.txt`
2. Specify a relative path to the file `example1.txt` from my home directory (`/home/ebac2018/`).

I've created a file named `example2.txt` in my home directory (`/home/ebac2018/`).

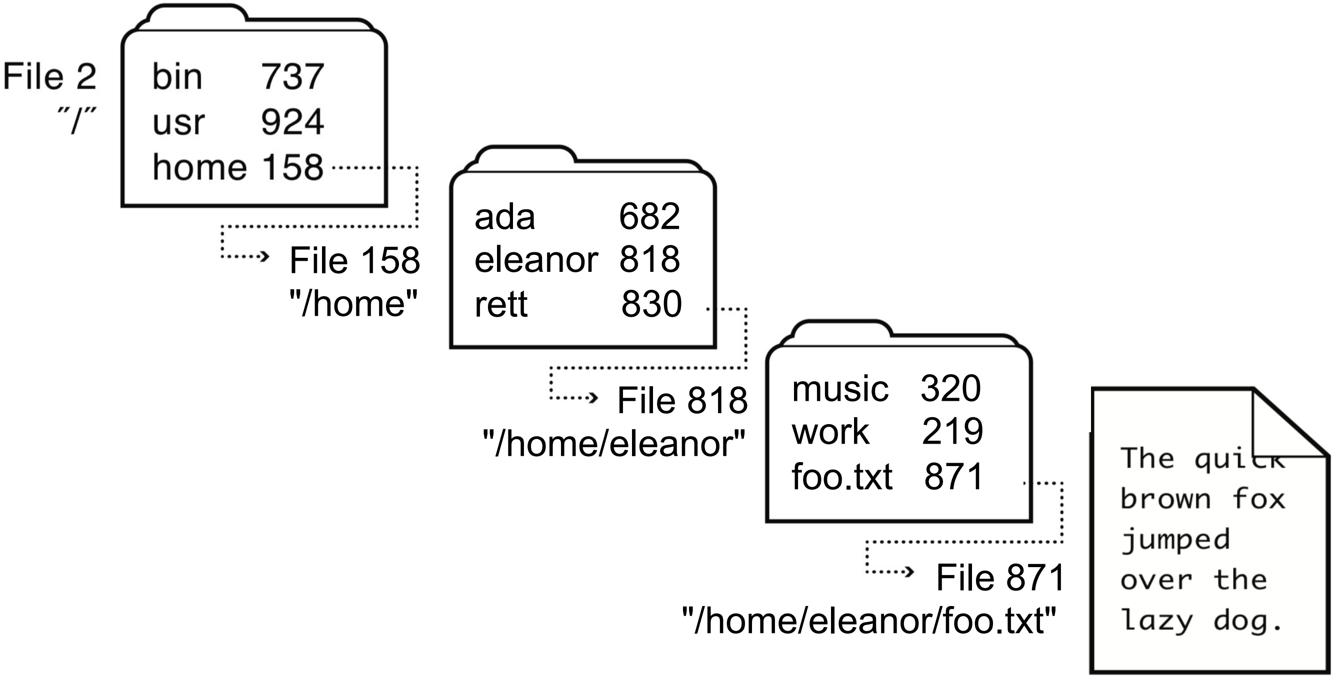
3. Specify an absolute path to the file `example2.txt`
4. Specify a relative path to the file `example2.txt` from your home directory

Hint: you can always get back to your home directory with `cd ~`

Hint: the name of your home directory is your username

Directories

- a **directory** is a file that provides mappings from human-readable names to low-level names (i.e., file numbers):
  - contents of a file are any array of directory entries
  - each directory entry contains a human-readable name and the corresponding file number
- 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.

