香港中文大學
The Chinese University of Hong Kong

CSCI5550 Advanced File and Storage Systems

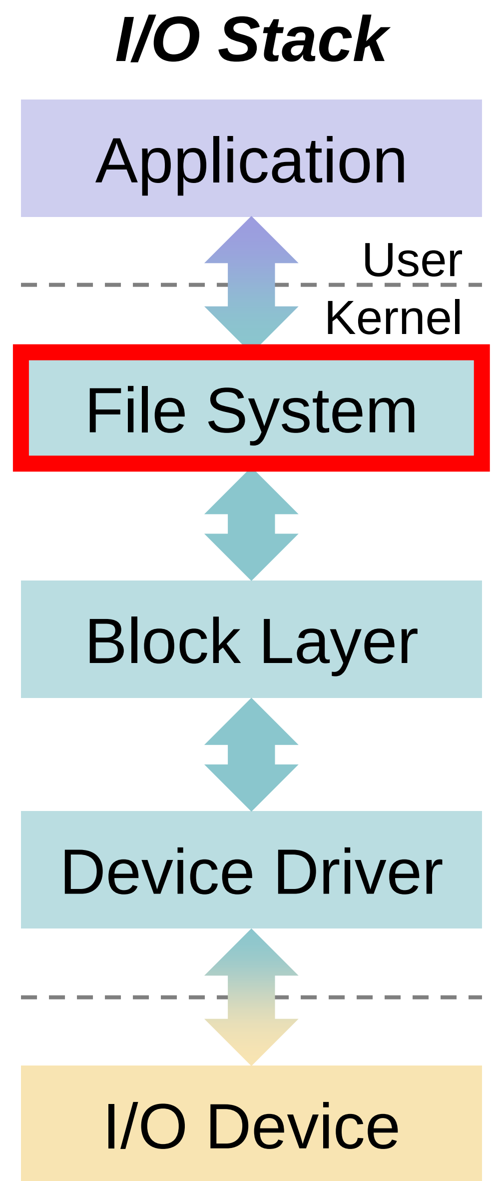
Lecture 03: File System Basics

Ming-Chang YANG

mcyang@cse.cuhk.edu.hk

Outline

- File System Organization
  - Abstraction: Files and Directories
  - Metadata Region and Data Region
- File System Interface
- File System Implementations
  - UNIX File System
    - Access Paths: Reading and Writing
    - Caching and Buffering
  - Fast File System (FFS)
    - Disk Awareness
    - Data Locality
  - Crash Consistency
    - File System Checker
    - Journaling

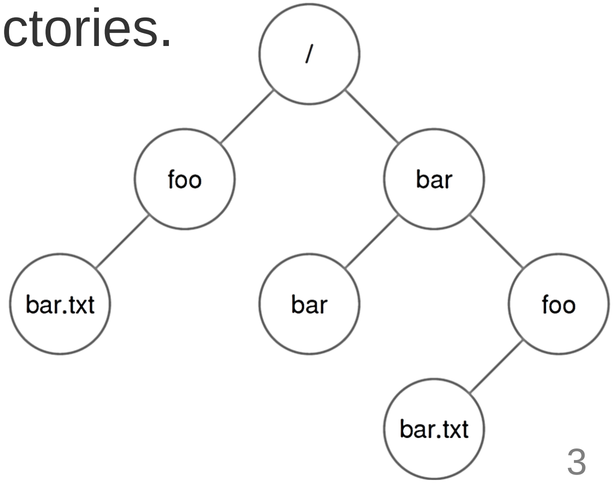


Abstraction: Files and Directories

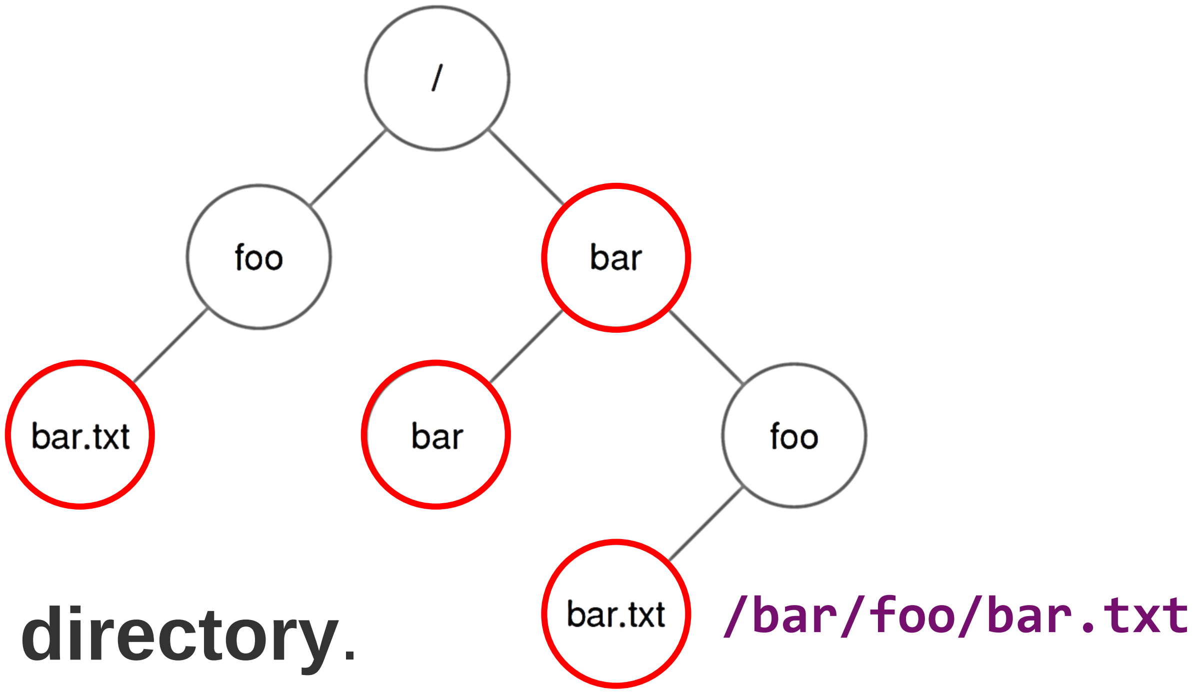
- **File**: a linear array of bytes that can be read/written
  - Each file has a **low-level name** (or **inode number**) that uniquely identifies itself in the file system.
    - Often, the user is not aware of this name.

- **Directory**: a list of entries
  - A directory has an **inode number** as well.
  - A directory is just a **special type of file** with specific content.
    - Each entry is a pair of (user-readable name, inode number).
    - Each entry refers to *either* files *or* other directories.

- A **directory tree** is formed
  - Leaf node: *file*
  - Non-leaf node: *directory*



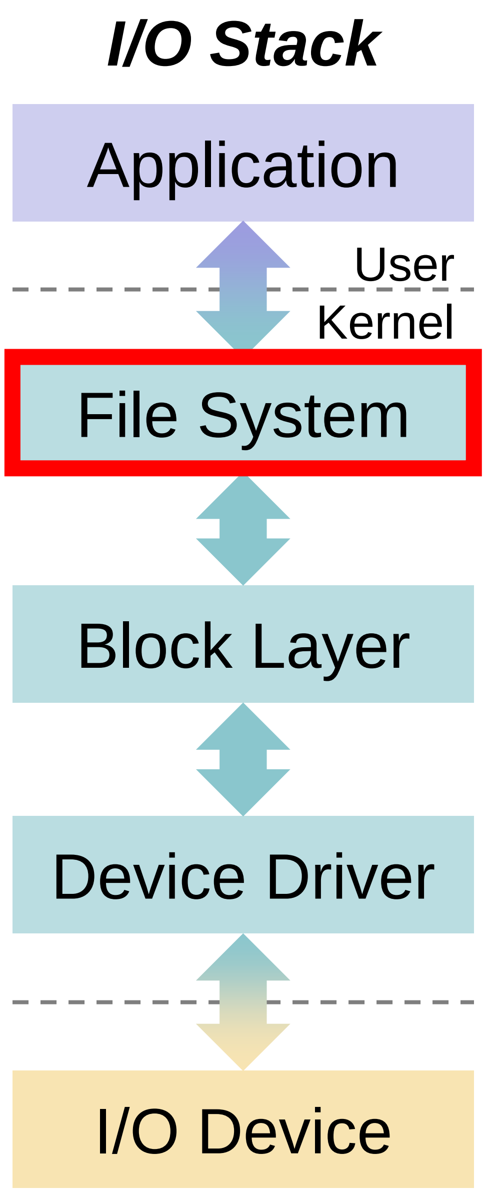
Directory Tree



- **/** is the **root directory**.
  - / is also used as a "**separator**" to name subsequent sub-directories and files.
- A file/directory is referred by the absolute pathname.
  - Directories and files can have the same name.
    - If they are in different locations/directories (e.g., `/bar/foo/bar.txt`).
  - The file extension is to indicate the type of a file (e.g., `.txt`).

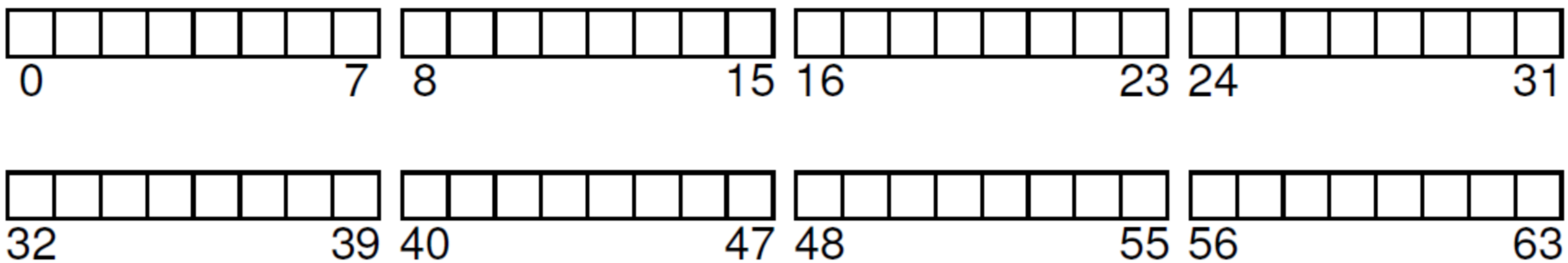
Outline

- File System Organization
  - Abstraction: Files and Directories
  - Metadata Region and Data Region
- File System Interface
- File System Implementations
  - UNIX File System
    - Access Paths: Reading and Writing
    - Caching and Buffering
  - Fast File System (FFS)
    - Disk Awareness
    - Data Locality
  - Crash Consistency
    - File System Checker
    - Journaling

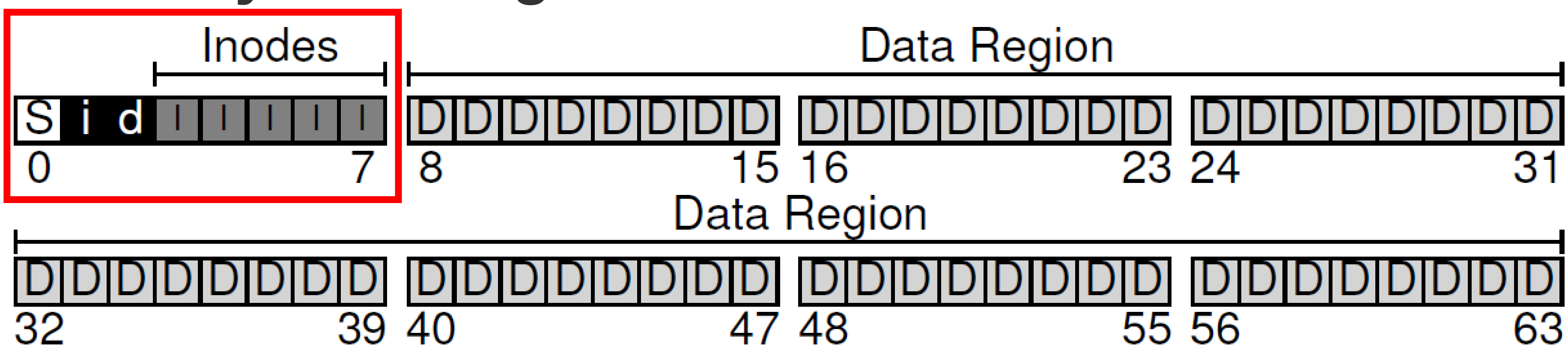


Overall Organization

- **On-Disk Organization**
  - A series of blocks (e.g., 4 KB) is addressed from 0 to N −1.



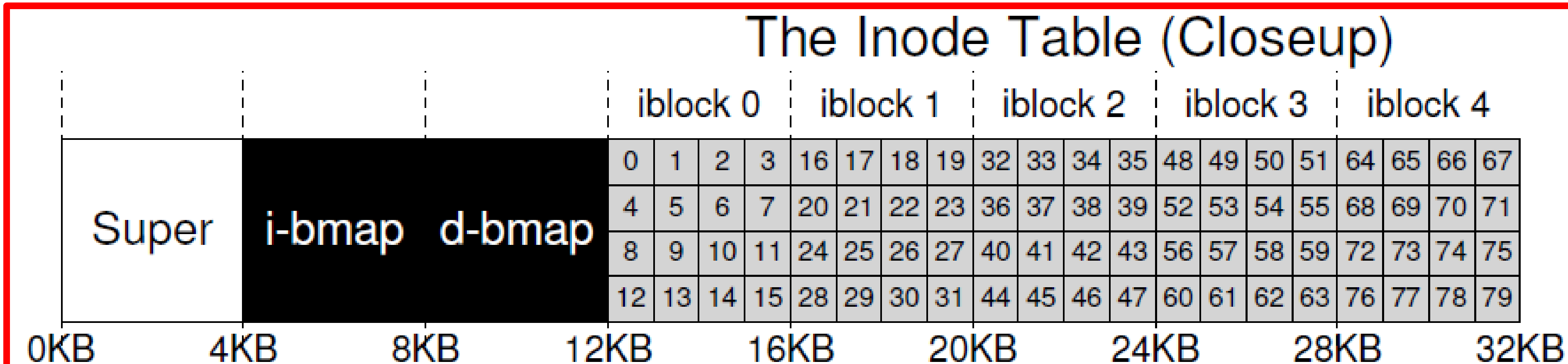
- **File System Organization**



  - **Metadata Region**: tracks data and file system information.
  - **Data Region**: stores user data and occupies most space.

