Operating Systems

(Sistemas Operativos)

File System Interface and Design

University of Minho

2024-2025

What will we learn?

Persistence

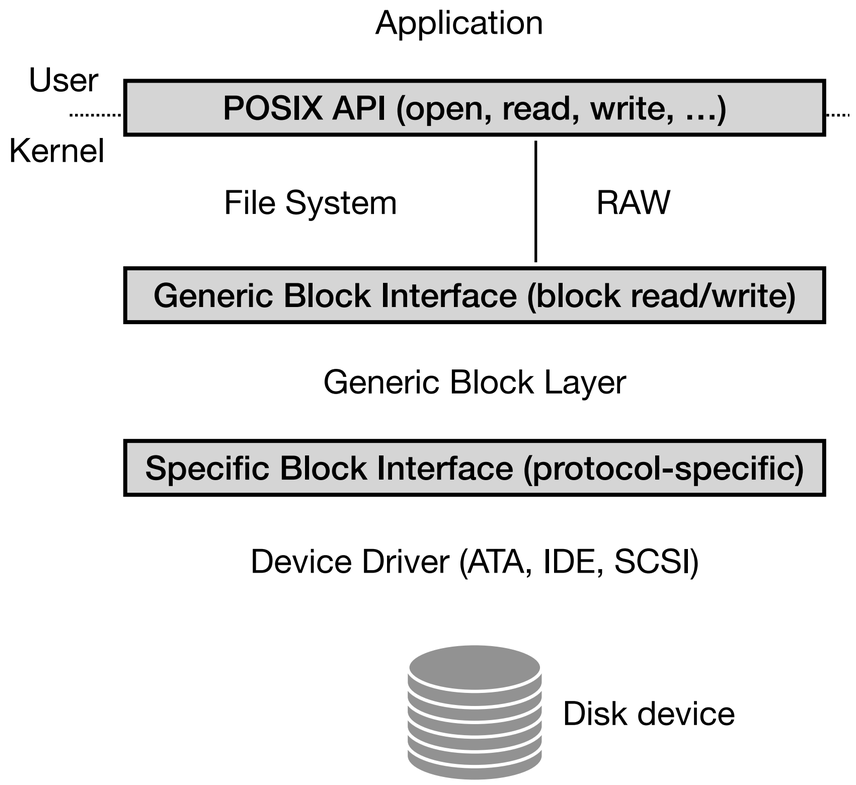
- Memory is volatile, meaning that data is lost upon a reboot (or crash) of the computer. Disk devices (e.g., HDDs, SSDs) allow programs to store their data persistently
  - How does the OS abstract the access (API) to the low-level device drivers of these disks?
  - How does the OS make disk operations efficient when multiple programs use the same device simultaneously?
  - We typically access our data using a file-based abstraction (files, folders, …).
    How does the OS enable such abstraction?

- We will study the interface and design of file systems to answer these questions

Storage Interfaces

The most common ones

- **Block Device**
  - Data is managed as a set of blocks (closest abstraction to the disk)
  - Solutions: Linux block device, iSCSI
- **File System (FS)**
  - Data is managed as a hierarchy of directories and files (the interface most users rely on their personal computers)
  - Solutions: Ext4, ZFS, NFS
- **Key-value Store**
  - Data is managed as key-value pairs (i.e., the key is a unique identifier, and the value is the content)
  - Solutions: LevelDB, RocksDB



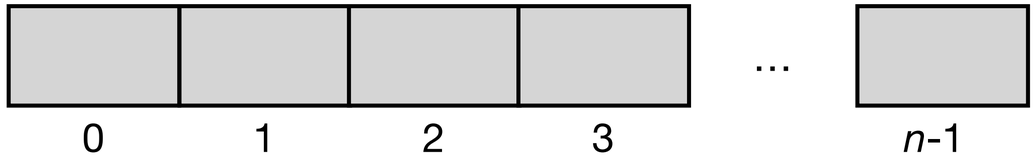
A simplified Linux File System stack

File System Abstractions

The file

- **File:** a linear array of bytes that can be read and written
  - Typically, users are only aware of the **high-level user-readable name** of a file (e.g., foo.txt)
  - Uniquely identified by a **low-level** name, the **inode number (i-number)**
- User-readable names are commonly composed by the file name (e.g., foo, bar) and the type (e.g., .txt, .mp3, .jpg)
  - Usually, this is just a **convention** (e.g., a file named main.c may not contain C code)
  - Most OSs know little about the file's structure (e.g., whether it is a picture, text file, code)

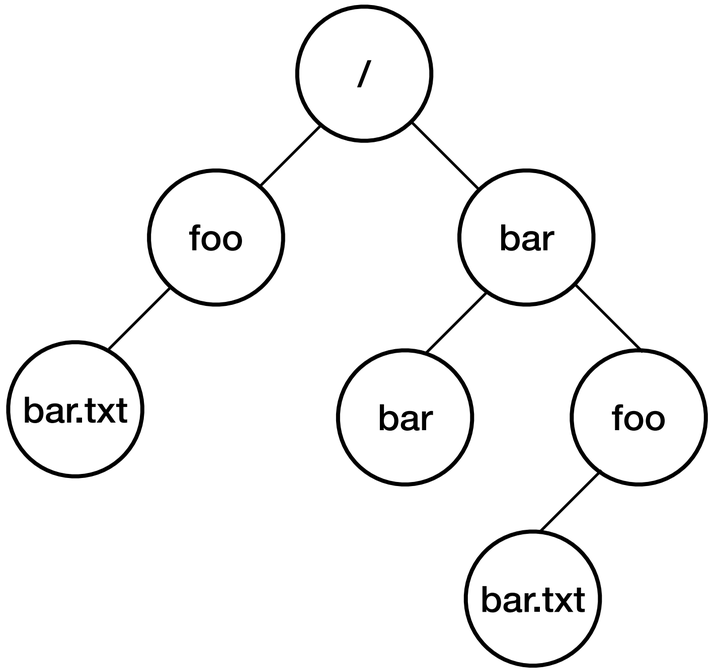
Representation of a file
(an array of bytes 0 to *n*-1)



File System Abstractions

The directory

- **Directory:** a container of files and other directories (also identified by a **low-level i-number**)
- A directory contains a list of pairs, each mapping a **user-readable name** to its **i-number** (i.e., each referring to a file or another directory)
  - **Example:** assuming the file *bar.txt* has i-number *10*, the directory *foo* where the file resides has the pair (*bar.txt*, *10*)
- By placing directories within directories, one can build **directory trees** (or **directory hierarchies**)
  - The hierarchy starts at the **root directory** (*e.g.*, / in UNIX) and uses a **separator** to name subsequent sub-directories
- A file **absolute pathname** is the full path from the root directory until that file (e.g., */bar/foo/bar.txt*)
  - Files and directories may have the same name as long as they are in different locations in the directory tree



Directory tree

File System Interface

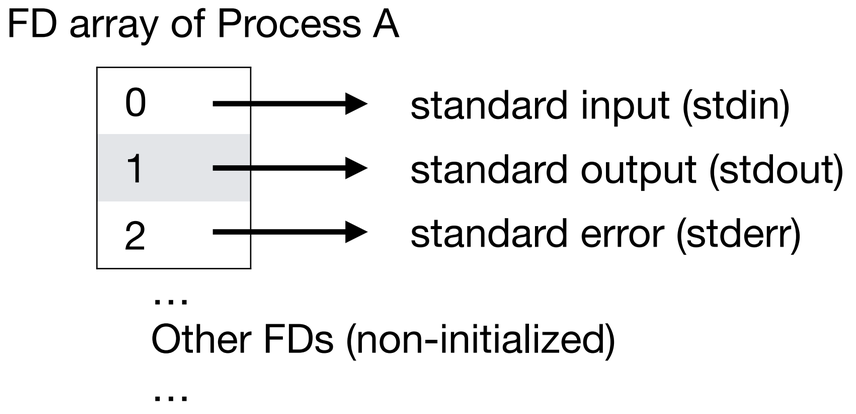
Opening and creating files

- The *int open(const char *pathname, int oflag [, mode])* system call allows opening and creating files
  - **pathname:** the user-readable file's pathname. It can be an absolute or relative (to the current working directory) pathname
  - **oflag** - flags specifying the opening mode
  - **mode** - permissions of the file. It must be used when creating a file (e.g., 0600 gives permissions for the owner of the file to read and write it)
- Some interesting **flags** (check the man page for more!)
  - O_WRONLY, O_RDONLY, O_RDWR - open the file for writing, reading, or both, respectively
  - O_CREAT - create the file if it does not exist
  - O_TRUNC - truncate the file size to zero bytes thus removing any existing content
  - O_APPEND - append file on each write
- Open returns a **file descriptor** (or -1 in case of error)

File System Interface

The File Descriptor (FD)