Implementation Basics

- Directories: file name -> low-level names (i.e., file numbers)
- File system index structures: file number -> block(s)

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

Storing Files

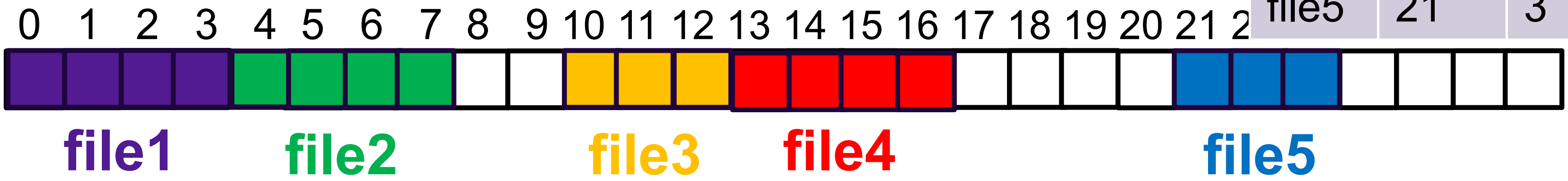
Possible ways to allocate files:

- **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

Continuous Allocation

	start	size
file1	0	4
file2	4	4
file3	10	3
file4	13	4
file5	21	3

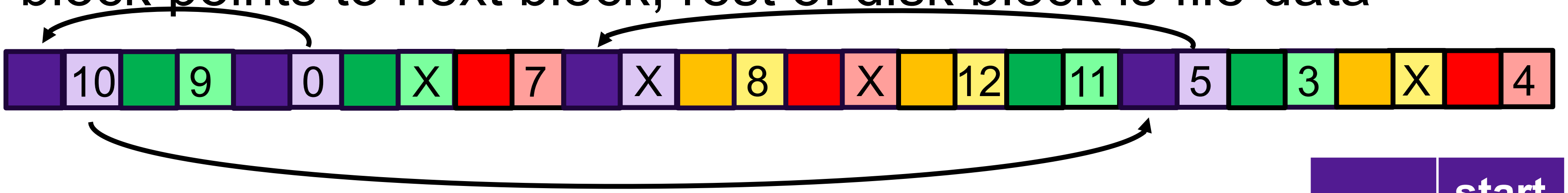
All bytes together, in order



+ **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

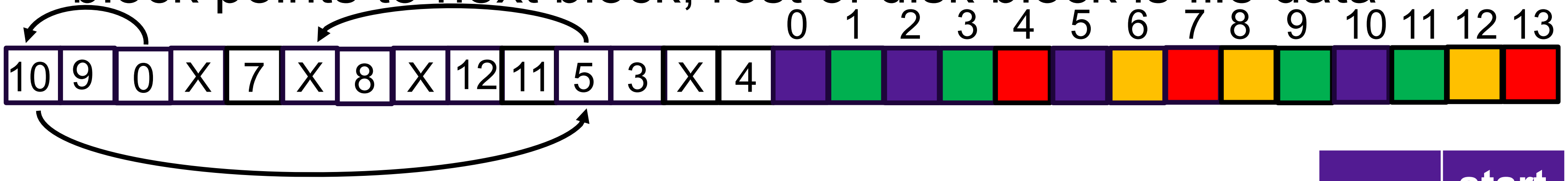
Each file is stored as linked list of blocks: First word of each block points to next block, rest of disk block is file data