File System Metadata

- **Inode (I)**: tracks "everything" about a **file / directory**.
  - Each inode is referenced by an **i-number** (**low-level name**).
    - Given an i-numbers, the inode can be located.
  - An inode keeps <u>which data block(s) are used for a file / dir</u>.
  - **Inode Table**: the collection of all inodes.
- **i-bmap**: tracks which inode is allocated.
- **d-bitmap**: tracks which data block is allocated.
- **Superblock (S)**: tracks a file system.



Discussion

- Question: How to locate an inode by the i-number in the disk?
  - Note 1: Each inode is small in size.
  - Note 2: A block can hold multiple inodes.
  - Note 3: disk is addressed by sectors.

- **Answer**:
  - Let `inodeStartAddr` be start address of the inode table.
  - Let `sizeof(inode_t)` be the size of a single inoode.

```
blk = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

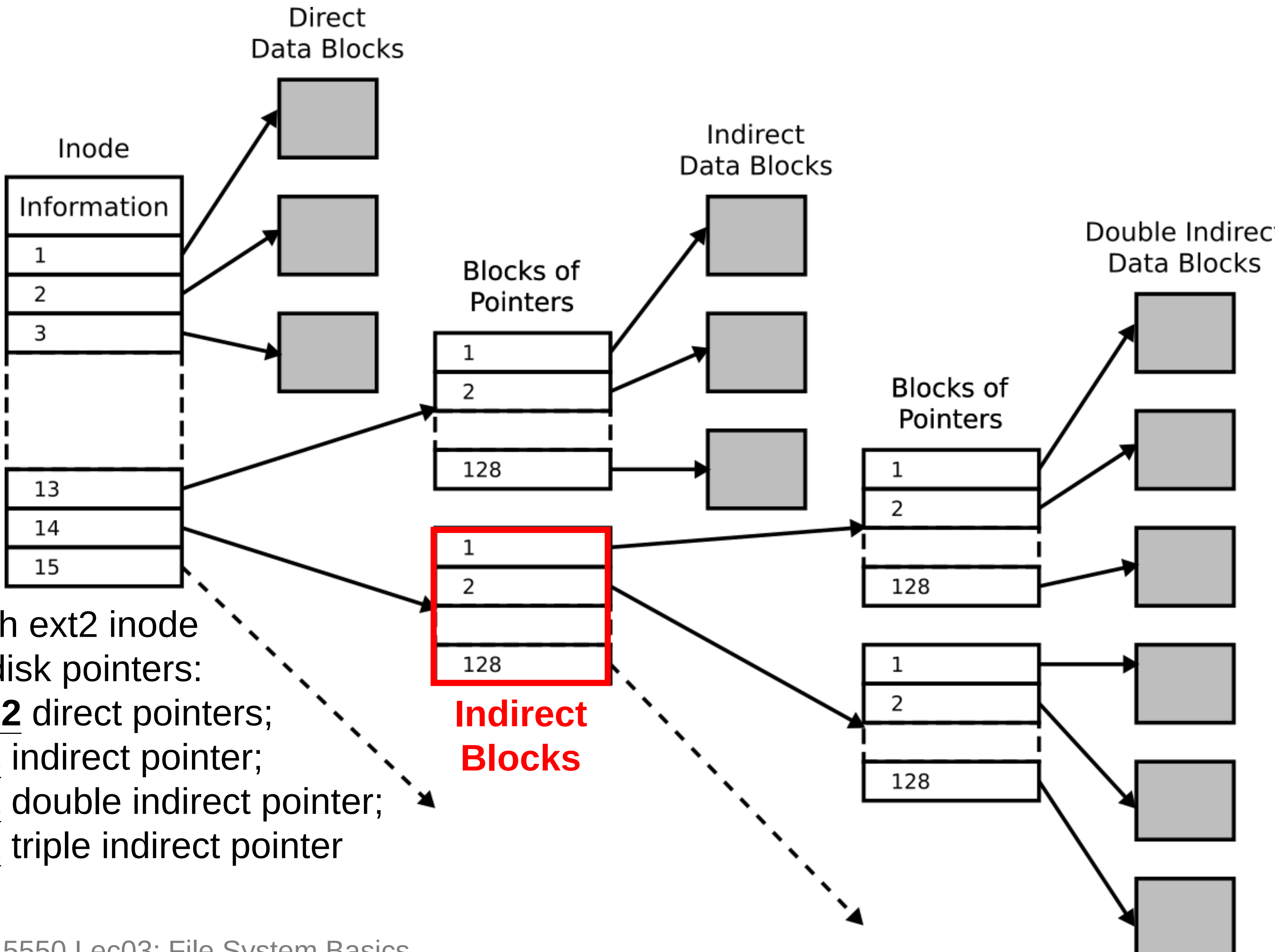
File Organization: Inode (1/3)

- The most important design of the inode:

  How it refers to where data blocks are.

- One simple approach would be to have one or more **direct pointers** (each refers to one data block).
  - Challenge: Hard to support files of big sizes.
- **Multi-Level Index**
  - **Direct Pointer**: points to a data block explicitly.
  - **Indirect Pointer**: points to an indirect block that holds (multiple) pointers to data blocks.
  - **Double Indirect Pointer**: points to pointers to indirect blocks.
  - **Triple Indirect Pointer**: points to pointers to pointers to indirect blocks.

File Organization: Inode (2/3)



Each ext2 inode **15** disk pointers:

- **12** direct pointers;
- **1** indirect pointer;
- **1** double indirect pointer;
- **1** triple indirect pointer

Discussion

- Question: Why we maintain an imbalance tree?

- **Answer**:
  - Most files are small in practice.
  - Access performance is optimized for small files.

- **Bonus**: How big can a file be in Ext2?
  - Let the block size be 4KB;
  - Let each pointer size be 4 bytes.
  - Note: Each inode in Ext2 has 12 direct pointers; 1 indirect pointer; 1 double indirect pointer; 1 triple indirect pointer.

File Organization: Inode (3/3)

- An inode tracks everything except “file name” (why?).

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 file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Directory Organization

- A **directory** is a special type of file.
  - Each directory is also associated with an inode number.
  - A directory contains a list of **(`file name`, `inode number`)** pairs in its corresponding data block(s).

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_lon

`.` = current directory

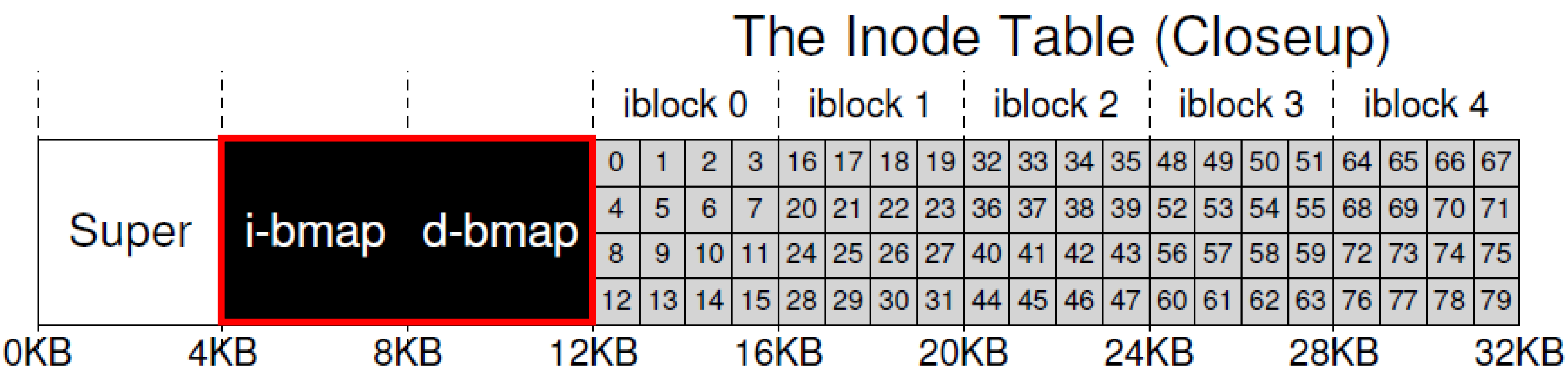
`..` = parent directory

`strlen` = length of the file name (including '\0')

`reclen` = actual space for an entry (used when deletion)

Free Space Management

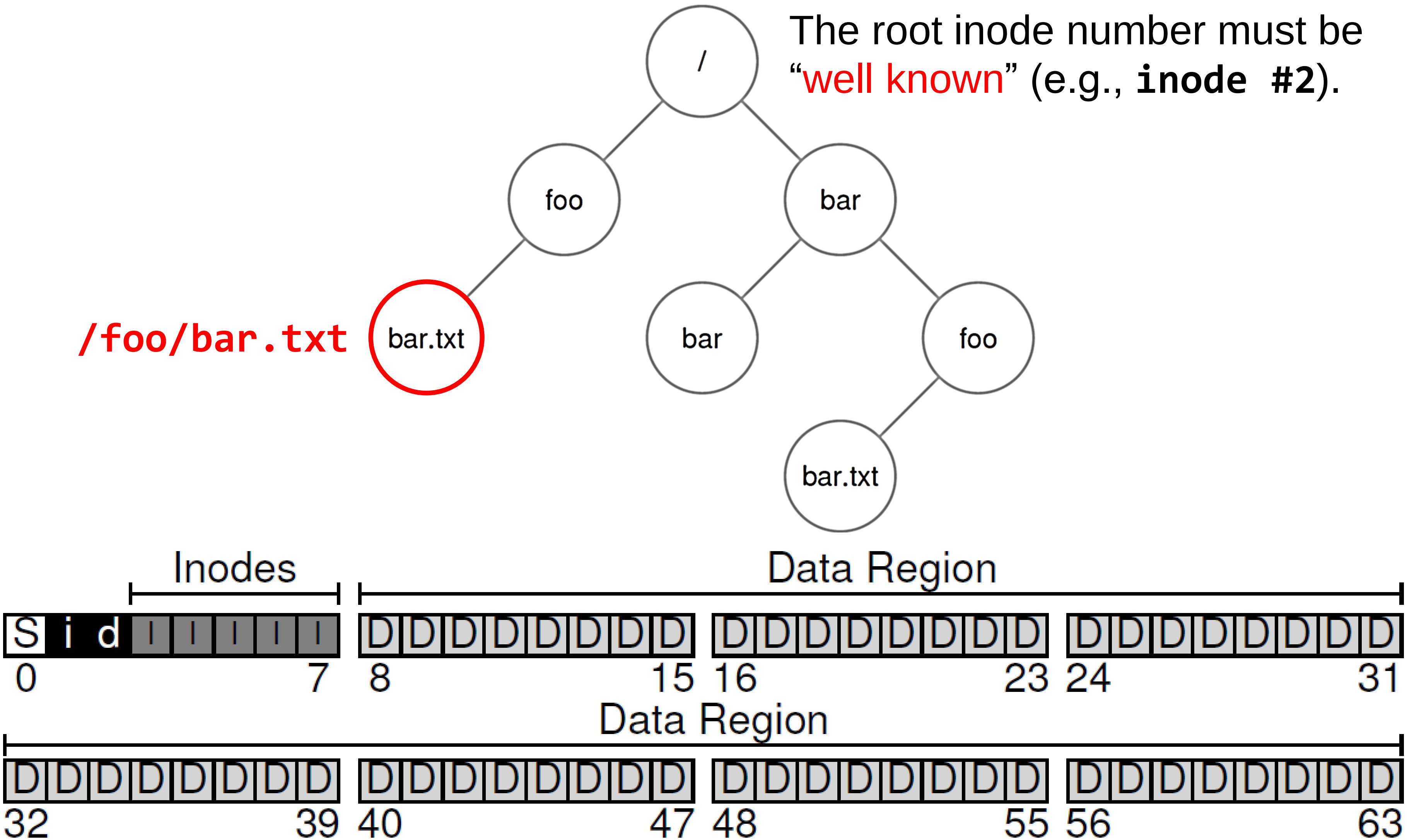
- A file system must track which inodes and data blocks are allocated or not:
  - **Bitmap** is one way for free space management.
    - 0: free; 1: used
  - Other structures, e.g., **free list** and **B-tree**, are feasible.
    - There is always a trade-off between time and space.
  - **Pre-allocation** may also be used.
    - Strategy: Always looking for a sequence of free blocks (say 8).
      - A portion of the file will be contiguous on the disk (better performance).



Exercise

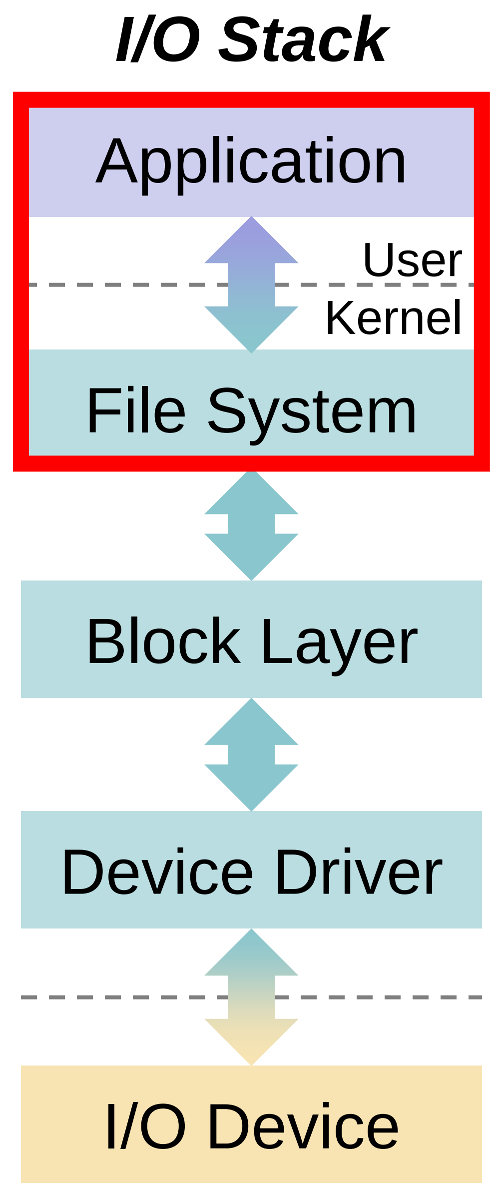
- Question: Can you locate the data block(s) for a file?

The root inode number must be "well known" (e.g., **`inode #2`**).



Outline

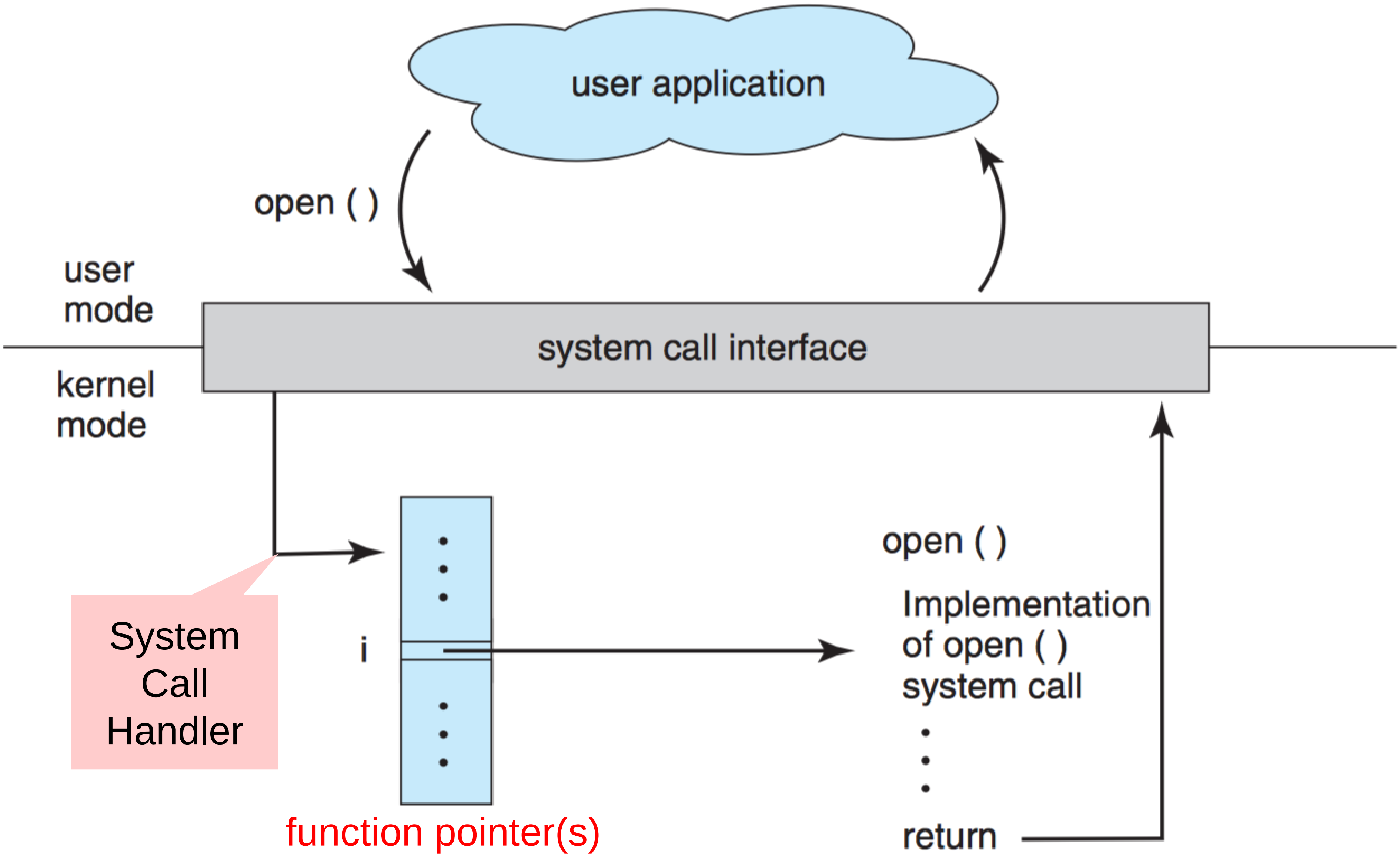
- File System Organization
  - Abstraction: Files and Directories
  - Metadata Region and Data Region
- File System Interface
- File System Implementations
  - UNIX File System
    - Access Paths: Reading and Writing
    - Caching and Buffering
  - Fast File System (FFS)
    - Disk Awareness
    - Data Locality
  - Crash Consistency
    - File System Checker
    - Journaling



File System Interface

- File system interface includes:
  - Creating files;
  - Reading/writing files;
  - Renaming files;
  - Getting information about files;
  - Removing files;
  - Managing directories;
  - Linking files/directories;
  - Mounting/unmounting a file system.
- The file system interface uses (or wraps) the OS **system calls** for file/directory management.
  - We focus on UNIX.

System Calls



Creating Files (1/2)

- The system call **`open()`** is to create or open a file:

```
int fd = open( "foo", O_CREAT | O_WRONLY | O_TRUNC,
                   S_IRUSR|S_IWUSR);
```

  - 1st argument: file name (absolute or relative pathname)
  - 2nd argument:
    - `O_CREAT`: creates a file;
    - `O_WRONLY`: only write is allowed;
    - `O_TRUNC`: truncate to zero size if a file exists.
  - 3rd argument: specifies permissions (readable or writable).

- On success, a **file descriptor** is returned
  - A pointer for subsequent accesses (function calls) to a file.
  - In UNIX, it's just an integer.

Creating Files (2/2)

- **File descriptors** are managed on a per-process basis by the operating system .
- For example, the UNIX systems (xv6 kernel) must keep some kind of structure in the `struct proc`:

```
struct proc {
    ...
    struct file *ofile[NOFILE]; // Open files
    ...
};
```

  – A simple array (with a maximum of `NOFILE` open files) tracks which files are opened on a per-process basis.
  – Each entry of the array is just a **pointer** to a `struct file`, which tracks the information of the “**open file**” being used.

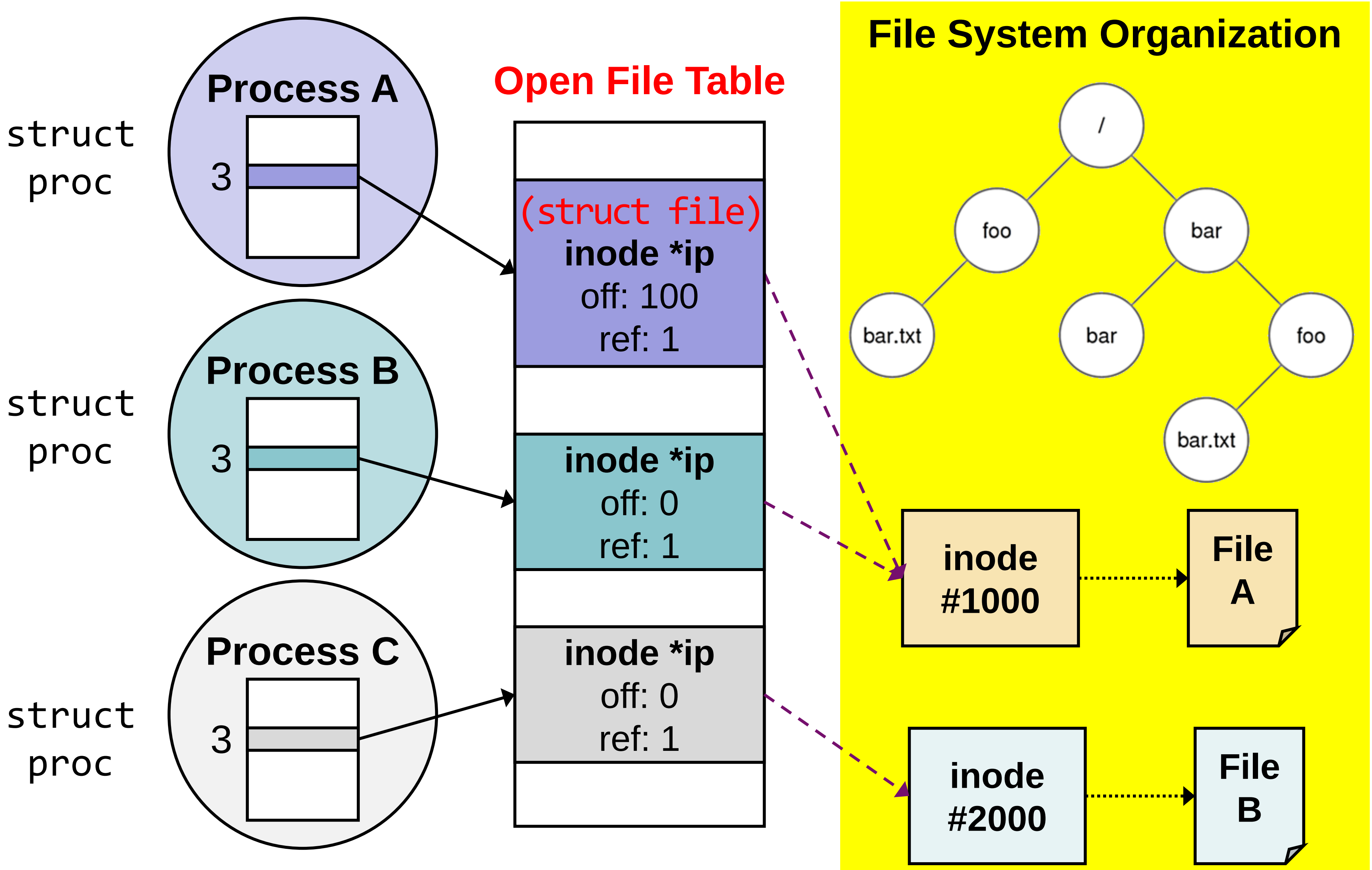
Management of Open Files (1/2)

```
struct proc {
    ...
    struct file *ofile[N];
    // open files
    ...
};
```

```
struct file {
    struct inode *ip;
    char readable;
    char writable;
    uint off;
    int ref;
};
```

- The `struct file` represents an **open file**:
  - The `struct file` (an open file) is referenced by a process.
  - The `readable`/`writable` specifies read/write permissions.
  - The `off` keeps the "current" offset, where the next read/write should take place, for this open file.
  - The actual file is referenced by the `struct inode`.
- All open files are kept in an **open file table** by OS.

Management of Open Files (2/2)



Reading and Writing Files (1/4)

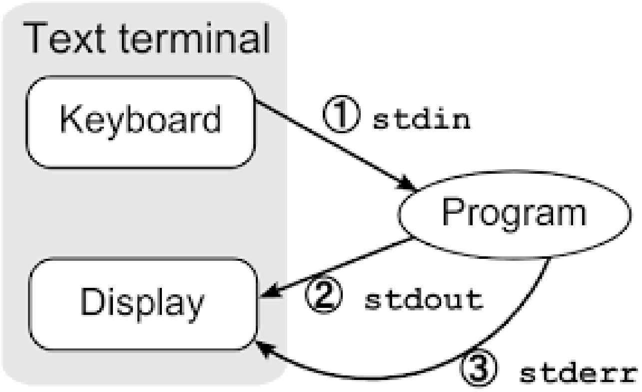
- How does a process actually read or write a file?
- Exercise: Let's use the **`strace`** tool to trace every system call made by reading (`cat`) the file `foo`:

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

Reading and Writing Files (2/4)

```
open("foo", O_RDONLY|O_LARGEFILE) = 3
```

- First, the **`open()`** system call opens a file for reading:
  - `O_RDONLY`: read only (writing is not allowed)
  - `O_LARGEFILE`: 64-bit offset is used.
- `open()` **returns** a file descriptor of **3**.
  - Each running process already has three “open files”:
    - Standard Input: 0
    - Standard Output: 1
    - Standard Error: 2



Reading and Writing Files (3/4)

```
read(3, "hello\n", 4096) = 6
write(1, "hello\n", 6)   = 6
hello
read(3, "", 4096)        = 0
close(3)                 = 0
```

- `read()` are called for **many times** to read the file:
  - 1st argument: file descriptor
  - 2nd argument: buffer where the results are stored
  - 3rd argument: size of the buffer
- `write()` is called to display output on screen (`fd=1`).
- `close()` is called when reaching the `EOF`.
- Writing a file? `open()` → `write()` → `close()`

Reading and Writing (4/4)

- `lseek()` is to read or write to a specific offset within a file (*rather than from the beginning to the end*).