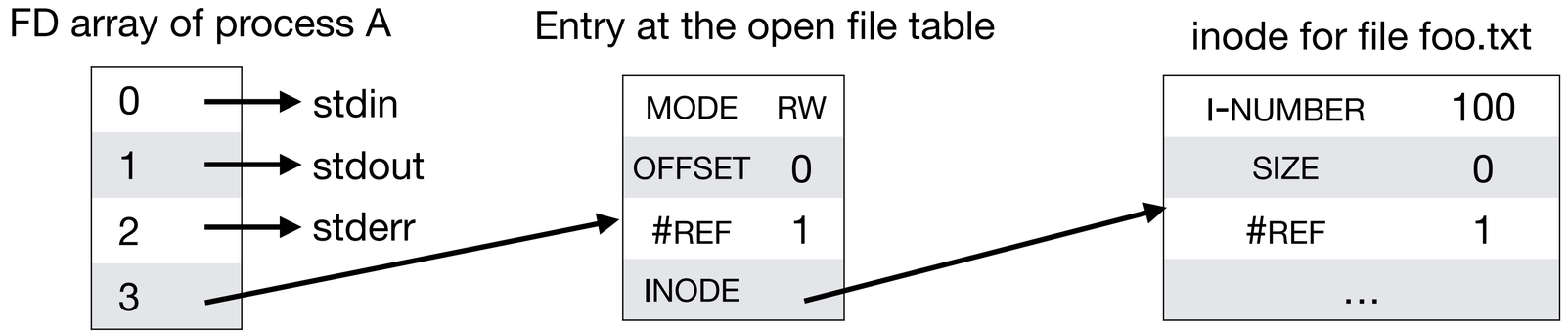
- A **File Descriptor (FD)** is an integer that can be used to read and/or write the file
  - Think of it as a pointer to an object that other system calls use to access the file
  - FDs can represent other Input/Output resources (e.g., pipes, other I/O devices)
- Each process maintains an **array of file descriptors** (also referred to as file descriptor table)
- When a process starts running, this process table already has **standard FDs** initialized for
  - the process to read input, for example, from the keyboard (standard input - FD 0)
  - to write output to the screen (standard output - FD 1)
  - to write error messages (standard error - FD 2)
- For simplicity, let's assume FDs 0, 1, and 2 point to some OS structure(s) that actually contains the information to read from stdin and write to stdout and stderr



File System Interface

Example: opening a file

- Process A executes the code *int fd = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0600)*
  - The file descriptor is initialized and returned by the *open()* call (i.e., *fd = 3*)
- An entry in the system-wide **open file table** is created, for which file descriptor 3 points to
  - Each entry tracks if the file is readable, writable or both, the current file offset, and some other information
  - In most cases, processes **do not share** open file table's entries. For example, two unrelated processes opening file *foo.txt* will have independent entries (we will discuss some exceptions later on)
- The open file table entry maps to an **inode** entry (identified by its **i-number**)
  - The inode contains, for instance, the file's size along with more information that we will discuss later on



File System Interface

Writing and reading files

- The *ssize_t write(int fd, const void *buf, size_t nbyte)* system call allows writing bytes to a file descriptor
  - **fd:** the file descriptor
  - **buf:** memory buffer with content to be written
  - **nbyte:** number of bytes to write from the buffer[1]
  - It returns the **number of bytes written** or an **error** (-1)
- The *ssize_t read(int fd, void *buf, size_t nbyte)* system call allows reading bytes from a file descriptor
  - **fd:** the file descriptor
  - **buf:** memory buffer to where the content is read
  - **nbyte:** max number of bytes to read[1]
  - It returns the **number of bytes read** or an **error** (-1)

[1] Be careful not to specify a number of bytes to read/write larger than the memory allocated for the buffer!

File System Interface

Writing and reading files

- The *ssize_t write(int fd, const void *buf, size_t nbyte)* system call allows writing bytes to a file descriptor
  - **fd:** the file descriptor
  - **buf:** memory buffer with content to be written
  - **nbyte:** number of bytes to write from the buffer[1]
  - It returns the **number of bytes written** or an **error** (-1)
- The *ssize_t read(int fd, void *buf, size_t nbyte)* system call allows reading bytes from a file descriptor
  - **fd:** the file descriptor
  - **buf:** memory buffer to where the content is read
  - **nbyte:** max number of bytes to read[1]
  - It returns the **number of bytes read** or an **error** (-1)

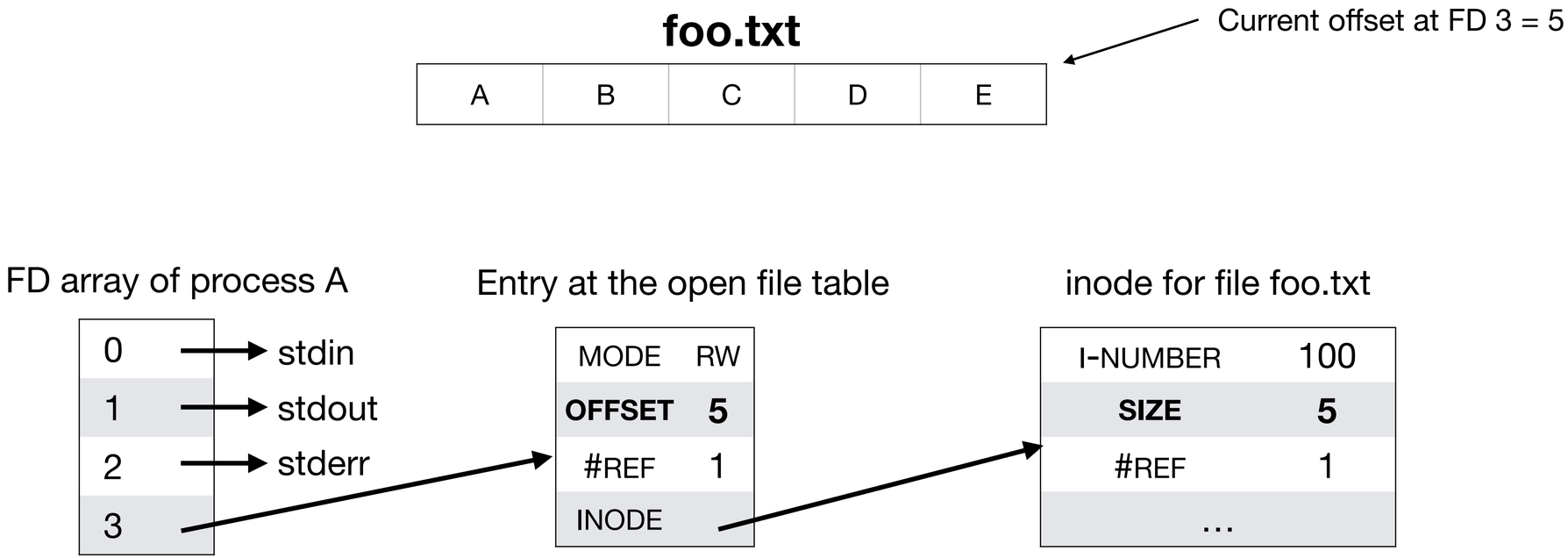
Alternatively, writes and reads to files can be done through the ***mmap()*** system call. It maps the file into memory so that it can be manipulated with memory instructions

[1] Be careful not to specify a number of bytes to read/write larger than the memory allocated for the buffer!

File System Interface

Example: writing a file

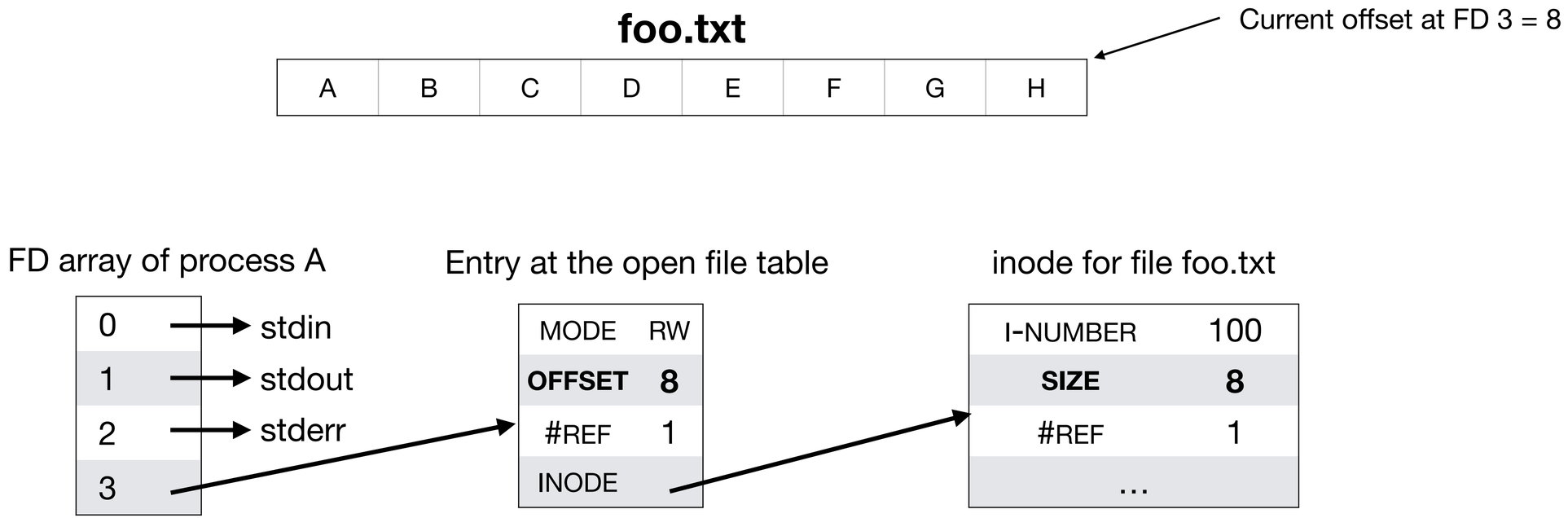
- Process A now executes a write to the file - *ssize_t res = write(fd, “abcde”, 5)*
  - The *write()* call writes the 5 bytes into the file and returns (i.e., res=5)
  - The **offset** at the open file table and the **size** at the inode are updated