	start
file1	2
file2	9
file3	6
file4	13
file5	15

Linked Allocation

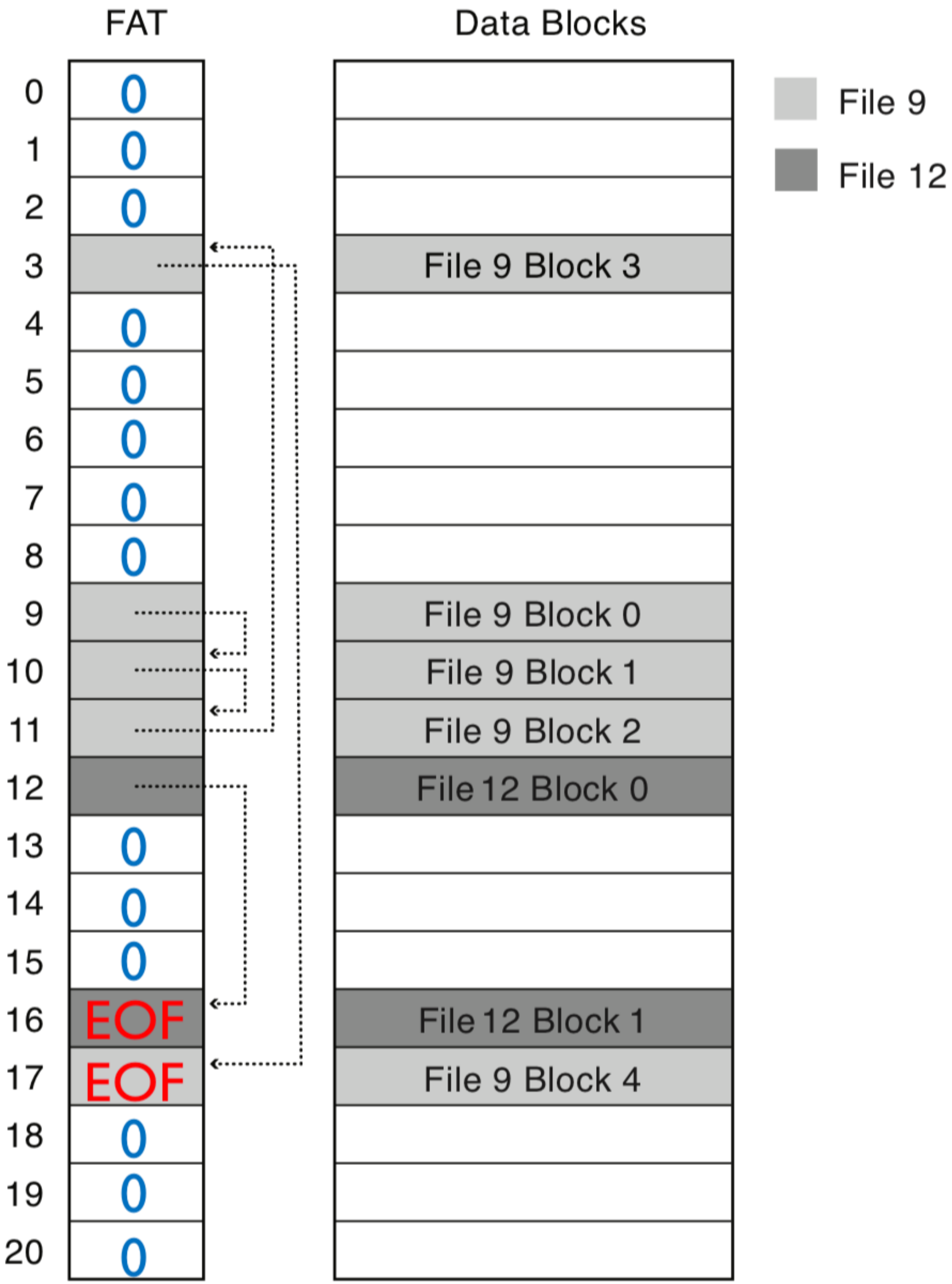
Each file is stored as linked list of blocks: First word of each block points to next block, rest of disk block is file data



	start
file1	2
file2	9
file3	6
file4	13
file5	15

FAT File System

- Developed by Microsoft for MS-DOS
- decoupled linked allocation
- 1 FAT entry per block ("next pointer")
  - EOF for last block
  - 0 indicates free block
- low-level file name = FAT index of first block in file



Evaluating FAT

How is FAT good?

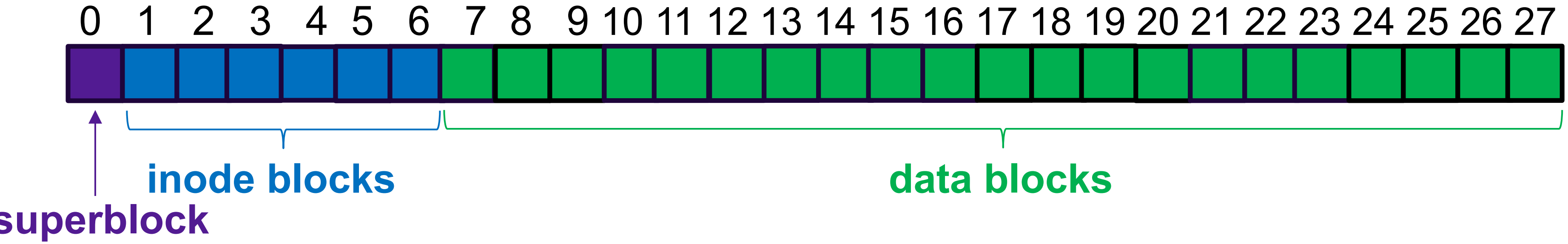
- Simple: state required per file: start block only
- Widely supported
- No external fragmentation
- block used only for data