```
off_t lseek(int fd, off_t offset, int whence);
```

  – 1st argument: file descriptor
  – 2nd argument: positions the `offset` to a particular location within a file (for subsequent reads/writes).
    - `lseek()` has nothing to do with a disk seek!
  – 3rd argument: specifies how `lseek()` is performed.
    - `SEEK_SET`: set to offset bytes from the **beginning**
    - `SEEK_CUR`: set to **current location** plus offset bytes
    - `SEEK_END`: set to offset bytes from the **end**

Exercise

- Let's track a process that
  - opens a file named "file" (of size 300 bytes);
  - reads it by calling the read() system call repeatedly (each time reading 100 bytes).

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`		
`read(fd, buffer, 100);`		
`read(fd, buffer, 100);`		
`read(fd, buffer, 100);`		
`read(fd, buffer, 100);`		
`close(fd);`		

Exercise

- Let’s track a process that
  - uses `lseek()` to reposition the current offset;
  - reads 50 bytes from the file;
  - closes the file.

System Calls	Return Code	Current Offset
`fd = open("file", O RDONLY);`		
`lseek(fd, 200, SEEK_SET);`		
`read(fd, buffer, 50);`		
`close(fd);`		

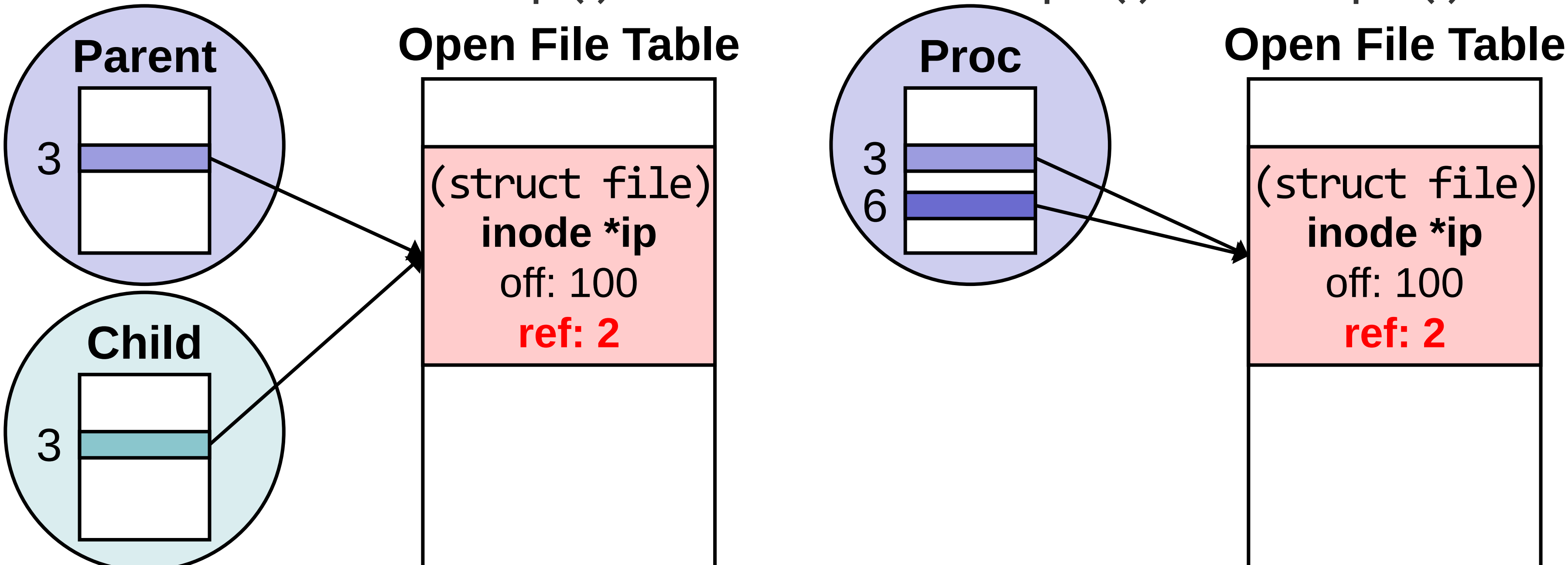
Exercise

- Let's track a process that
  - opens the same file (named "file") twice;
  - issues a read to each of them.

System Calls	Return Code	Current Offset (fd1)	Current Offset (fd2)
`fd1 = open("file", O RDONLY);`			
`fd2 = open("file", O RDONLY);`			
`read(fd1, buffer1, 100);`			
`read(fd2, buffer2, 100);`			
`close(fd1);`			
`close(fd2);`			

Shared File Table Entries

- In many cases, the mapping of file descriptor to an entry in the open file table is a **one-to-one** mapping.
- An entry in the open file table can be *shared* when
  - A parent process creates a child process with `fork()`;
  - A process creates a few file descriptors that refers to the same file with `dup()` or its cousins `dup2()` and `dup3()`.



Forcing Writes

- For performance, the file system **buffers writes** in memory (e.g., for 5 sec or 30 sec).

- The `fsync()` system call forces all dirty (i.e., not yet written) data to the disk.

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
              S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

Information of Files

- The file system keeps a fair amount of information about each file it is storing.
  - `stat()` or `fstat()` calls can be used to see the metadata.

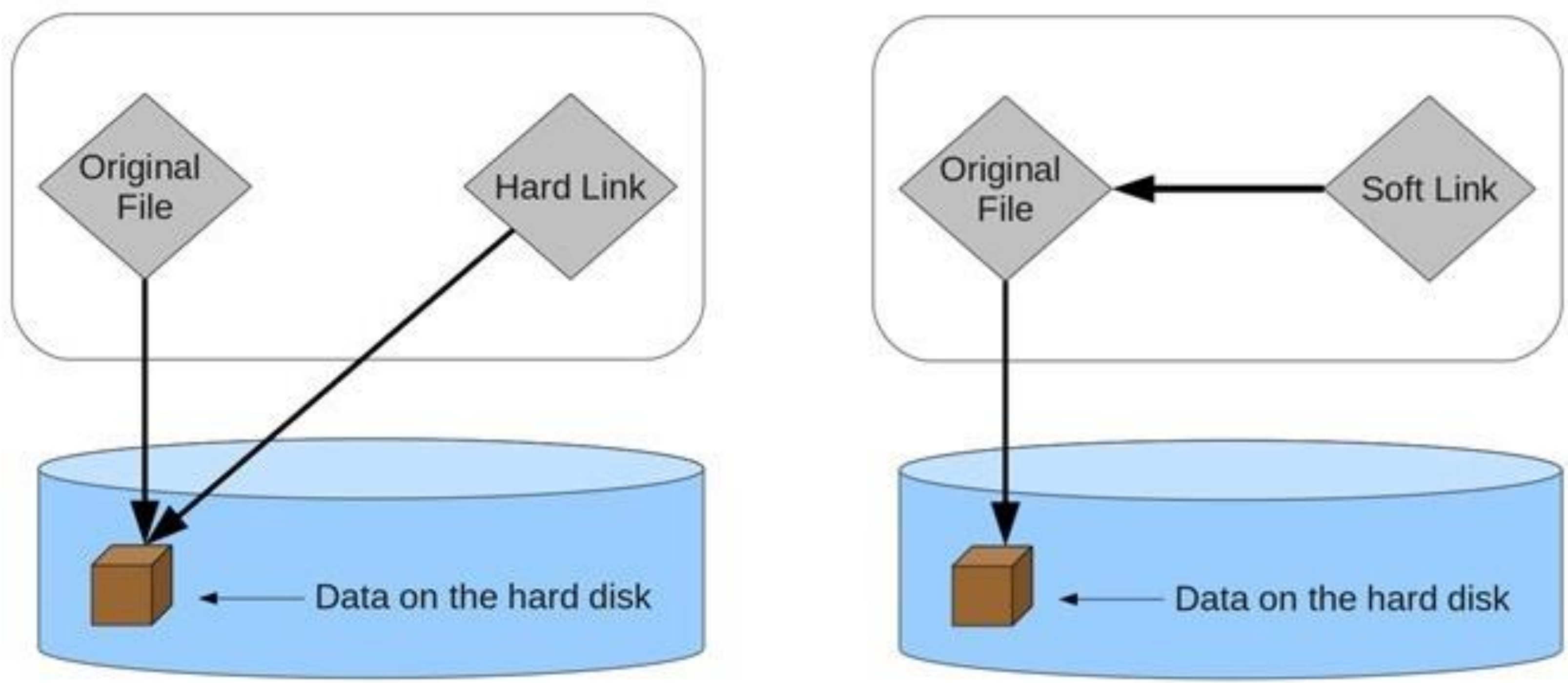
```
struct stat {
    dev_t st_dev; /* ID of device containing file */
    ino_t st_ino; /* inode number */
    mode_t st_mode; /* protection */
    nlink_t st_nlink; /* number of hard links */
    uid_t st_uid; /* user ID of owner */
    gid_t st_gid; /* group ID of owner */
    dev_t st_rdev; /* device 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 */ };
```

Summary of File System Operations

File System Operations	System Calls
Creating a file	`open()`
Reading a file	`read()`
Writing a file	`write()`, `fsync()`
Seeking to an offset	`lseek()`
Renaming a file	`rename()` (often an **atomic** call)
Getting file information	`stat()` or `fstat()`
Removing a file	`unlink()`
Making a directory	`mkdir()`
Reading a directory	`opendir()`, `readdir()`, `closedir()`
Removing a directing	`rmdir()` (must be **empty**)

Links

- File systems allow **links** to create multiple names (aliases) for the same file
  – **Hard Link**: holds the inode number of a file.
  – **Symbolic/Soft Link**: holds the pathname to a file.



Recall: Directory Organization

- A **directory** is a special type of file.
  - Each directory is also associated with an inode number.
  - A directory contains a list of **(file name, inode number)** pairs in its corresponding data block(s).

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_lon

`.` = current directory

`..` = parent directory

`strlen` = length of the file name (including '\0')

`reclen` = actual space for an entry (used when deletion)

Hard Link

- **Hard link** (`ln`) creates **a new entry** in the directory to refer to the **same inode number** of the original file.

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file hard_link
prompt> cat hard_link
hello
```

Create a Hard Link

```
prompt>
ls -i file hard_link
67158084 file
67158084 hard_link
```

Show inode Numbers

```
prompt> rm file
removed 'file'
prompt> cat hard_link
hello
```

Remove a File

- The inode has a **reference count** that keeps track of how many hard links refer to it.
  - Only when the reference count is zero, the file system frees the inode and related data blocks.
  - This explains why `unlink()` is called when removing a file.

Recall: File Organization: Inode

- An inode tracks everything except "file name" (why?).

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 file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

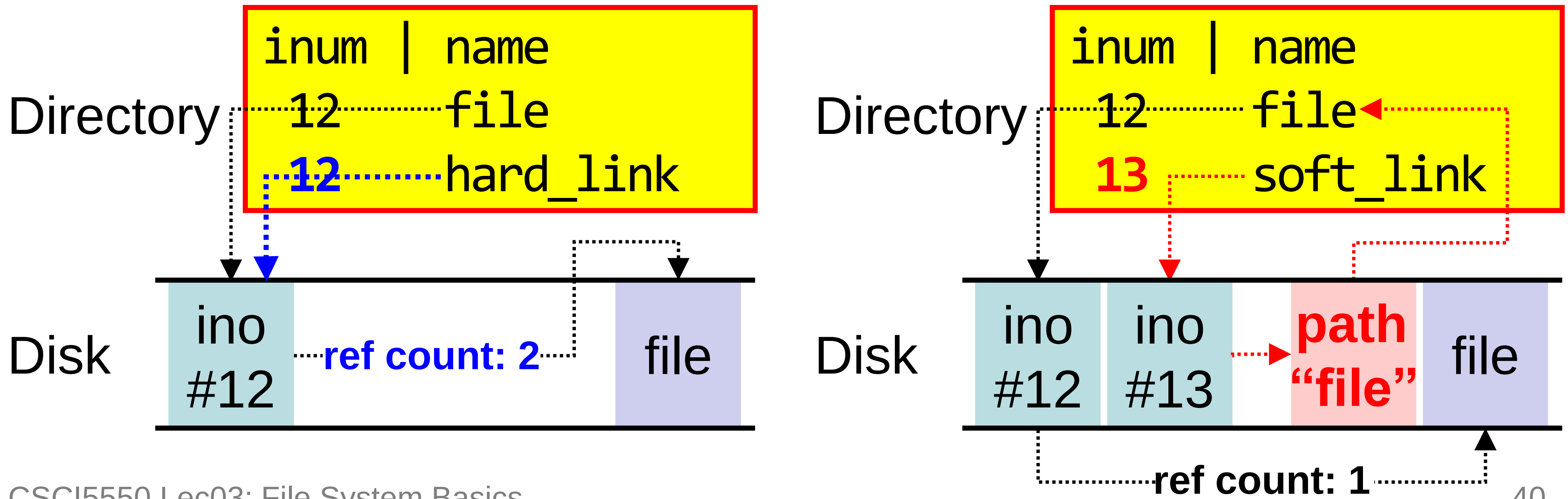
Symbolic/Soft Link

- Hard links are limited:
  - Cannot link to a directory (to avoid creating a cycle).
  - Cannot link to a different partition (only within a file system).
- **Symbolic Link** (`ln -s`)
  - **It is a special file** with its own inode number.
  - It holds a pointer to link but may cause dangling reference.