File System Interface

Example: writing a file

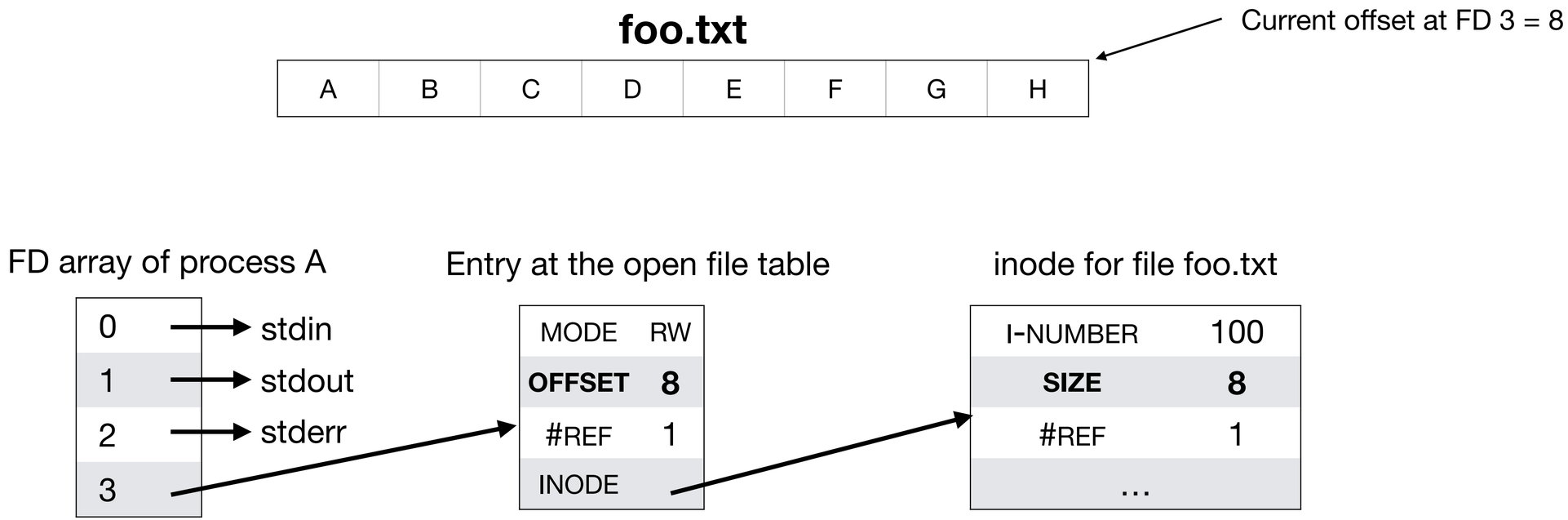
- Process A now executes another write to the file - *res = write(fd, “fgh”, 3)*
  - The *write()* call writes the 3 bytes into the file (starting at offset 5) and returns (i.e., res=3)
  - The **offset** at the open file table and the **size** at the inode are updated



File System Interface

Example: reading a file

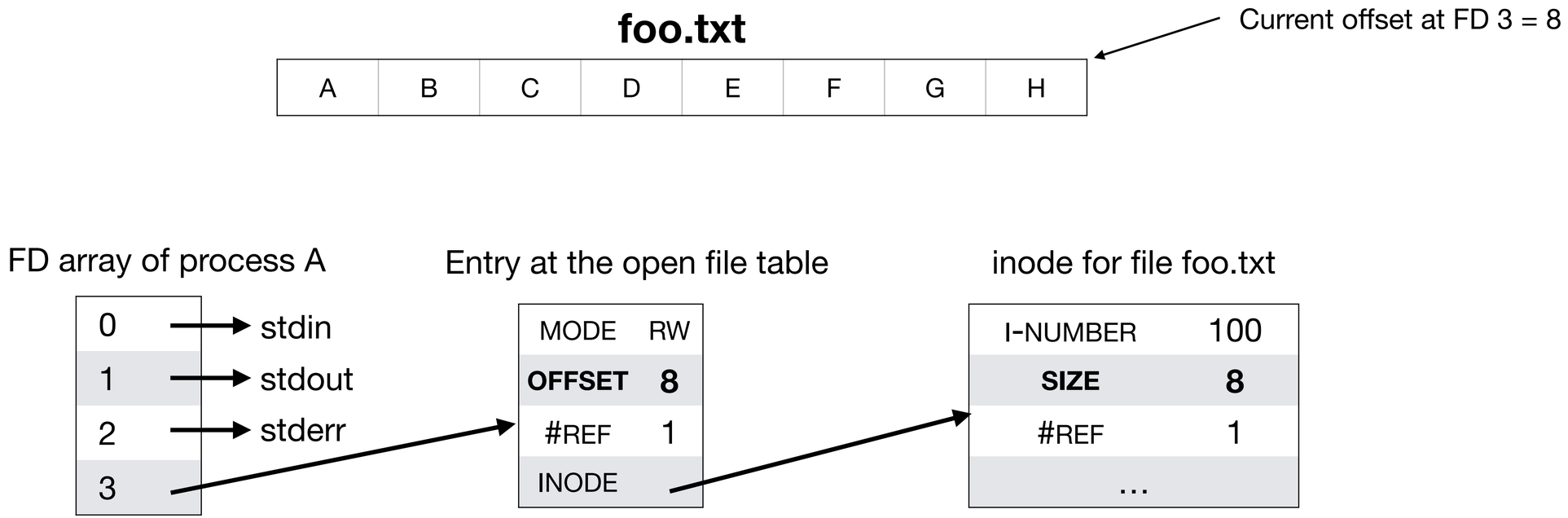
- Process A now executes a read to the file - *res = read(fd, buf, 8)*
  - What will be the return value of the *read* call?



File System Interface

Example: reading a file

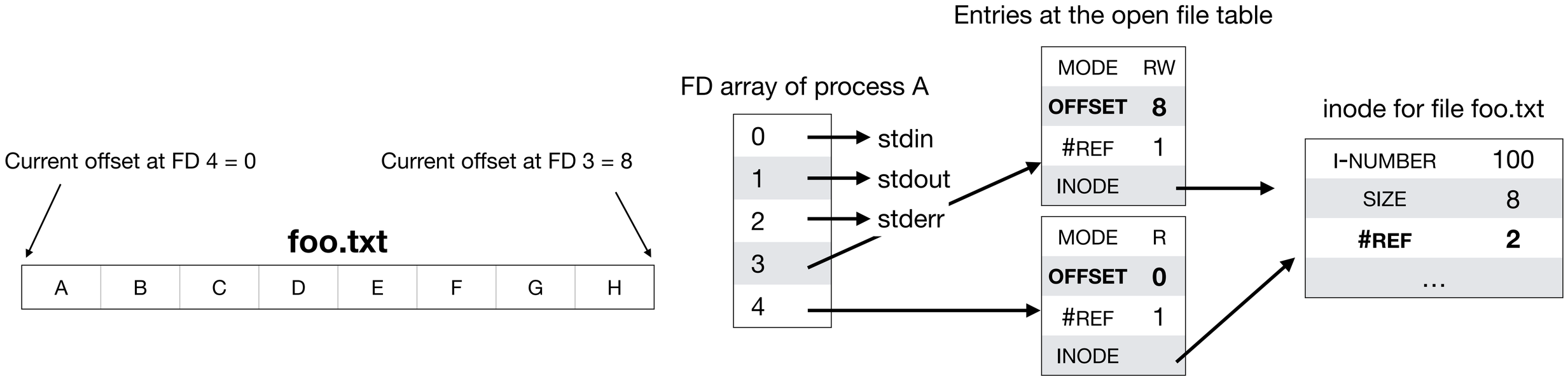
- Process A now executes a read to the file - *res = read(fd, buf, 8)*
  - What will be the return value of the *read* call?
    - It will **return 0**, the program is reading (the offset is positioned) at the **end of the file**!



File System Interface

Multiple file descriptors

- **Option 1:** To read the full content of the file one can issue another *open()* call
- For example, by executing the code *int fd1 = open("foo.txt", O_RDONLY)*
  - The *open()* will create a new entry at the **open file table** that will have its offset set to 0
  - Issue the *read()* call to the newly opened descriptor (i.e., *fd1 = 4*)
  - Note that the **inode is shared** across the two file table entries