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)

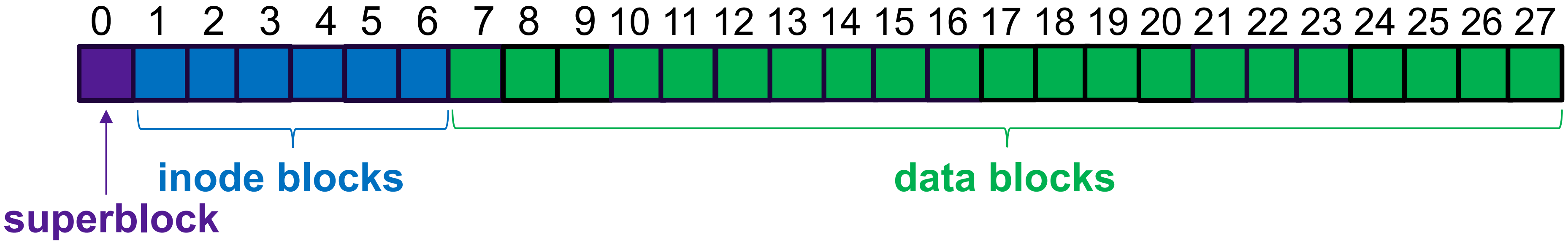
- tree-based, multi-level index



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

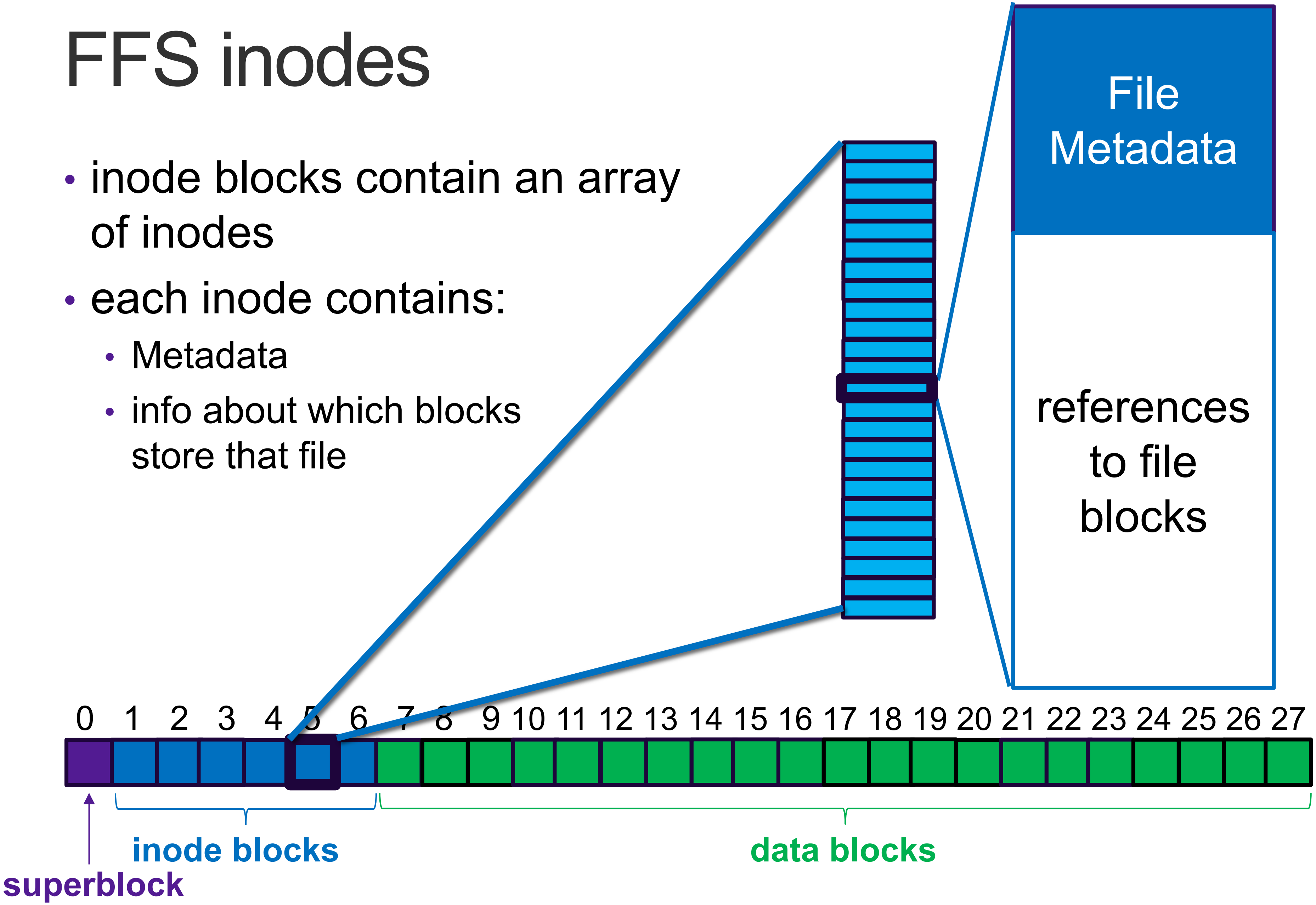
FFS Superblock