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln -s file soft_link
prompt> cat soft_link
hello
prompt> rm file
prompt> cat soft_link
cat: soft_link: No such file or directory
```

Hard Link vs. Soft Link

- File systems allow **links** to create multiple names (aliases) for the same file
  - **Hard Link**: holds the inode number of a file
    - By only creating a **new directory entry**.
  - **Symbolic/Soft Link**: holds the pathname to a file
    - By creating a **new file of special type**.
    - Three types of file: 1) Data File; 2) Directory File; 3) Soft Link File.



Mounting a File System

- Final Step: Set up a file system to make it run.
- **Mounting** (`mount`) a file system:
  - – Create a mount point
  - – Paste a file system onto the directory tree at that point

your I/O device (SCSI)

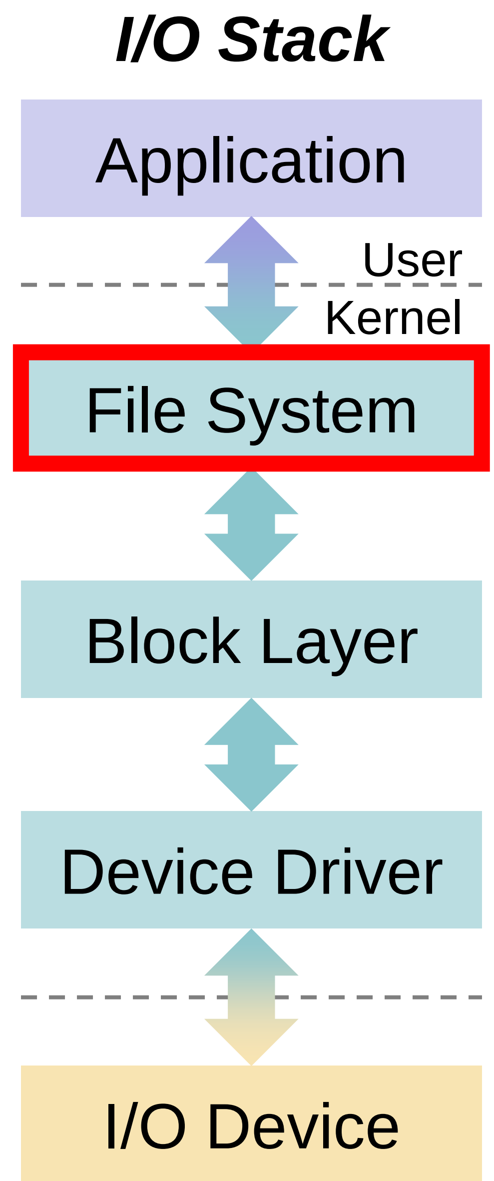
```
prompt> mount -t ext3 /dev/sda1 /home/users
```

- You can have multiple file systems on the same machine, and mounts all file systems into one tree!

```
/home/users/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

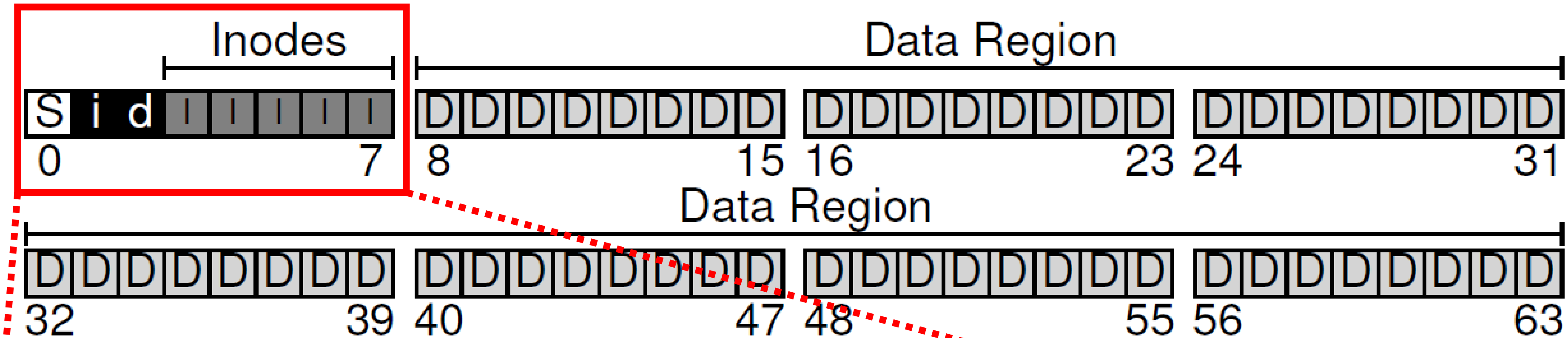
Outline

- File System Organization
  - Abstraction: Files and Directories
  - Metadata Region and Data Region
- File System Interface
- File System Implementations
  - UNIX File System
    - Access Paths: Reading and Writing
    - Caching and Buffering
  - Fast File System (FFS)
    - Disk Awareness
    - Data Locality
  - Crash Consistency
    - File System Checker
    - Journaling

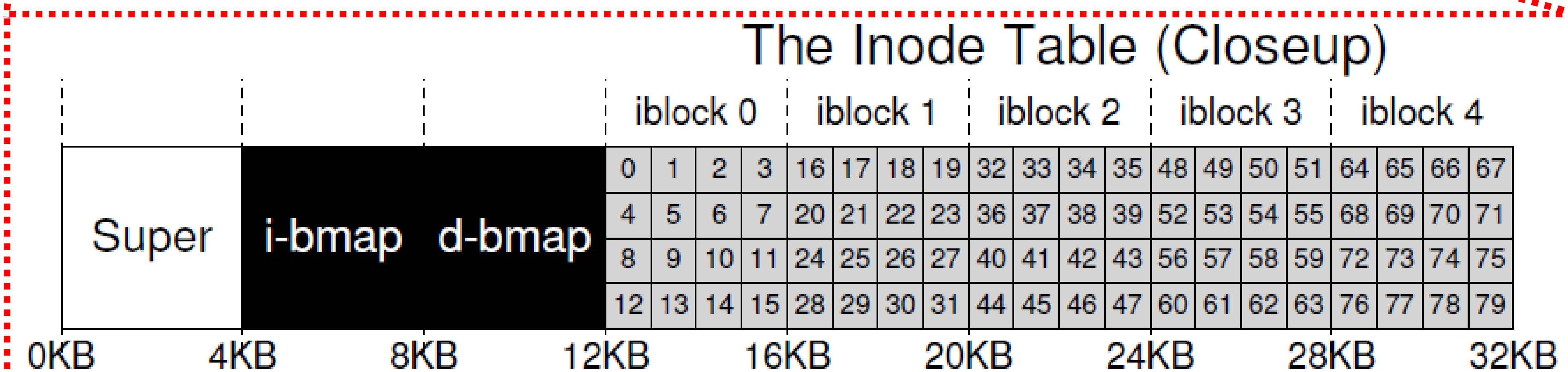


UNIX File System

- The organization we have learnt is a simplified version of a typical **UNIX file system**:

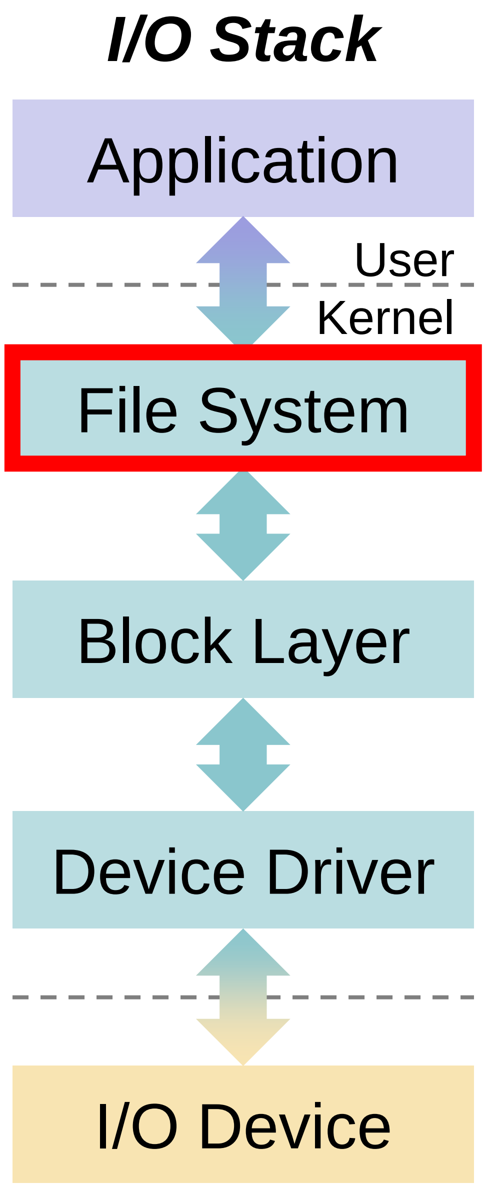


  - **Metadata Region**: tracks data and file system information.
  - **Data Region**: stores user data and occupies most space.



Outline

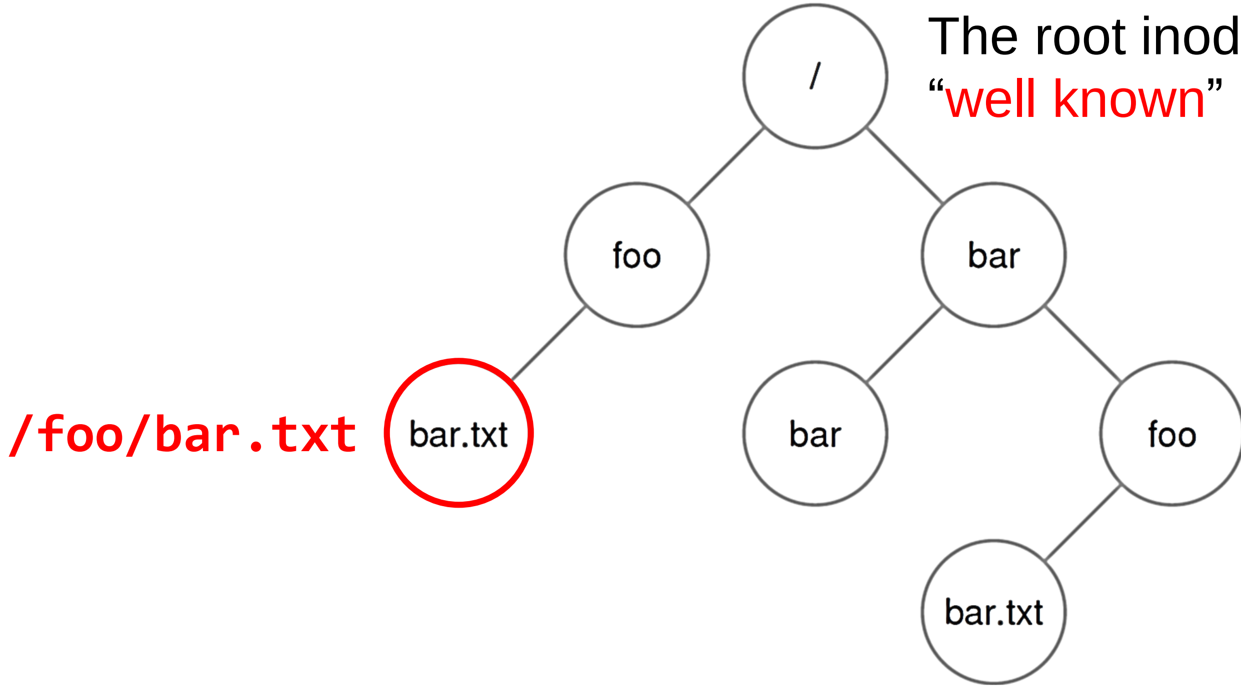
- File System Organization
  - Abstraction: Files and Directories
  - Metadata Region and Data Region
- File System Interface
- File System Implementations
  - UNIX File System
    - Access Paths: Reading and Writing
    - Caching and Buffering
  - Fast File System (FFS)
    - Disk Awareness
    - Data Locality
  - Crash Consistency
    - File System Checker
    - Journaling

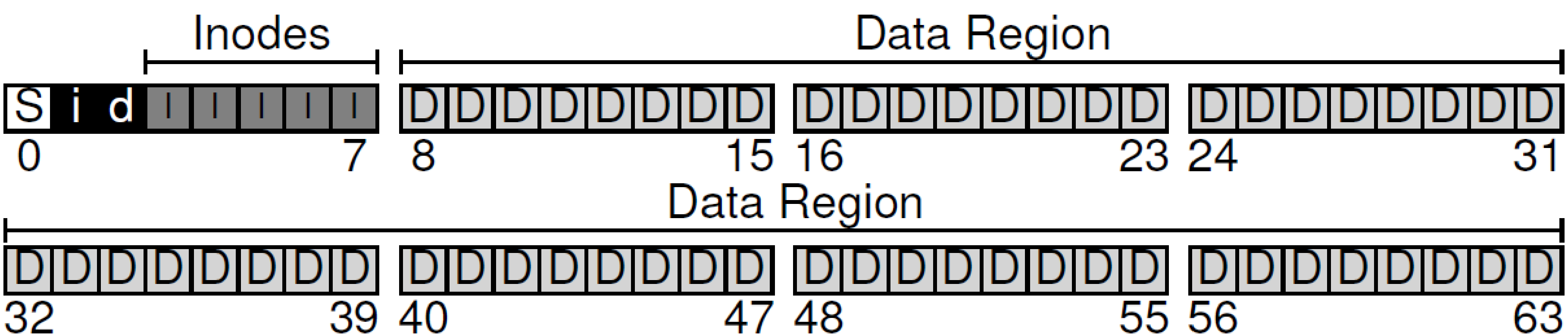


Recall: Exercise

- Question: Can you locate the data block(s) for a file?

The root inode number must be "well known" (e.g., **`inode #2`**).