File System Interface

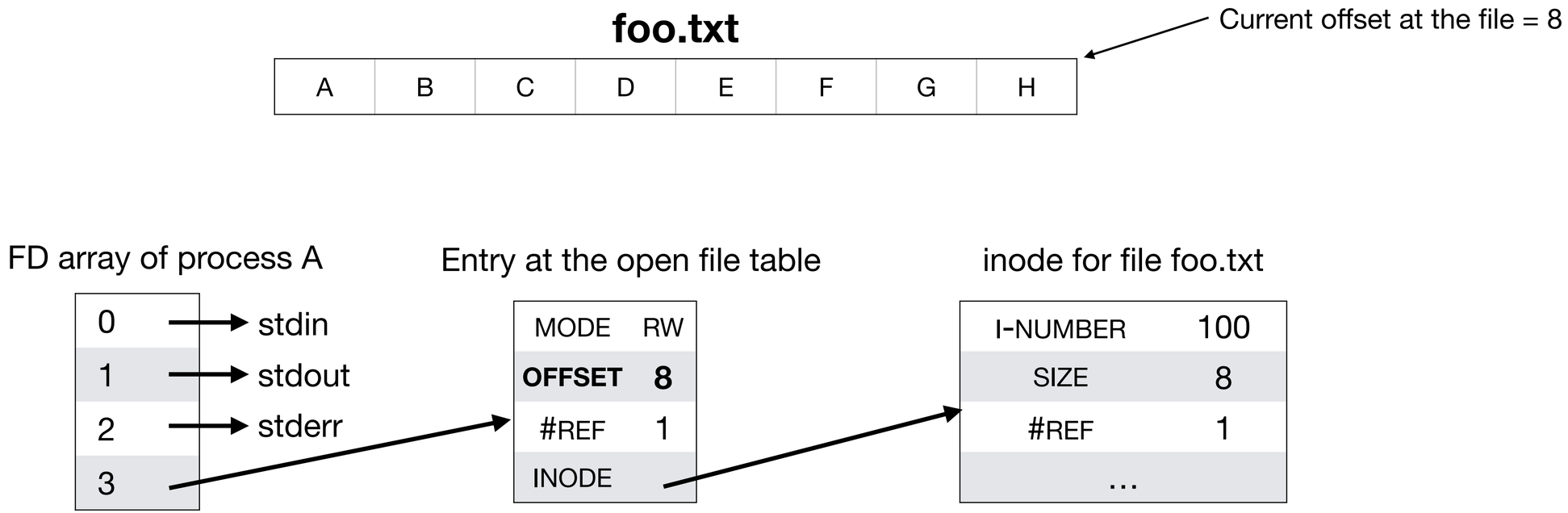
Random file access

- **Option 2:** Do a random access to the descriptor's offset 0
- The *off_t lseek(int fd, off_t offset, int whence)* system call allows changing the offset for a given file descriptor
  - **fd:** the file descriptor
  - **offset:** the number of bytes to move forward or backward (it can be negative)
  - **whence:** the **base offset** at the file to which the **offset argument** should be added to
    - SEEK_SET - position at the beginning of the file and apply the offset argument
    - SEEK_END - position at the end of the file and apply the offset argument
    - SEEK_CUR - apply the offset argument at the current location (current offset) in the file
  - It returns the **resulting offset at the file** or an **error** (-1)
- **Note:** *lseek()* does not perform a **disk seek**. It only changes the offset variable in the file table entry. A disk seek is performed when accessing sectors in distinct tracks

File System Interface

Example: seeking and reading a file

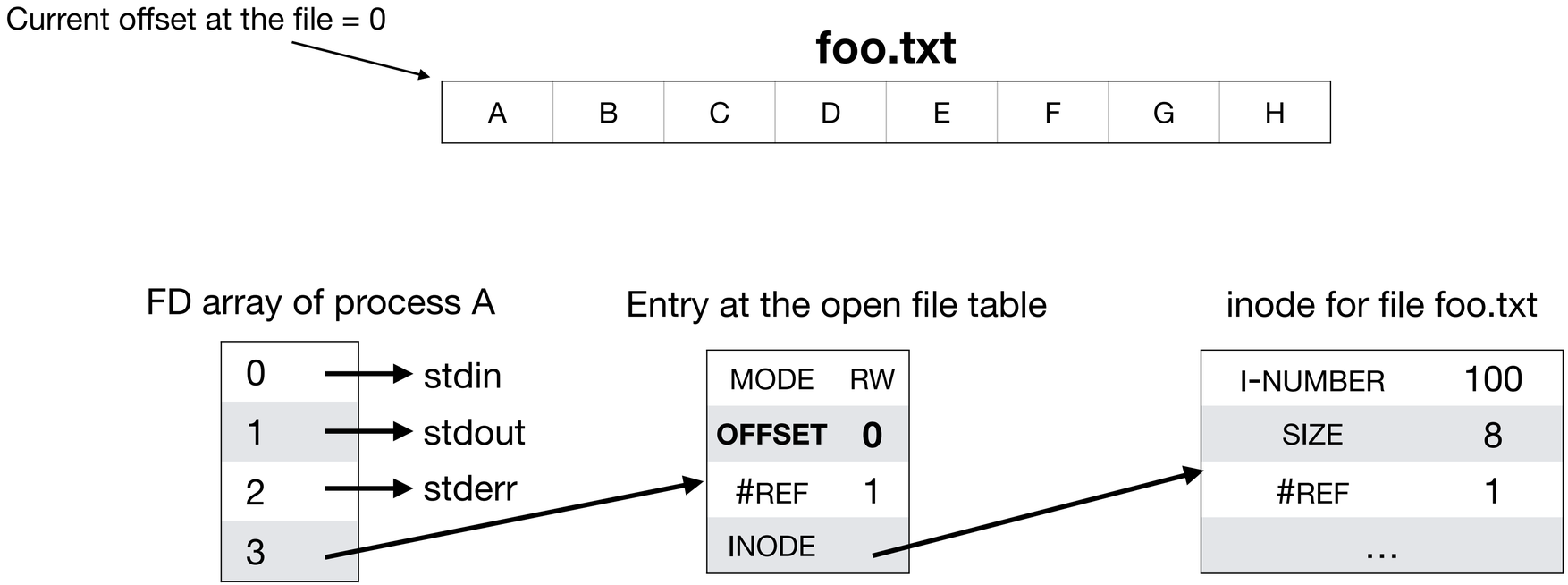
- Process first executes *lseek(fd, 0, SEEK_SET)*



File System Interface

Example: seeking and reading a file

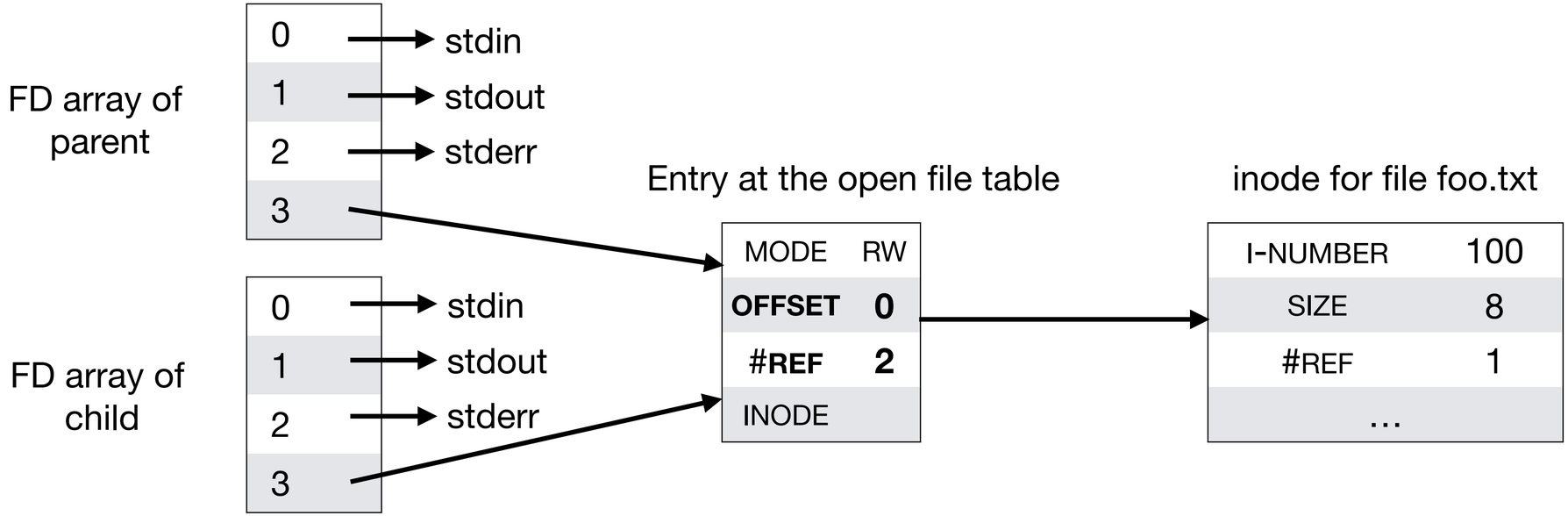
- Process first executes *lseek(fd, SEEK_SET, 0)*
  - This means the **offset** is positioned at the beginning of the file
  - Now, one can execute the *read()* call



File System Interface

Shared open file table entries with fork

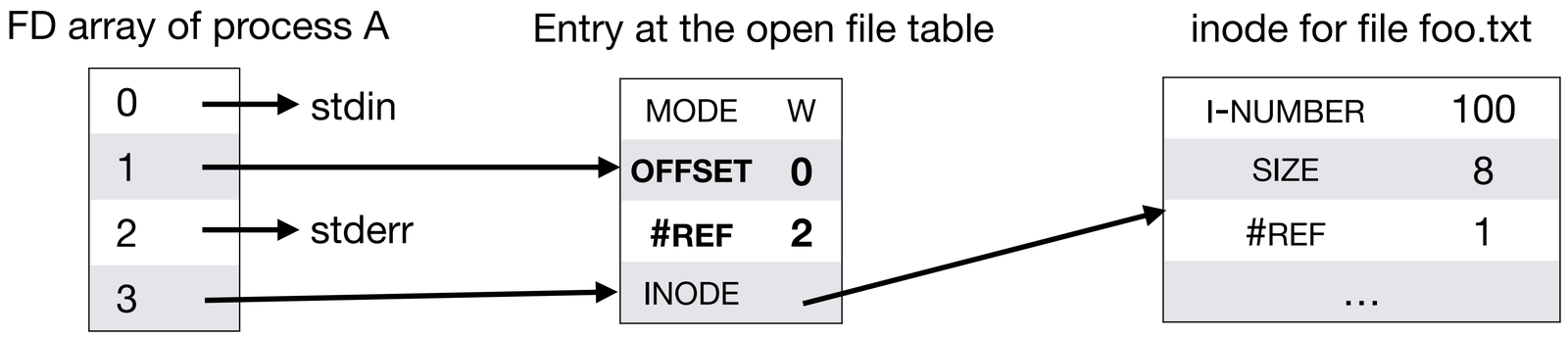
- **Unrelated processes** opening the same file have **their own file table entries** (**inode is shared**)
- **One exception:** When a process calls *fork()*, the child process has its **own array of file descriptors** but **shares the open file table entry** with the parent
  - **Be careful:** multiple related processes reading, writing, and seeking a file, may update the **offset field concurrently**!