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



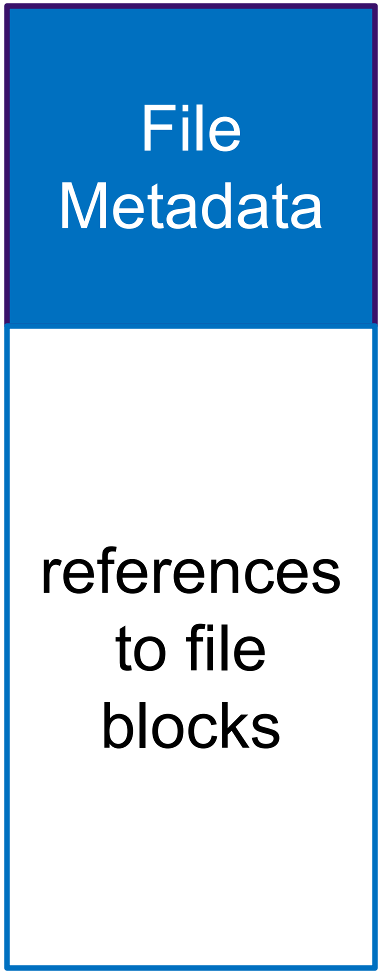
FFS inodes

- inode blocks contain an array of inodes
- each inode contains:
  - Metadata
  - info about which blocks store that file