Access Path: Read (1/2)

- Example: Read a file `/foo/bar`
  - **Traverse the pathname** to locate the requested inode:
    - `root` → `foo` → `bar`

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read	read	read	read	read			
read()					read write			read		
read()					read write				read	
read()					read write					read

1. Read `root` inode (**must be known**) to locate `root` data
2. Read `root` data to find `foo` inode
3. Read `foo` inode to locate `foo` data
4. Read `foo` data to find `bar` inode
5. Read `bar` inode into memory*

6. Read `bar` inode to locate data
7. Read data block of `bar`
8. Update timestamp of `bar` inode

Access Path: Read (2/2)

open(bar)		read 1. Read root inode (**must be known**) to locate root data read 2. Read root data to find foo inode read 3. Read foo inode to locate foo data 4. Read foo data to find bar inode read read 5. Read bar inode into memory*

- **Note 1**: The amount of I/O generated by the open() is proportional to the length of the pathname.
  - Large directories would make this worse. (Why? Step 4)
- **Note 2**: The following work is also needed but not listed:
  - **Step 5** also needs to check permissions; allocate a file descriptor for this process; create an entry in the open-file table; return the file descriptor to the user.

6. Read bar inode to locate data

- **Note 3**: The read will further update the in-memory open file table to maintain the file offset for this file descriptor.
  - Such that the next read will read the subsequent file block.

Access Path: Write (1/2)

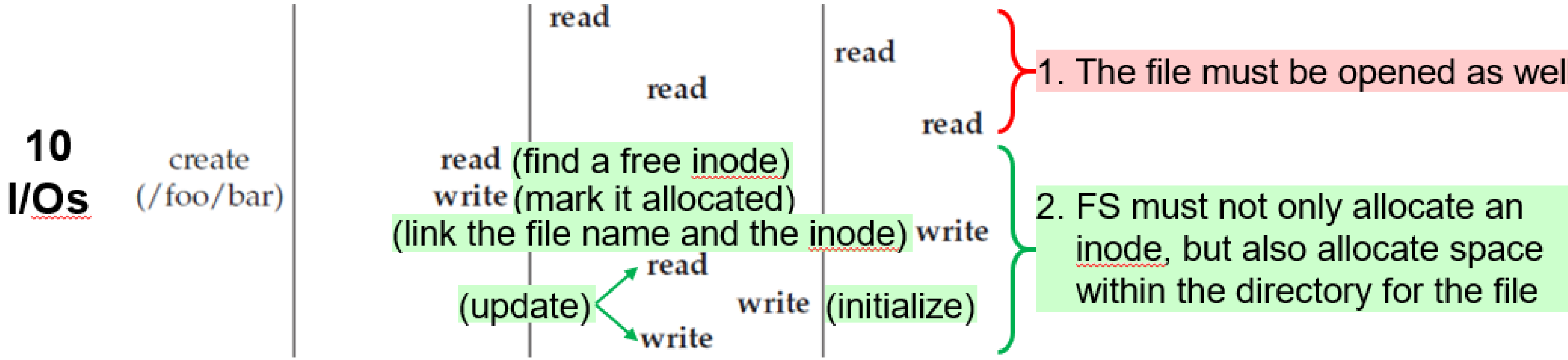
- Example: Create (write) a new file `/foo/bar`

I/Os		data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
10 I/Os	create (/foo/bar)			read							
							read				
					read						
								read			
			read (find a free inode)								
			write (mark it allocated)								
								write (link the file name and the inode in dir)			
					read (update)						
						write (initialize)					
					write (update)						
5 I/Os	write()					read					
		read (find a free data block)									
		write (update data bitmap)									
									write (write the actual data block)		
						write (update with the new data block location)					
5 I/Os	write()					read					
		read									
		write									
										write	
						write					
5 I/Os	write()					read					
		read									
		write									
											write
						write					

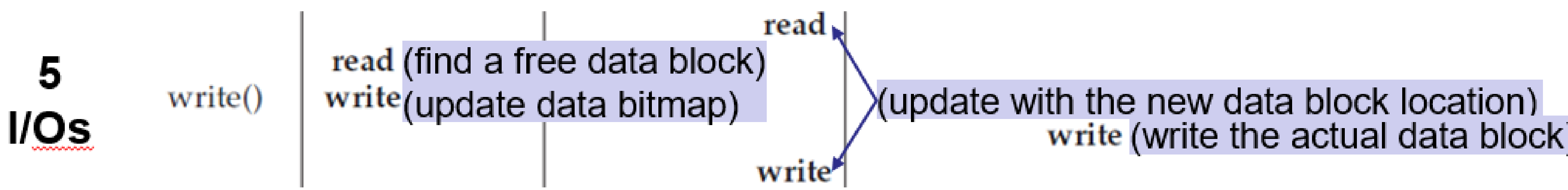
1. The file must be opened as well

2. FS must not only allocate an inode, but also allocate space within the directory for the file

Access Path: Write (2/2)



– **10 I/Os** are needed to walk the pathname and create file.
  - If the directory needs to grow, additional I/Os are needed.
    – i.e., to the data bitmap, and the new directory block.

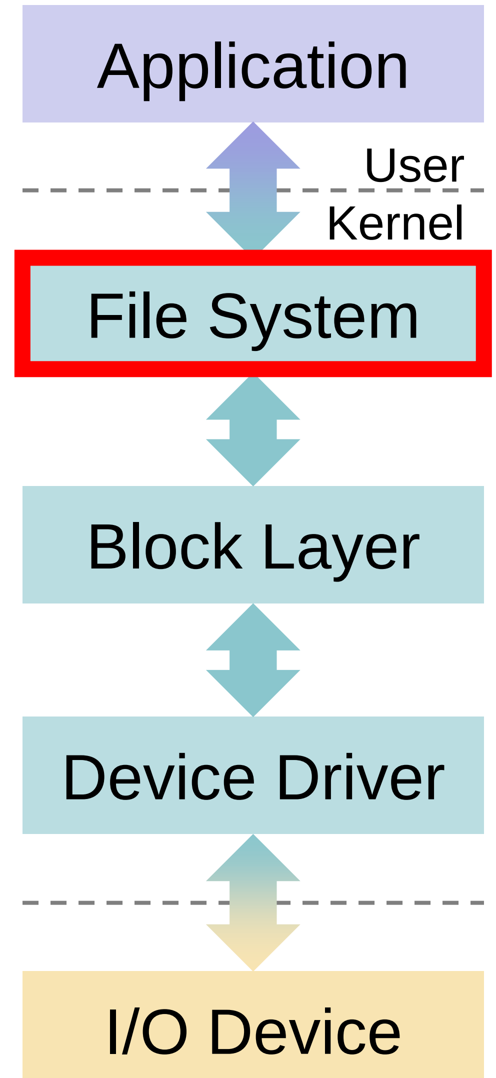


– Each data block write logically generates **5 I/Os**.
  - If `write()` involves indirect pointers, more I/Os are needed as well.

Outline

- File System Organization
  - Abstraction: Files and Directories
  - Metadata Region and Data Region
- File System Interface
- File System Implementations
  - UNIX File System
    - Access Paths: Reading and Writing
    - Caching and Buffering
  - Fast File System (FFS)
    - Disk Awareness
    - Data Locality
  - Crash Consistency
    - File System Checker
    - Journaling

I/O Stack

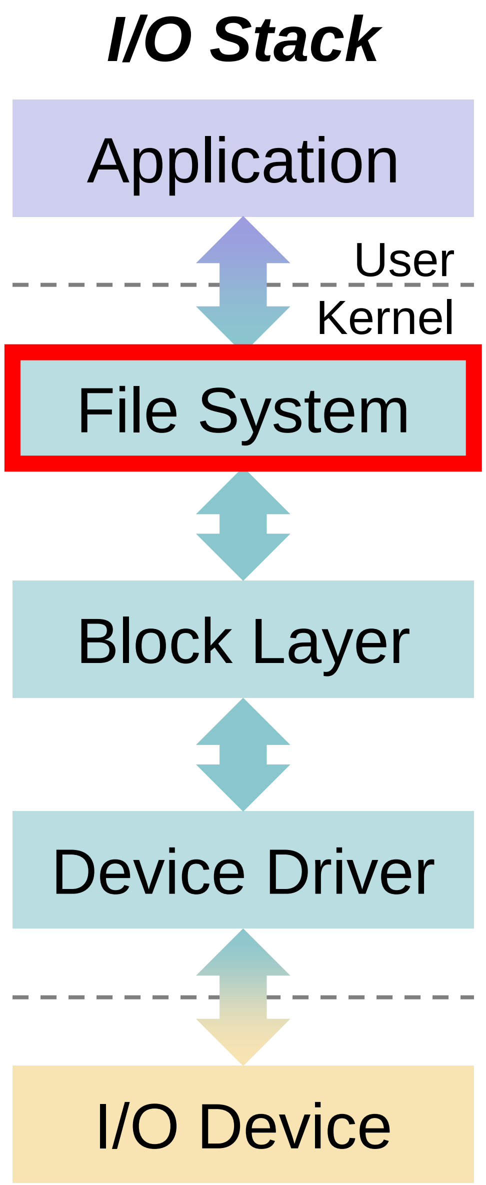


Caching and Buffering

- Like in UNIX file system, reading and writing files can be expensive, incurring many I/Os to the (slow) disk.
- Most file systems leverage the system memory to:
  - **Cache** some important or popular blocks
    - To avoid repeated reads to the same blocks
    - To avoid performing hundreds of reads to open a file with long pathname (e.g., `/1/2/3/…/100/file.txt`).
  - **Buffer** a number of writes (for 5~30 seconds)
    - To allow writes to the same location (in memory)
    - To batch updates into a smaller set of I/Os
    - To allow rescheduling of I/Os
  - **Cache/buffer trades reliability for performance!**
    - But **not** everyone likes it; some applications (e.g., databases) require frequent `fsync()` to avoid losing data kept in the write buffer.

Outline

- File System Organization
  - Abstraction: Files and Directories
  - Metadata Region and Data Region
- File System Interface
- File System Implementations
  - UNIX File System
    - Access Paths: Reading and Writing
    - Caching and Buffering
  - Fast File System (FFS)
    - Disk Awareness
    - Data Locality
  - Crash Consistency
    - File System Checker
    - Journaling



Revisit File System Organization (1/2)

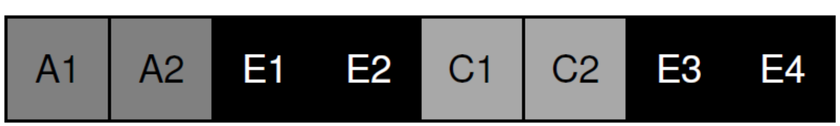
- "Old UNIX File System" by Ken Thompson:

  - The **super block (S)** contained the file system information:
    - How big the volume is, how many inodes there are, a pointer to the head of a free list of blocks, and so forth.
  - The inode region contained all **inodes** for the file system.
  - Most of the disk space was taken up by **data blocks**.
- **Problem 1: Poor Performance**
  - The file system was delivering only 2% of disk bandwidth, because of expensive disk positioning costs.
    - The data blocks of a file were often very far away from its inode.
      - An expensive seek was induced whenever one first read the inode, and then read the file system block.

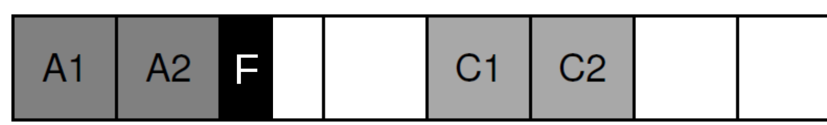
Revisit File System Organization (2/2)

- "Old UNIX File System" by Ken Thompson:

- **Problem 2: Fragmentation**
  - **External Fragmentation**
    - Free block space is not contiguous.
    - A large file may have blocks scattered across disk.
    - Disk defragmentation tools may help by reorganization.
  - **Internal Fragmentation**
    - Reads/writes are in units of blocks.
    - If a small file cannot cover a block, block space is wasted.
    - Smaller blocks may have more positioning overhead.