File System Interface

Shared open file table entries with dup

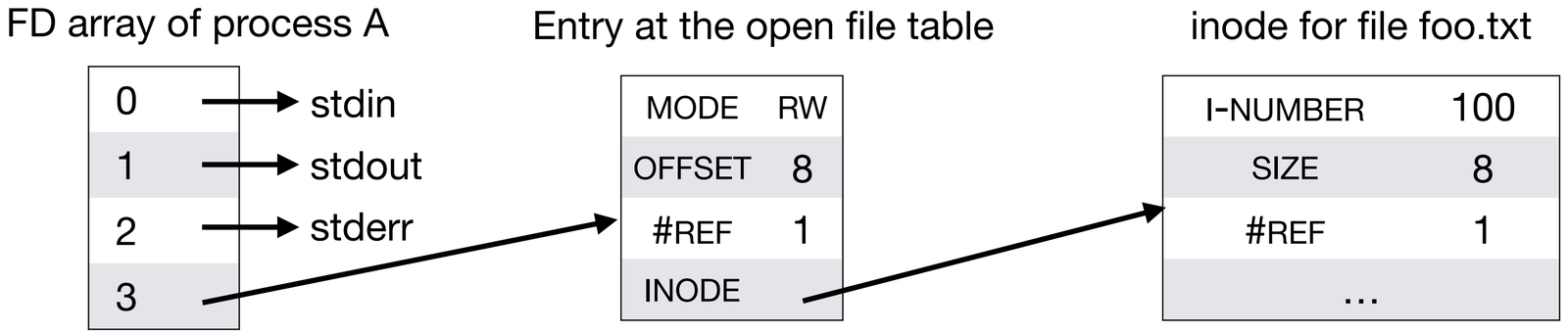
- **Another exception**, where file table entries are shared is when using the family of *dup()* system calls
  - These duplicate an existing file descriptor by making its copy point to the same file table entry
- *int dup2(int oldfd, int newfd)* allows duplicating file descriptor **oldfd** into **newfd**
- The example below shows the result of executing *dup2(3,1)*



File System Interface

Closing and removing files

- The *int close(int fd)* system call “frees” the corresponding descriptor (**fd**) at the per-process array. If the entry in the file table is no longer referenced, it is deleted
  - The **inode is not deleted**!
- The *int unlink(const char *pathname)* only **deletes the inode** associated with the **pathname** when the number of references to it reaches 0
  - Remember that multiple unrelated processes may have the same file opened



File System Interface

Flushing file system buffered data to disk

- When a program calls *write()* it asks the file system (FS) to persist data
  - However, the FS will **buffer** written data in memory for some time (e.g., 5 seconds, 30 seconds…)
  - Any Idea on why?
- If the machine crashes meanwhile, buffered data gets lost…
  - Applications such as databases require strong persistence guarantees
- The *int fsync(int fd)* call forces the FS to write all dirty data of a given file descriptor (**fd**) to disk
  - In some cases, one also needs to sync the directory where the file resides (*i.e.*, if the file is newly created, the directory must know that it exits…) This is often overlooked, leading to bugs…

File System Interface

Renaming files

- The *int rename(const char *oldpath, const char *newpath)* system call renames a file (i.e., changes its **user-readable name**)
  - This is how the **mv** command is implemented
- In many file systems, this operation is **atomic** (i.e., if the system crashes during a *rename*, the file will either be named the old name or the new one)
  - Very useful to support applications that require atomic updates to certain files
  - E.g., when users update text in the middle of a file, some text editors create a temporary file for rearranging the content. When this is done, the temporary file is renamed again into the original one

File System Interface

Getting file information

- The *stat()* and *fstat()* system calls allow consulting metadata about a file, given its filename or descriptor, respectively
- Information is provided at a **stat structure**
  - The information contained in this structure corresponds to the one stored at the file's **inode**

```c
struct stat {
  dev_t st_dev; // ID of device containing file
  ino_t st_ino; // inode number
  mode_t st_mode; // protection
  nlink_t st_nlink; // number of hard links
  uid_t st_uid; // user ID of owner
  gid_t st_gid; // group ID of owner
  dev_t st_rdev; // device ID (if special file)
  off_t st_size; // total size, in bytes
  blksize_t st_blksize; // blocksize for filesystem I/O
  blkcnt_t st_blocks; // number of blocks allocated
  time_t st_atime; // time of last access
  time_t st_mtime; // time of last modification
  time_t st_ctime; // time of last status change
};
```

File System Interface

Manipulating directories

- Directories **cannot be written to**. These are updated indirectly by creating files or other directories in it
  - m*kdir()* creates an empty directory
  - *opendir() opens* the directory
  - *readdir()* allows reading each entry contained within the directory
  - *closedir()* closes the directory
  - *rmdir()* deletes the directory (fails if the directory is not empty)

File System Interface

Access Control

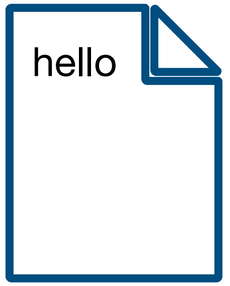
- **Files** and **directories** are shared across several users
- Permission bits (rw-r—r--) allow specifying what the **owner** of the file, someone in a **group**, and **other** users can do
  - Permissions can be defined in an octal format (0644 for the example above) Explore https://chmod-calculator.com/ to know more
  - The ***Chmod*** command can be used to change such permissions
- Some file systems provide other protection mechanisms
  - AFS, for instance, allows defining an **Access Control List (ACL)** per directory

System call tracing

The strace tool

- Strace is a program to trace system calls
  - It can be used with any program, not requiring its source code
  - Useful for observing what OS services the program is requesting
- **Example:** *strace cat foo.txt*

```
prompt> strace cat foo.txt
...
open("foo", O_RDONLY|O_LARGEFILE) = 3
read(3, "hello\n", 4096) = 6
write(1, "hello\n", 6) = 6
…
read(3, "", 4096) = 0
close(3) = 0
...
prompt>
```



foo.txt
(contains one line with the word “hello”)

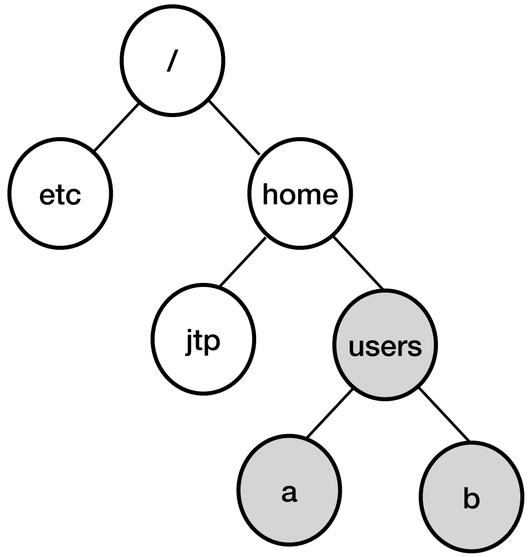
Setting up a File System

Making and mounting the FS

- To make a file system, most FSs provide a tool usually referred to as ***mkfs***.
  - Receives as input a **device** (e.g., a disk partition /dev/sda1) and a **file system type** (e.g., ext4) and writes an empty file system, starting with the root directory into that disk partition
- The ***mount*** command (which internally uses the ***mount()*** system call) takes an existing directory as the target **mount point** (i.e., the directory where the file system is attached to and made available to the system)

/dev/sda1
(has an unmounted FS with a root directory containing two subdirectories “a” and “b”)

mount /dev/sda1 /home/users



File System

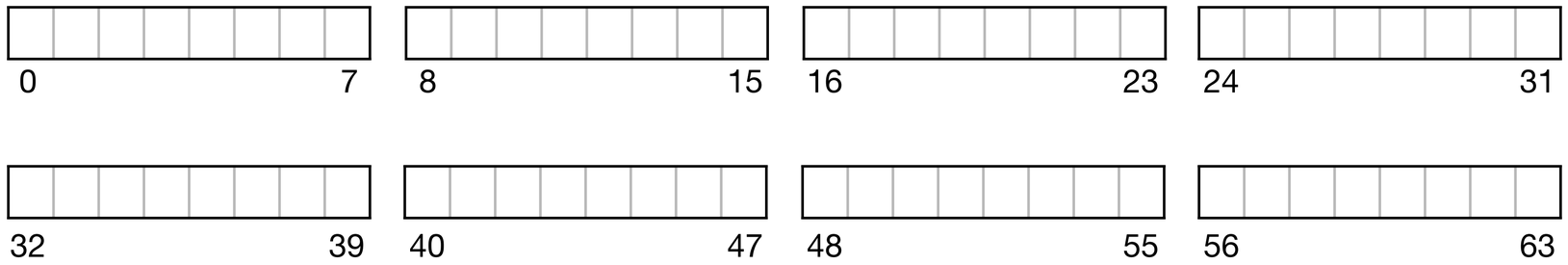
A simplified version of the UNIX file system

- Let's discuss a simple FS implementation: the **Very Simple File System (vsfs)**
- The FS is pure software, i.e., it does not require aid from the hardware
  - But it must know hardware details to be **efficient**!
- There are two important aspects to a file system
  - **Data structures:** what on-disk structures are used by the FS to organize its data and metadata
  - **Access methods:** How are calls like *open()*, *read()*, *write(),* etc mapped to these data structures?