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



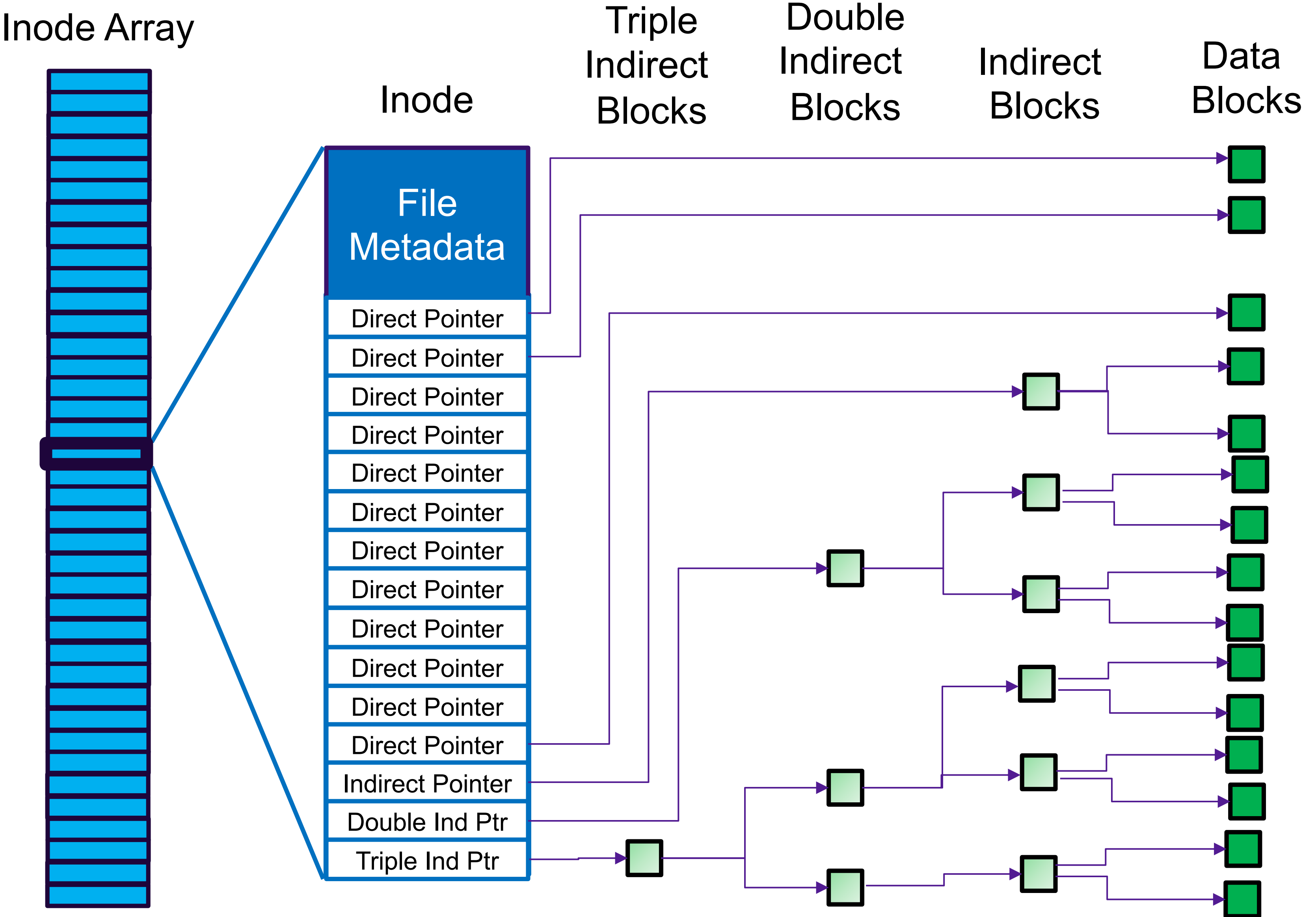
File Metadata

references to file blocks

Each "Pointer" is a block number, not a memory address

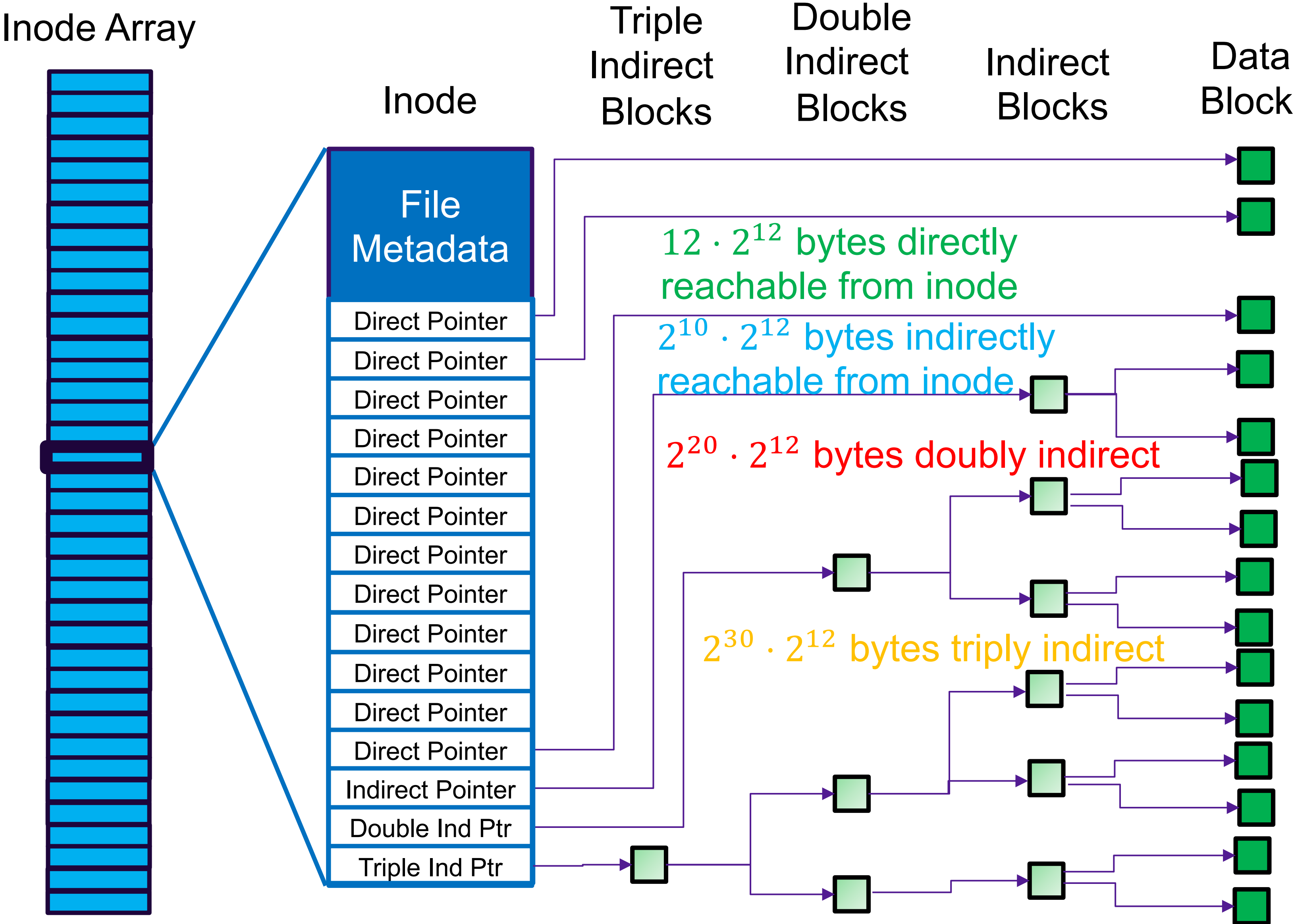
Indirect blocks contain arrays of block numbers

FFS Index Structures



Assume: blocks are 4KB (2^{12} bytes)
block numbers are 4 byte values

Max File Size



$12 \cdot 2^{12}$ bytes directly reachable from inode

$2^{10} \cdot 2^{12}$ bytes indirectly reachable from inode

$2^{20} \cdot 2^{12}$ bytes doubly indirect

$2^{30} \cdot 2^{12}$ bytes triply indirect

Exercise 2: Inode Structures

Assume we are using the inode structure we just described, and assume again that each block is 4K ((2^{12})) and that each block reference is 4 bytes.