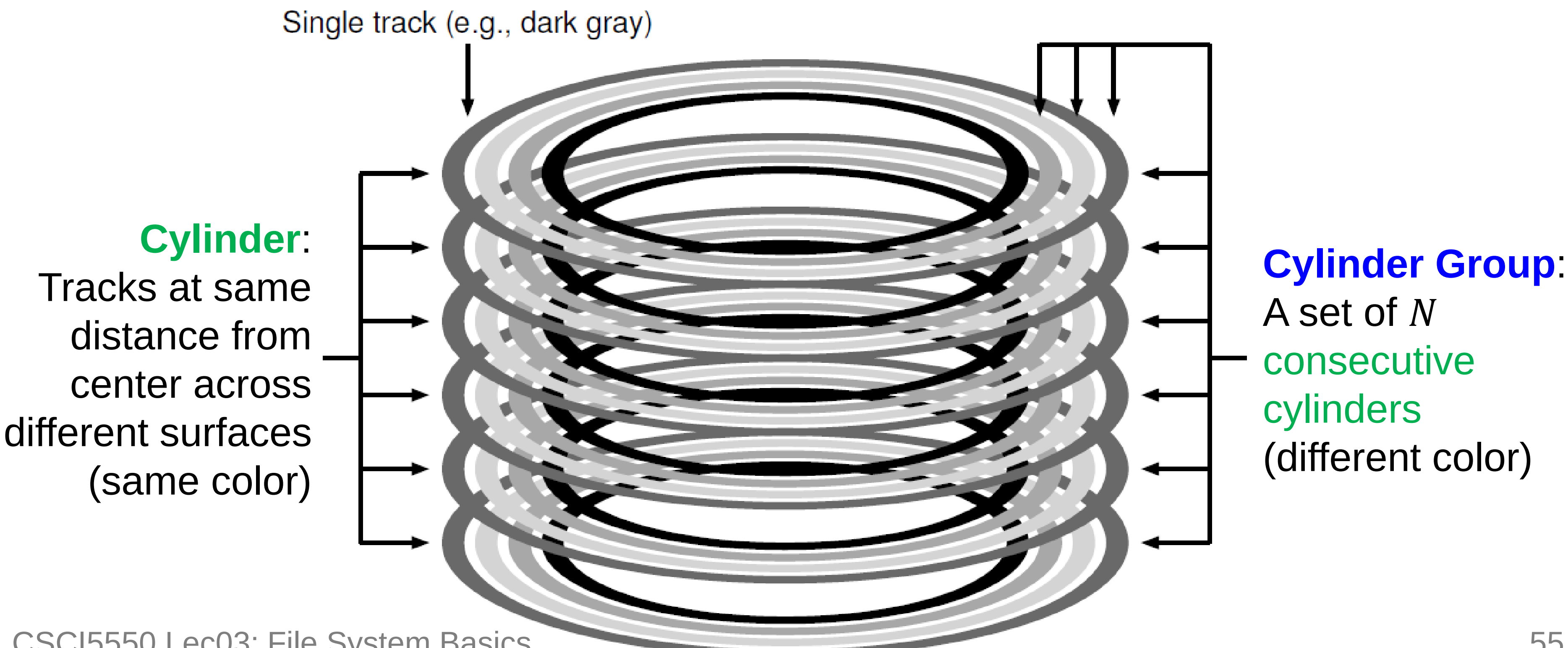
After writing file E of four blocks



After writing file F of 1/2 block

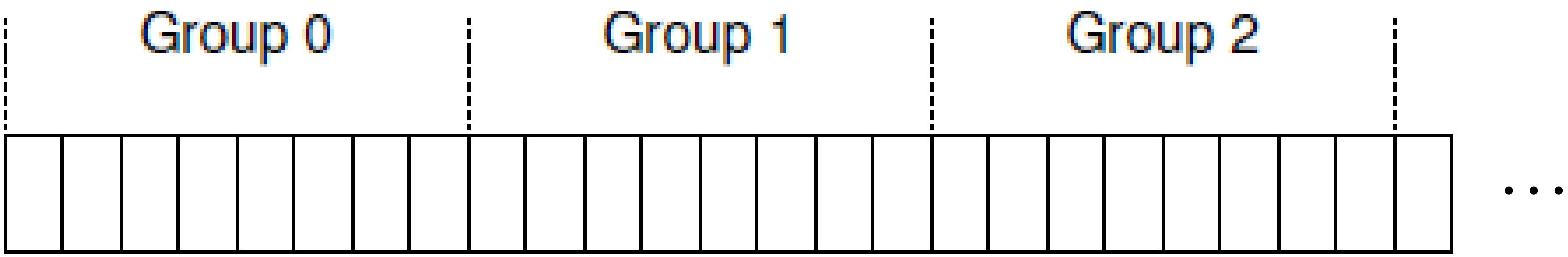
Fast File System (FFS) *by Berkeley*

- **Goal**: Make the file system structures and allocation policies to be "disk-aware" to improve performance.
  - By keeping the same file system interface (i.e., system calls) but changing the internal implementation.
- **Key**: FFS divides disk into **cylinder groups**.



Fast File System (FFS) (2/3)

- FFS aggregates N consecutive cylinders into a group, and the disk is of a collection of **cylinder groups**.



- Modern disks do **not** export cylinder information for the file system to explore.
- Modern FSs instead organize disk into **block groups.**
  - Each block group is of the consecutive block addresses (rather than consecutive cylinders).

Fast File System (FFS) (3/3)

- FFS maintains similar structures **for each group**:
  - A copy of superblock (`S`)
  - Per-group inode bitmap (`ib`) and data bitmap (`db`)
  - Per-group inode and data block regions.

- FFS further explores the data locality to place files, directories, and associated metadata on disk:

keep related stuff together, keep unrelated stuff far apart

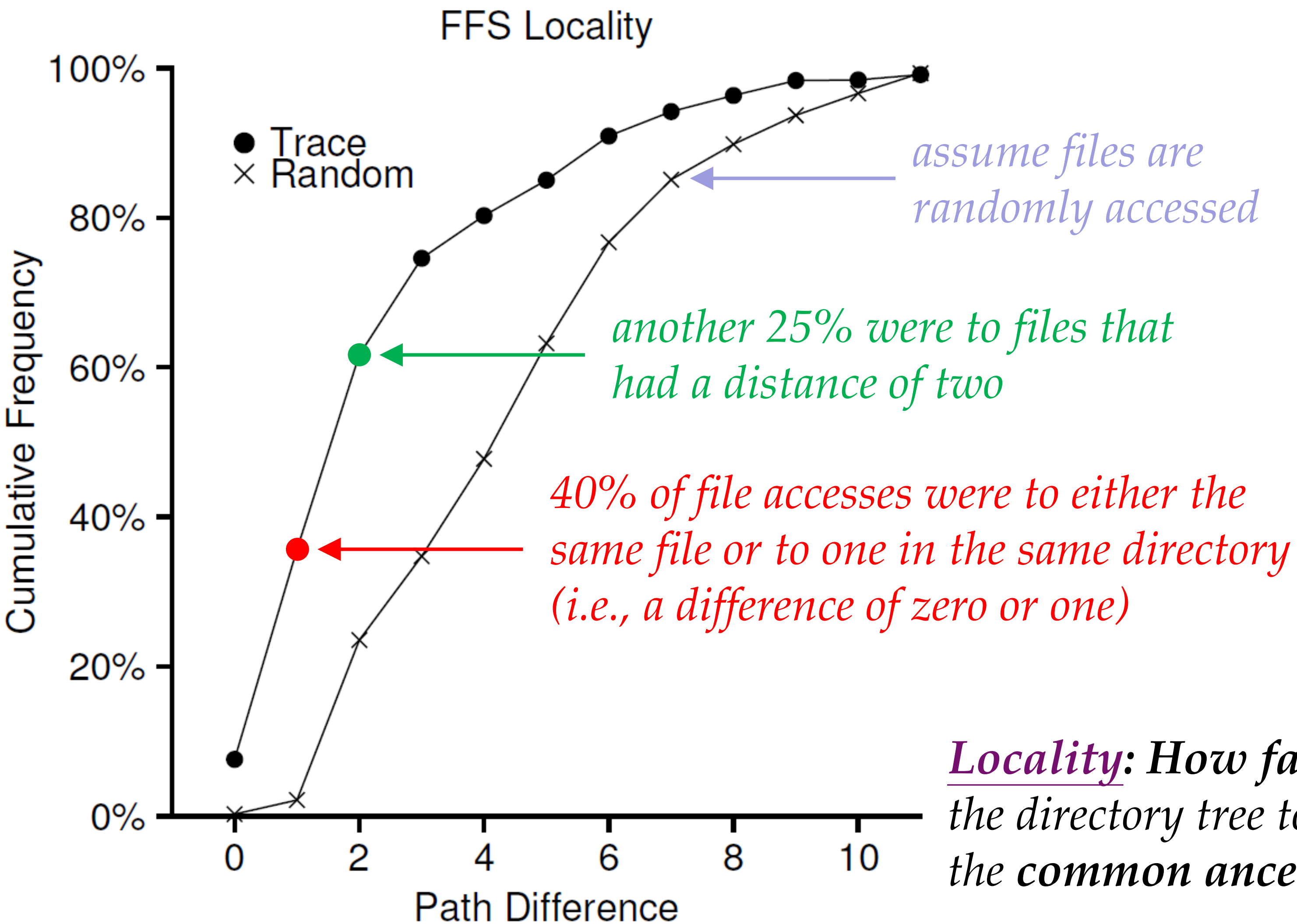
① Allocate **data blocks** of a file in the same group as its inode

② Place **files** of the same directory in the same group

③ Balance **directories** across groups

Data Locality

- Locality (i.e., tendency) is found in real file accesses.



Locality*: How far** *up the directory tree to find the* ***common ancestor

Large-File Exception

- What if file size is larger than group size?
  - Filling the whole group with a large file is undesirable.
  - It prevents "related" files being placed in the same group.

```
group inodes      data
    0 /a-------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
    1 ---------- ---------- ---------- ---------- ----------
    2 ---------- ---------- ---------- ---------- ----------
    ...
```

- FFS divides a file into **chunks** and stores chunks in different groups evenly.
  - Large-enough chunk **amortizes** the positioning overhead.

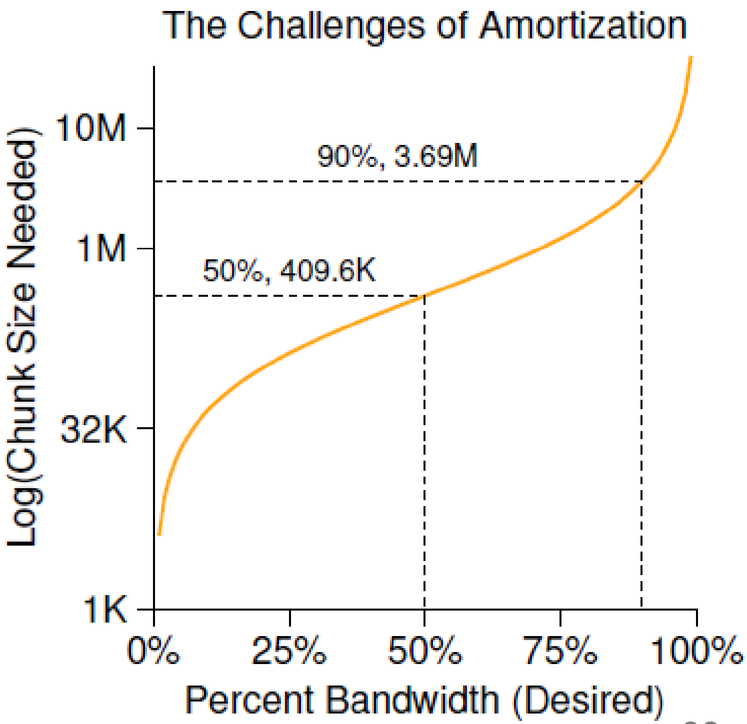
```
group inodes      data
    0 /a-------- /aaaaa---- ---------- ---------- ----------
    1 ---------- aaaaa----- ---------- ---------- ----------
    2 ---------- aaaaa----- ---------- ---------- ----------
    3 ---------- aaaaa----- ---------- ---------- ----------
    4 ---------- aaaaa----- ---------- ---------- ----------
    5 ---------- aaaaa----- ---------- ---------- ----------
    6 ---------- ---------- ---------- ---------- ----------
```

Discussion

- Question: How big does a chunk have to be in order to spend **half (i.e., 50%)** of time in transfer?
- Let's assume that
  - Data transfer rate: 40 MB/s
  - Average disk positioning time: 10 ms
- **Answer**:
  - **Half** of time: **10 ms transferring** for every **10 ms positioning**
  - That is, how many data we can transfer in 10 ms?

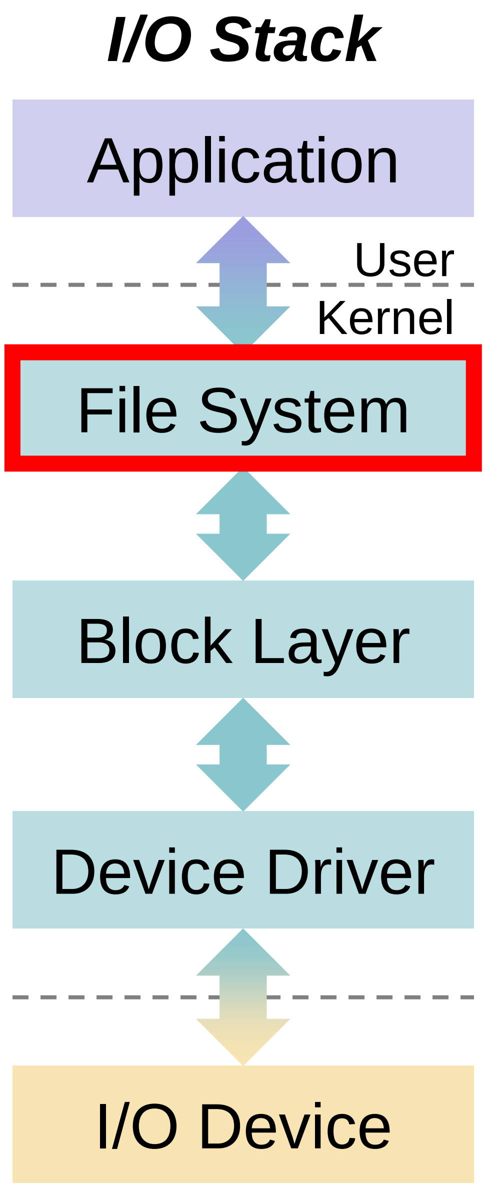
$$\frac{40\ \cancel{MB}}{\cancel{sec}} \cdot \frac{1024\ KB}{1\ \cancel{MB}} \cdot \frac{1\ \cancel{sec}}{1000\ \cancel{ms}} \cdot 10\ \cancel{ms}$$

$$= 409.6\ KB$$



Outline

- File System Organization
  - Abstraction: Files and Directories
  - Metadata Region and Data Region
- File System Interface
- File System Implementations
  - UNIX File System
    - Access Paths: Reading and Writing
    - Caching and Buffering
  - Fast File System (FFS)
    - Disk Awareness
    - Data Locality
  - Crash Consistency
    - File System Checker
    - Journaling



Crash Consistency

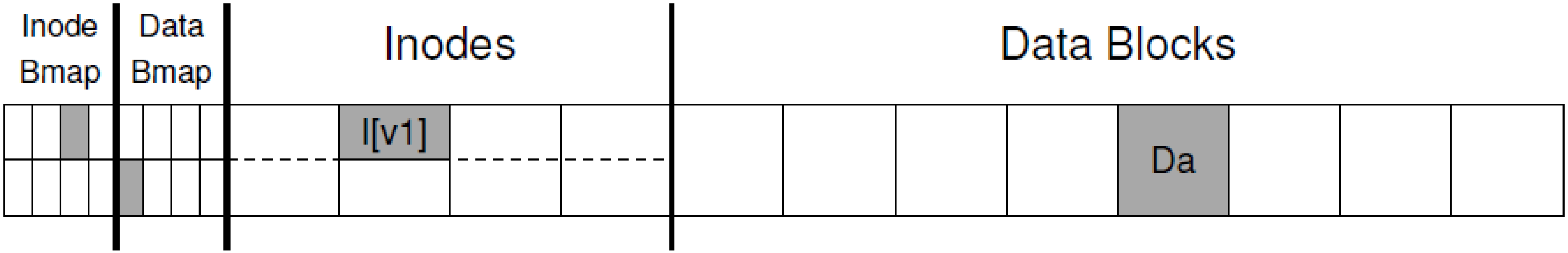
- File system data structures must **persist**.