Very Simple File System

Overall organization

- The disk is organized as an array of blocks (e.g., 4KB)

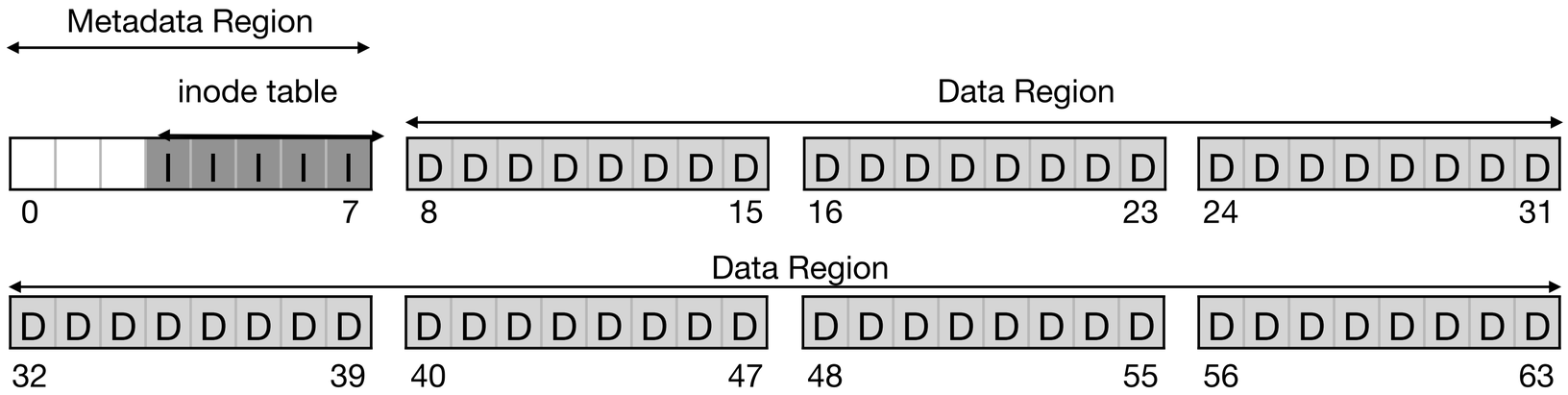


A really small disk with 64 blocks of 4KB

Very Simple File System

Data and metadata regions

- The disk is organized as an array of blocks (e.g., 4KB)
  - The **data region** holds data from the user and applications (e.g., files' content)
  - The **metadata region** holds meta-information about files, and the file system
- The information of each file is stored in a different **inode**
  - Each inode is typically small (e.g., 256 bytes per inode)
  - An **inode table**, holding all the inodes, is stored on the metadata region

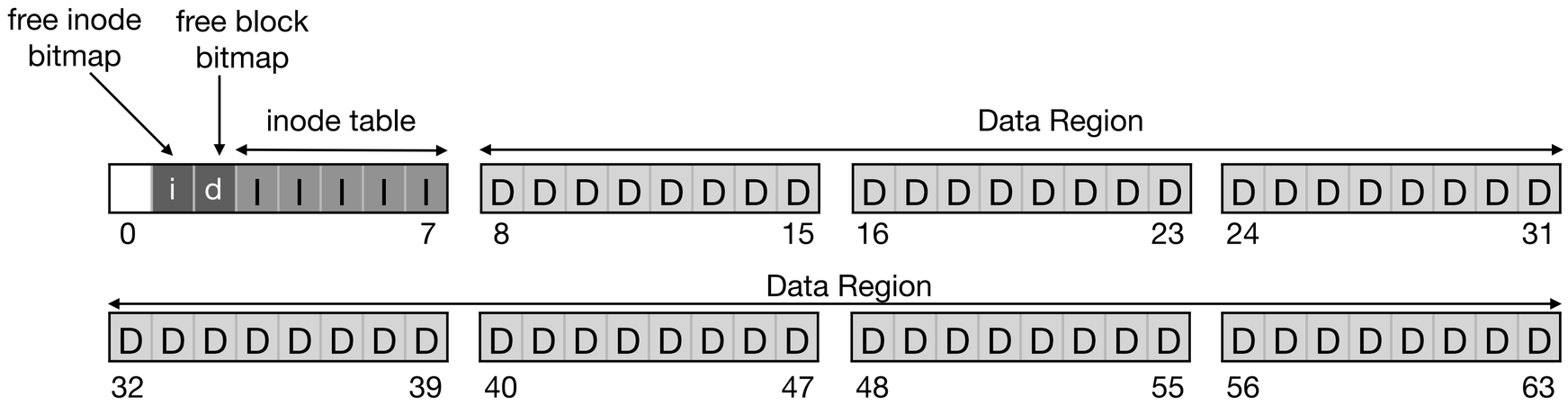


A really small disk with 64 blocks of 4KB

Very Simple File System

Free space management

- The **metadata region** must also hold structures that track free inodes and data blocks
  - **Example:** two simple **bitmaps**, one for inodes and another for data blocks. Each bit at the bitmap represents if a given inode/block is used (1) or free (0)
  - When an allocation is needed, the bitmap must be searched for an unused inode or data block
  - Some FSs (e.g., ext2, ext3) try to find a list of contiguous blocks to allocate when a file is created. Ideas on why using this **pre-allocation** strategy?
- Other structures are possible, for example, a free list where each free inode/block points to the next one

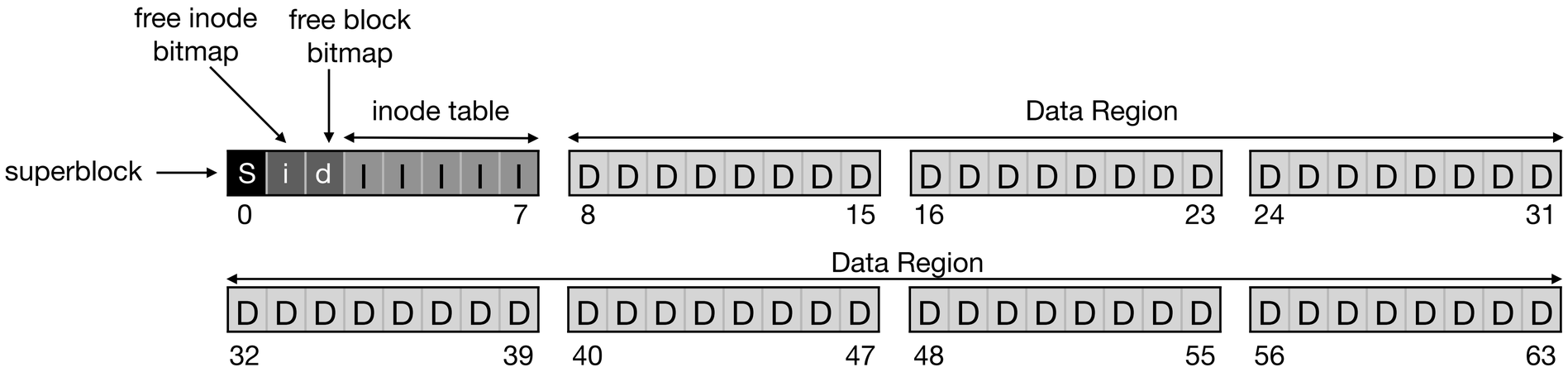


A really small disk with 64 blocks of 4KB

Very Simple File System

Overall organization

- The first block at the metadata region holds the **superblock** that keeps information about the FS organization (e.g., the number of nodes and data nodes, where the bitmaps and inode table begin, …)
  - When mounting the file file system, the OS will read the superblock first

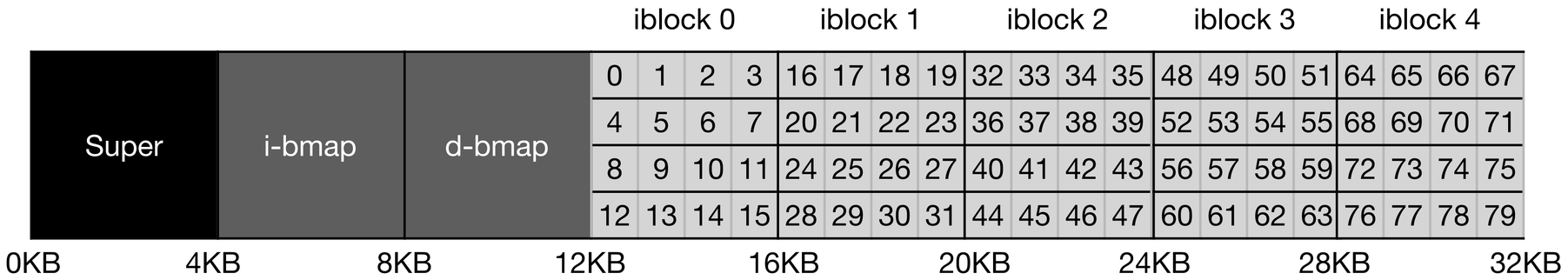


A really small disk with 64 blocks of 4KB

The inode

The inode table

- **inode** is short for index-node (one of the most important FS structures!)
  - Different OSs may name this structure differently
  - Each inode is identified by a number (the **i-number**).
  - Since the **inode table** is located in a fixed disk region, it is easy to quickly find the information from disk for a given inode