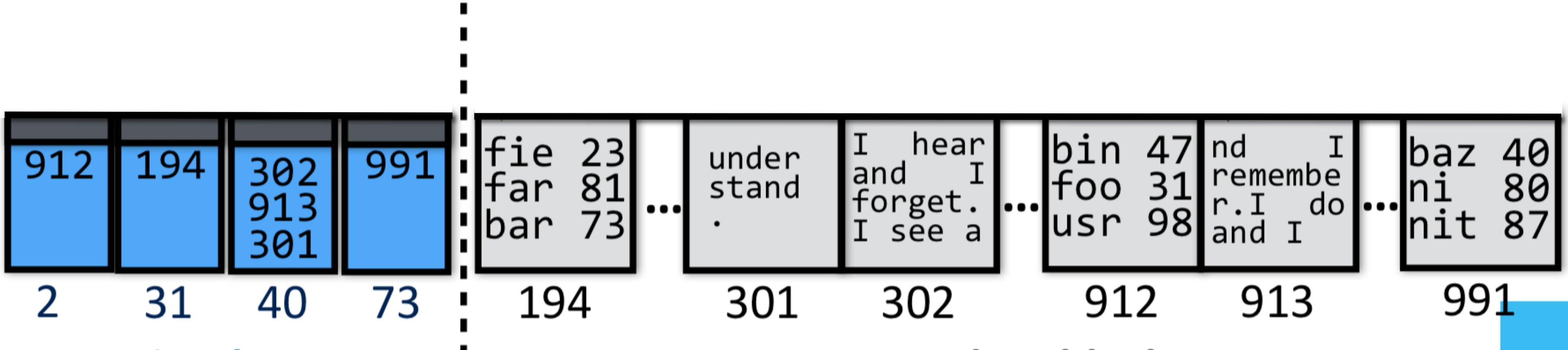
- Which pointers in the inode of a 32KB file would be non-null?
- Which pointers in the inode of a 47MB file would be non-null?

FFS Directory Structure

- Originally: directory was array of 16 byte entries
  - 14 byte file name
  - 2 byte i-node number
- Now: implicit list. Each entry contains:
  - 4-byte inode number
  - Full record length
  - Length of filename
  - Filename
- First entry is “.”, points to self
- Second entry is “..”, points to parent inode

Exercise 3: Indexed Allocation

Which inodes and data blocks would need to be accessed to read (all of) file /foo/bar/baz?



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

Implementation Basics

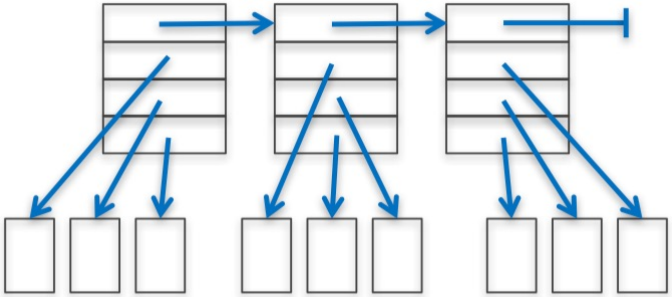
- Directories: file name -> low-level names (i.e., file numbers)
- Index structures: file number -> block
- Free space maps: find a free block (ideally nearby)

Free List

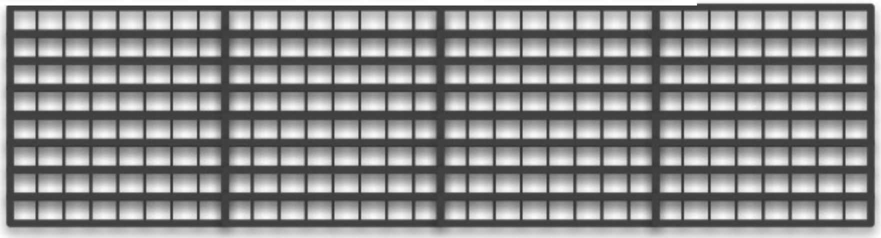
To write files, need to keep track of which blocks are currently free

How to maintain?

- linked list of free blocks
  - inefficient (why?)



- linked list of metadata blocks that in turn point to free blocks
  - simple and efficient



- bitmap
  - actually used

Problem: Poor Performance

- In a naïve implementation of FFS, performance starts bad and gets worse
- One early implementation delivered only 2% disk bandwidth
- The root of the problem: poor locality
  - data blocks of a file were often far from its inode
  - file system would end up highly fragmented: accessing a logically continuous file would require going back and forth across the

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)
- Performance optimizations (e.g., locality heuristics)

Performance Optimizations

- **Grouped Allocation:** disk organized into groups that are (temporally) close, try to allocate all file blocks in same group
- **Defragmentation:** periodically rearrange files to improve locality
- **Page Cache:** to reduce costs of accessing files, cache file contents in memory (e.g., device data, memory-mapped files)
- **Copy-on-write (COW):** create new, updated copy at time of update
- **Write Buffering:** buffer writes and periodically flush to disk