A closeup at the inode table

The inode

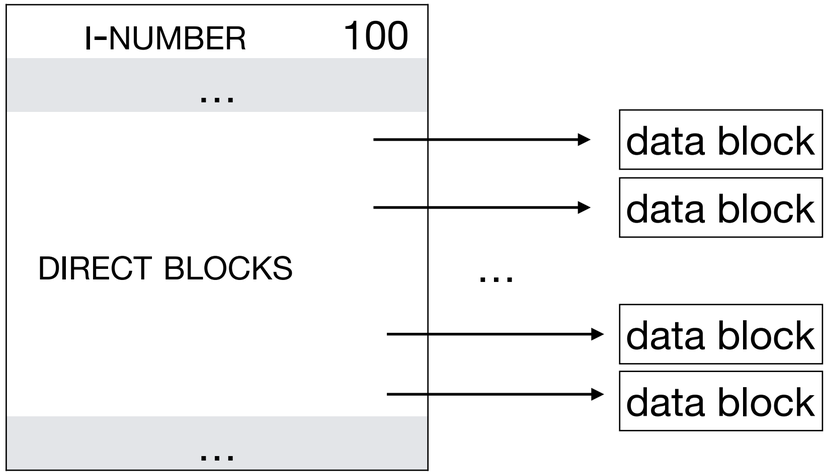
Simplified version of an ext2 inode

Size	Name	What is this inode field for?
2	MODE	CAN THIS FILE BE READ/WRITTEN/EXECUTED?
2	UID	WHO OWNS THIS FILE?
4	SIZE	HOW MANY BYTES ARE IN THIS FILE?
4	TIME	WHAT TIME WAS THIS FILE LAST ACCESSED?
4	CTIME	WHAT TIME WAS THIS FILE CREATED?
4	MTIME	WHAT TIME WAS THIS FILE LAST MODIFIED?
4	DTIME	WHAT TIME WAS THIS INODE DELETED?
2	GID	WHICH GROUP DOES THIS FILE BELONG TO?
2	LINKS_COUNT	HOW MANY HARD LINKS ARE THERE TO THIS FILE?
4	BLOCKS	HOW MANY BLOCKS HAVE BEEN ALLOCATED TO THIS
4	FLAGS	HOW SHOULD EXT2 USE THIS INODE?
60	BLOCK	A SET OF DISK POINTERS (15 TOTAL)
4	FILE_ACL	A NEW PERMISSIONS MODEL BEYOND MODE BITS
4	DIR_ACL	CALLED ACCESS CONTROL LISTS

The inode

Direct pointers to data blocks

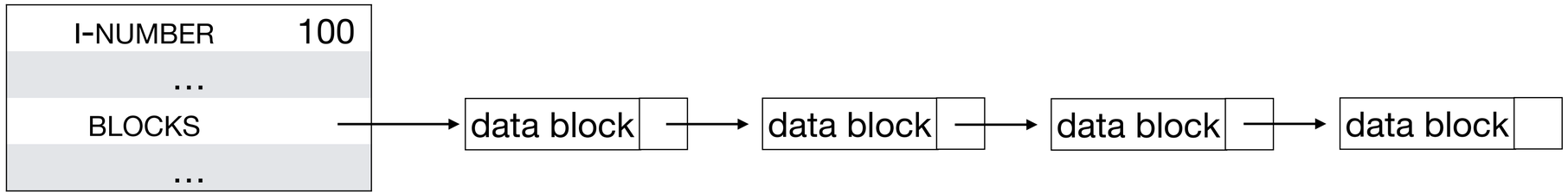
- The inode must point to the corresponding data blocks (i.e., content of the file) (*blocks* field in the previous slide)
- One option is to use **direct** pointers (i.e., each pointing to the address of a data block)
  - Imagine a large file with thousands (or more) 4KB blocks. The size of the inode will increase and may not fit on the space allocated for it…



The inode

List of blocks

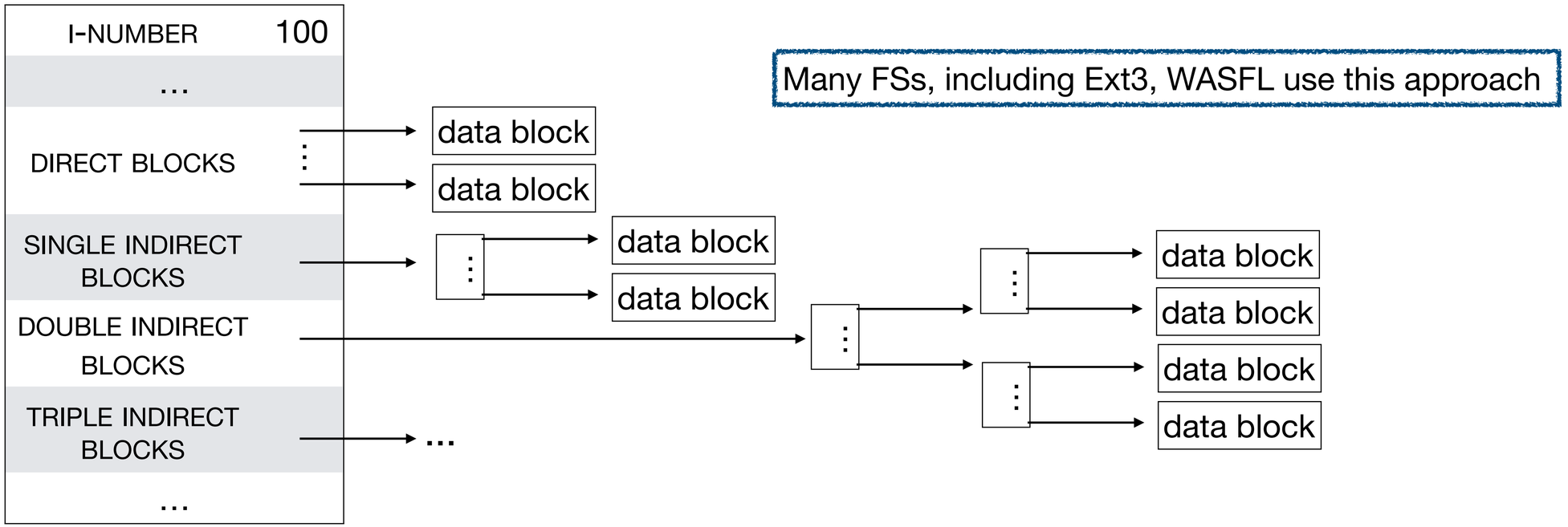
- As an alternative, the inode may contain a **single pointer** to the file's first block, then this block points to the next one, and so on…
- **Problem:** bad for random accesses (if one wants to access the last block…)
  - This is what the basic structure for the classic **FAT file system** in Windows does



The inode

Combining direct and indirect pointers

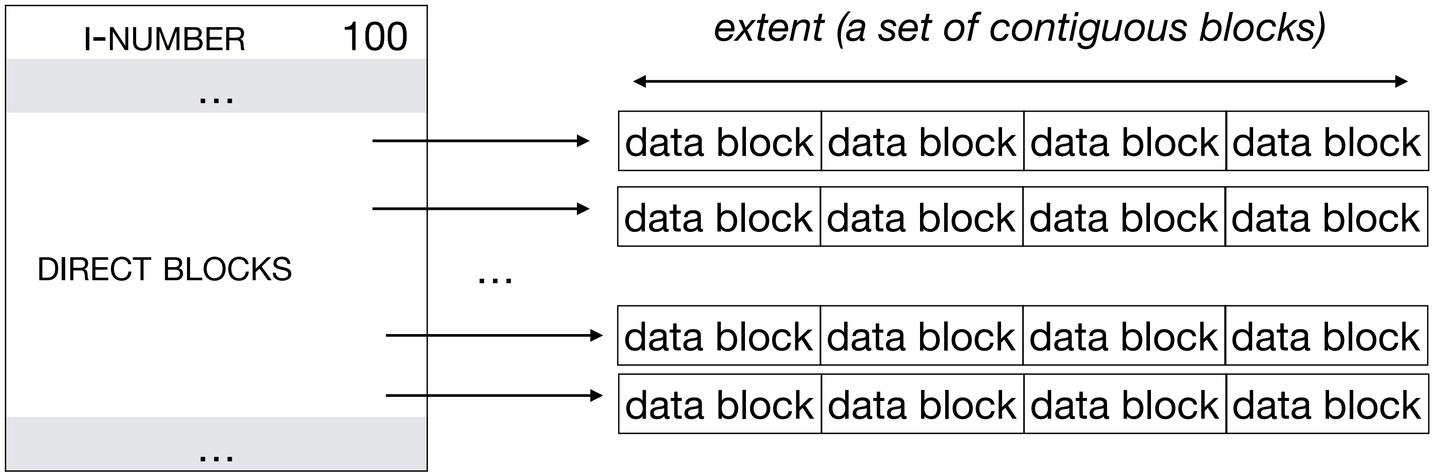
- Another option is to have a combination of **direct** and **indirect** pointers to blocks, i.e., to follow a **multi-level index approach**
  - Small files will have direct pointers to all their blocks, while large files will have single, double or even triple indirect pointers to blocks



The inode

Using extents

- Multi-level indexes may require going through several pointers to reach a block (**performance** penalty), along with allocating blocks for indirect pointers (**space** penalty)
- A final option is to use **extents**, i.e., large sequences of contiguous disk blocks
  - **Advantage:** the number of pointers decreases (one can use only direct pointers)
  - **Disadvantage:** finding contiguous blocks is not always easy



FSs like ext4 and XFS use the extent approach

Directories

Directory organization

- Directories also have inodes stored somewhere in the inode table. The inode type is marked as a **directory** instead of a **regular file**
  - Directory inodes also have **direct (or indirect) pointers** to data blocks[1].
    But what do these data blocks store?

- A **directory data block** contains a list of pairs (i-number, user-readable name) (actually, it may also contain the name length and record length)

data block for folder /home/user (i-number 5)

I-NUMBER	NAME	…	
5	.	…	← the current directory and its i-number (user directory)
2	..	…	← the parent directory and its i-number (home directory)
12	FOO.TXT	…	← i-number of file foo.txt
13	BAR	…	← i-number of directory bar

[1] Other structures are possible such as B-trees (used in XFS), to be searched more efficiently (e.g., when creating a file to know if the name is valid)

Very Simple File System

A file open request

- A request to open file /foo/bar.txt is issued. The file system needs to find the inode of bar.txt, but since it is unknown, it must **transverse** the full pathname to find it
  1. The location of the *root* directory (/) inode is known. By reading it, one gets the pointer to (location of) the corresponding data block(s)
  2. By reading the *root* directory's data block(s), one gets the i-number of the *foo* directory
  3. Now, one must repeat the process above for the *foo* directory to get *bar.txt's* i-number
  4. The inode of *bar.txt* is read into memory, the permissions are checked, and finally, an entry at the open file table is created

- If the **file is being created**, the FS must do steps 1, 2, and 3 above and then
  4. Read (find a free inode) and update (to mark the allocation) the bitmap of free inodes
  5. Write the new file's inode to disk (to initialize it)
  6. Update the data block of the file's directory (to link the user-readable name with the i-number of the file)
  7. Read and write the file's directory inode (to update its information)

Very Simple File System

Read and write requests

- Once opened, reading from the *bar.txt* file requires
  1. Reading again the inode of the file (if it was evicted from memory) to get the block's location
  2. Reading the data block(s)
  3. Updating the file's inode (e.g., the access time) and the open file table's entry (i.e., the offset)
- Writing to a file may require
  1. Reading again the inode of the file (if it was evicted from memory) to get the block's location
  2. Allocating data block(s) (i.e., read and update the data bitmap)
  3. Writing the data block(s)
  4. Updating the file's inode (e.g., size, access time, modification time, …)
- Should the inode be solely updated in memory (steps 3 for reads and 4 for writes)?
  If the disk version is not updated and the system crashes…

Optimizing I/O performance

Caching and buffering

- Opening, reading, and writing to a file requires several disk reads and writes!
  - Given the large amount of I/O to disk (and since disks are slow), most file systems aggressively use RAM
- Early file systems introduced a **fixed-size cache** (e.g., 10% of RAM) to hold popular inodes and data blocks
  (with eviction strategies such as LRU)
  - What if the FS does not need 10% of RAM? We are wasting space with a **static partitioning** approach

Optimizing I/O performance

Unified Page Cache

- Modern FSs employ a **dynamic partitioning** approach that integrates virtual memory pages and file system pages (*i.e.*, file data) into a **unified page cache**
  - The **page cache** space is dynamically allocated according to the needs of the FS and virtual memory components
  - Other dedicated caches exist at the OS level, for example, for inodes, directories, ..
- Updates (writes) to inodes and data blocks can also be **buffered**
  - It gives the opportunity to **batch** updates (or even avoid I/O when files are created and deleted in a short window)
  - **Problem:** If the system fails (e.g., due to a power loss) buffered data is lost! That's why we have the *fsync()* call and the *I/O direct* mode
  - **Buffering** data **improves performance** at the **cost of data durability** (a common tradeoff)

Optimizing I/O performance

Disk-awareness and fragmentation

- Our **vsfs** design is still inefficient
  - It treats the disk as random-access memory, which is not true, especially for HDDs
  - inodes, and corresponding data blocks are usually accessed together. However, in vsfs, these are placed in far away disk regions (bad for seek time)
- Modern file systems are disk-aware
  - Place inodes and data blocks of a file (or files in the same directory) in a **block group** that is aware of the disk layout (e.g., **minimizes seek time**). These designs leverage the **locality** of files' accesses!
- Another important concern is that **free space management** must be done very carefully to avoid **fragmentation** (i.e., free blocks being scattered across the disk, making it difficult to allocate contiguous blocks for files)
  - Such motivated the emergence of **disk defragmentation** tools that would rearrange free and used blocks to make room for contiguous free space

Crash Consistency

Challenges

- System failures are expected to happen (e.g., sudden power losses). Ensuring a **crash-consistent** state of the file system under failures is a complex and hard problem!
  - FSs buffer data and metadata updates to boost performance, leaving an open window for data loss
  - Creating and writing files requires several disk I/O requests to update inodes, bitmaps, and data blocks. In which order should these be done?
- There are multiple crash scenarios where data and metadata may be **lost** or left **inconsistent**!
  - The file's inode was updated, but the data block was never written (garbage pointer to data)
  - A data block was marked as used in the bitmap, but the inode and data were not written (space leak)
  - Data and inode were updated, but the bitmap was not (the block is marked as free and can be overwritten!)
  - Bitmap and data block were updated, but the inode was not (again, space leak)
  - And, unfortunately, there are other crash scenarios…
- Ideally, these 3 operations would be done **atomically**! This is difficult to do as the disk only commits one write at a time, and therefore the **crash-consistency problem**

Crash Consistency

File system checker and journaling

- Use a **file system checker** (e.g., the fsck tool). Let inconsistencies happen and then have a tool checking for these and trying to repair them

- **Journaling:** Use a write-ahead log[1] (journal) to write a transaction (containing pending data and metadata updates) before updating key structures. If a crash occurs, the journal has the necessary steps to try the request again (e.g., ext4)
  - Write (*i.e.*, checkpoint) data and metadata updates to their final locations later
  - Most FSs only register metadata updates in the journal. Data is first written to disk. After confirming that data is persisted, metadata updates are added to the journal
    - With this approach, no garbage pointers can occur, keeping metadata consistent while avoiding writing data updates to the journal (i.e., avoiding extra disk I/O requests)

[1] idea borrowed from the database community.

Crash Consistency

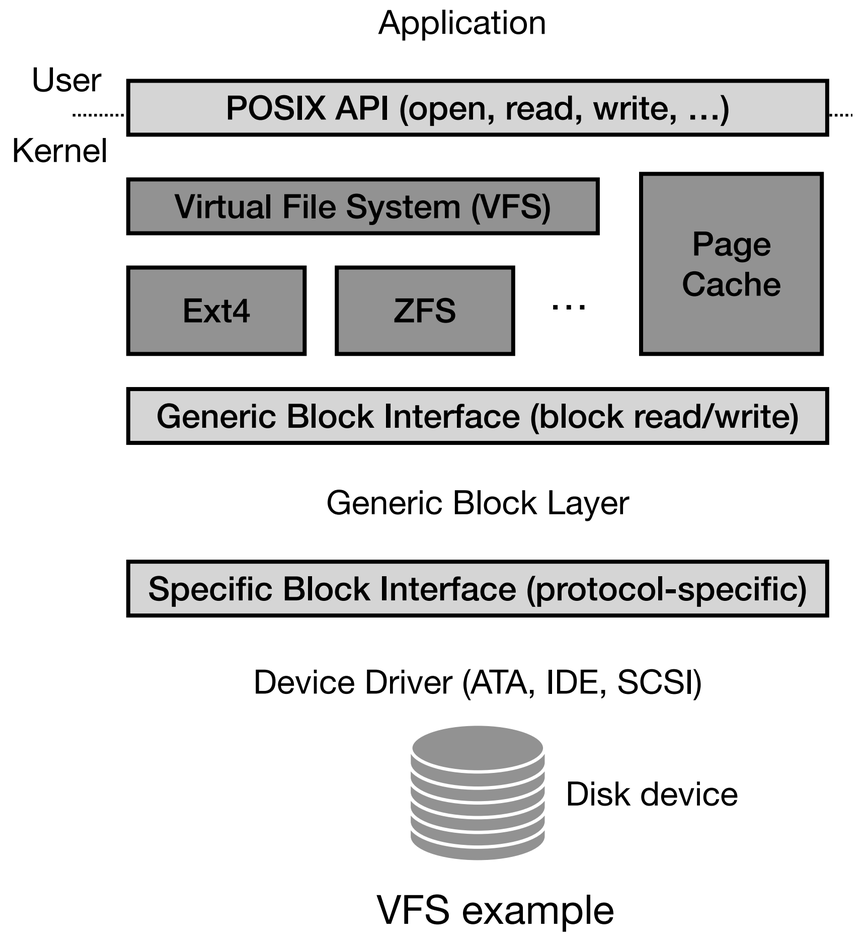
Log-structured file systems

- **Log-structured file systems (LFSs)** avoid overwriting (i.e., in-place updating) data on the disk. They always write to an unused portion of the disk and then later reclaim that old space through cleaning (e.g., ZFS)
  - This approach is known as **copy-on-write** and enables highly efficient writing, as LFSs can buffer data and metadata updates in memory and then write them out together sequentially to disk
  - We are leaving out several challenges from this very, very, very brief overview. For example, how can inodes be searched efficiently, how old data and metadata copies are garbaged collected, how consistent states are recovered under failures …

Virtual File System

Abstractions

- Given the multiple file system implementations available, how can the OS **abstract** them to applications?
- The **Virtual File System** (VFS) is an abstract layer on top of a specific file system implementation
  - It is useful for applications to access **transparently** different local file systems (*e.g.*, ext4, ZFS) and even remote ones (*e.g.*, NFS)
- It specifies an interface with common operations (e.g., mounting, umounting, accessing files, …)
  - Concrete file system implementations must define these “methods”
  - The interaction with OS caches is also facilitated by the VFS (*e.g.*, unified page cache, inodes/directories caches)



VFS example

More Information

- **Chapters 39 and 40** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces.** Arpaci-Dusseau Books, 2018.
- Avi Silberschatz, Peter Baer Galvin, Greg Gagne. **Operating System Concepts (10. ed)**. John Wiley & Sons, 2018.
- Want to know details about **disk-aware designs, crash consistency and log-structured file systems**?
  - **Chapter 41, 42 and 43** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces.** Arpaci-Dusseau Books, 2018.

Questions?