- **Challenge**: How to update persistent data structures despite the presence of **power loss** or **system crash**.
  - The on-disk structure may be left in an inconsistent state.

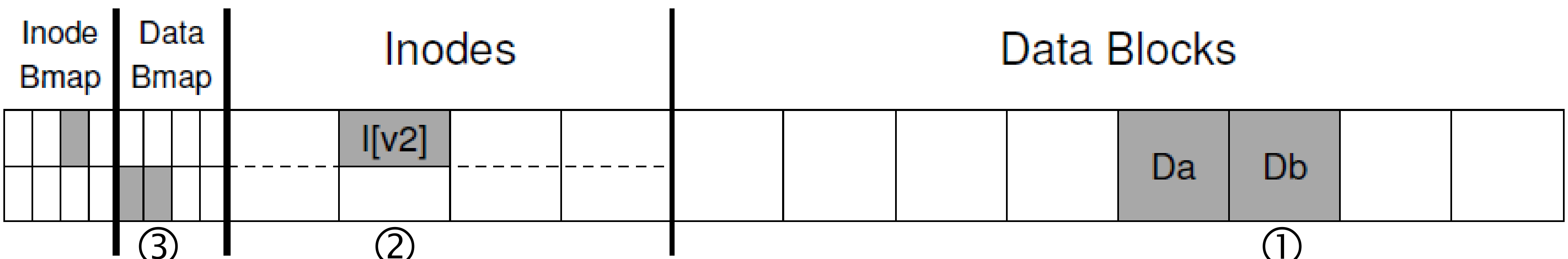
- Solutions to the **crash-consistency problem**:
  ① File System Checker (FSCK)
  ② Journaling (a.k.a. Write-ahead Logging)

Crash Scenarios (1/3)

- Consider **appending a data block** to an existing file:



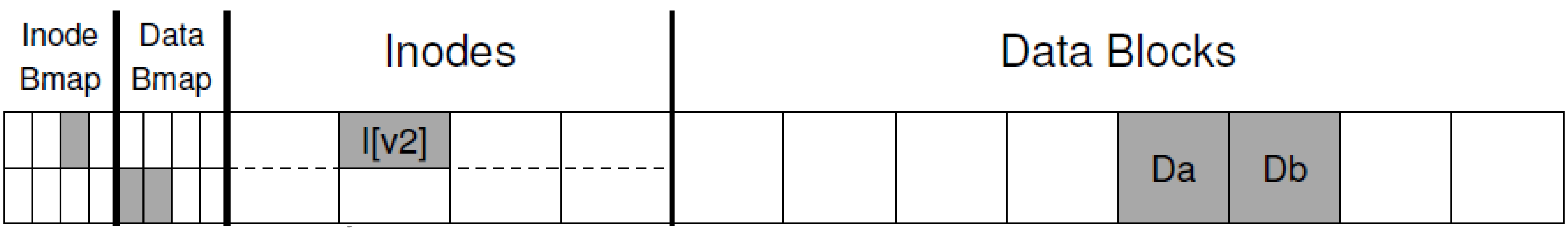
- The file system must perform **three writes**:
  ① the new data block (`Db`)
  ② the inode to point to the new block (`I[v1]` → `I[v2]`)
  ③ the data bitmap to indicate the allocation (`B[v1]` → `B[v2]`)



- A crash may happen at any time. (*How many types?*)

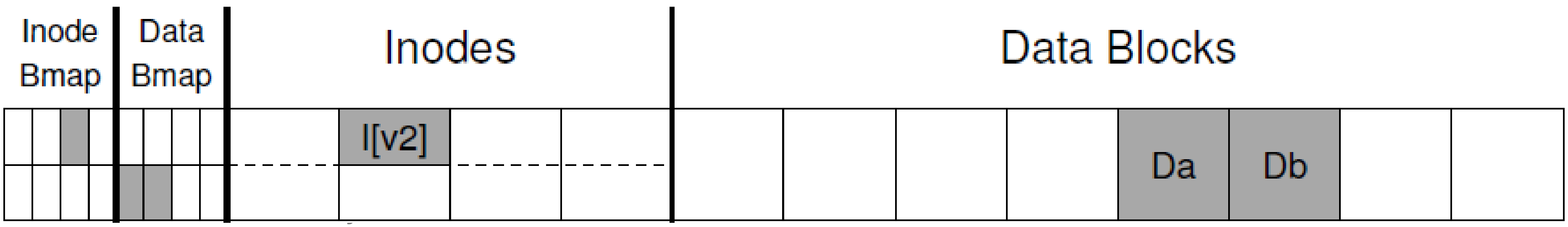
Crash Scenarios (2/3)

- Consider only **one** single write succeeds:
  - **Just the data block (`Db`):**
    - File system remains **consistent**, but user loses data.
    - It is as if the write never occurred.
  - **Just the updated inode (`I[v2]`):**
    - File system is **inconsistent**:
      - Inode says it has data, but bitmap says otherwise (disagreement).
    - If we trust inode, we will read **garbage data** (not `Db`) from the disk.
  - **Just the updated bitmap (`B[v2]`):**
    - File system is **inconsistent**:
      - Bitmap says the block is allocated, but inode says otherwise.
    - It would result in a **space leak**, as the block would never be used.



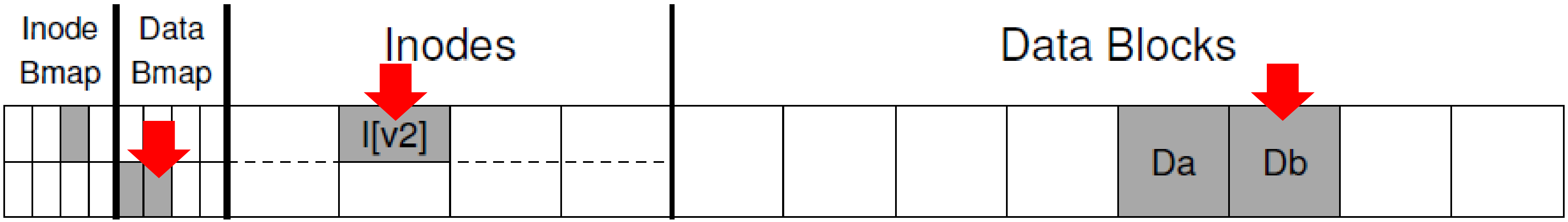
Crash Scenarios (3/3)

- Consider **two** writes succeed:
  - **The inode (`I[v2]`) and bitmap (`B[v2]`):**
    - The file system “metadata” is completely **consistent**:
      - Inode has a pointer to the block, and bitmap also indicates it is in use.
    - But we will read **garbage data** (not `Db`) from the disk.
  - **The inode (`I[v2]`) and data block (`Db`):**
    - File system is **inconsistent**:
      - Inode says it has data, but bitmap says otherwise (disagreement).
    - If we trust inode, we might read **right data** (i.e., `Db`) from the disk.
  - **The bitmap (`B[v2]`) and data block (`Db`):**
    - File system is **inconsistent**.
    - We have no idea which file `Db` belongs to, and face **space leak**.



Naive Solution

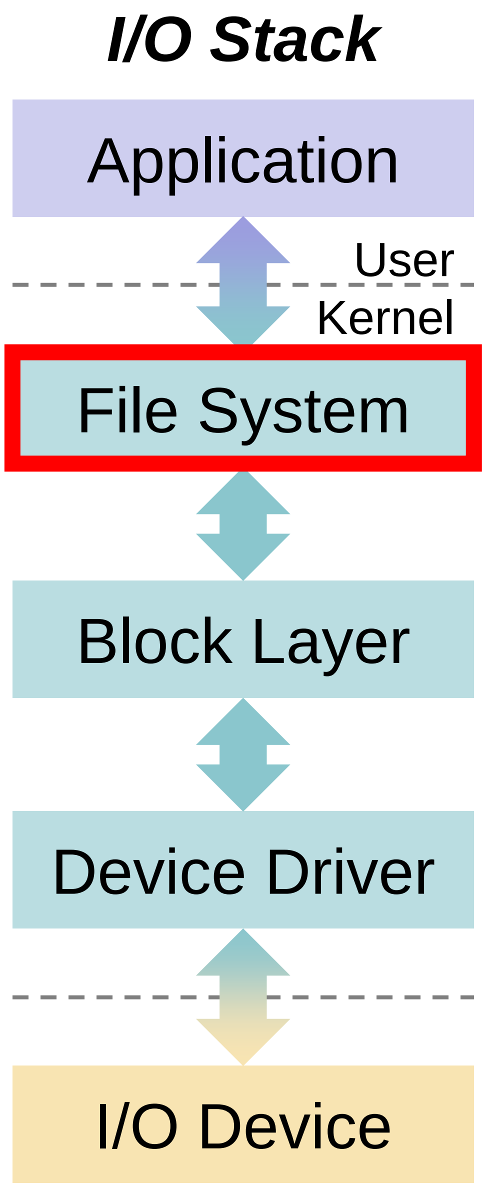
- What we'd like to do ideally is move the file system from one consistent state to another **atomically**.
  - E.g., (before the file got appended) → (after the inode, bitmap, and new data block have been written to disk)



- Unfortunately, we can't do this easily.
  - The disk only commits one write at a time.
  - Crashes or power loss may occur between these updates.

Outline

- File System Organization
  - Abstraction: Files and Directories
  - Metadata Region and Data Region
- File System Interface
- File System Implementations
  - UNIX File System
    - Access Paths: Reading and Writing
    - Caching and Buffering
  - Fast File System (FFS)
    - Disk Awareness
    - Data Locality
  - Crash Consistency
    - File System Checker
    - Journaling

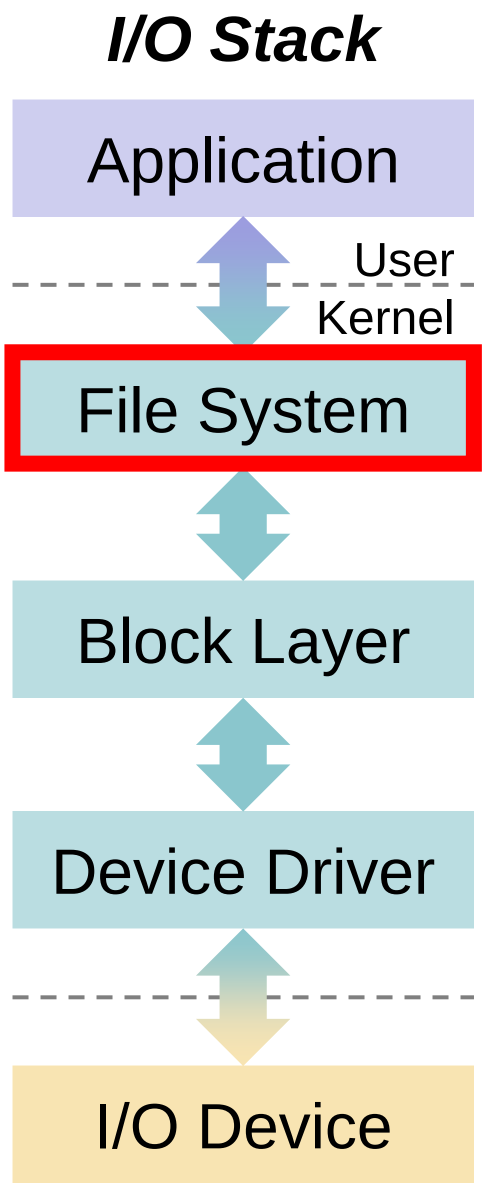


Solution #1: File System Checker

- Early file systems took a simple approach.
  - They let inconsistencies happen and then fix them later.
- `fsck` is a UNIX tool for fixing such inconsistencies.
  - It runs *before* the file system is mounted.
  - It checks superblock, free blocks, inode state, inode links, duplicates, bad blocks, etc., to make sure the **file system metadata** is internally consistent.
    - It does not understand the contents of user files; however, it can perform integrity checks on **contents of directories**.
- **Problems**:
  ① It is **very slow** (especially for large disk volume).
  ② It cannot fix all problems: For example, the file system looks consistent but the inode points to **garbage data**.

Outline

- File System Organization
  - Abstraction: Files and Directories
  - Metadata Region and Data Region
- File System Interface
- File System Implementations
  - UNIX File System
    - Access Paths: Reading and Writing
    - Caching and Buffering
  - Fast File System (FFS)
    - Disk Awareness
    - Data Locality
  - Crash Consistency
    - File System Checker
    - Journaling



Solution #2: Journaling

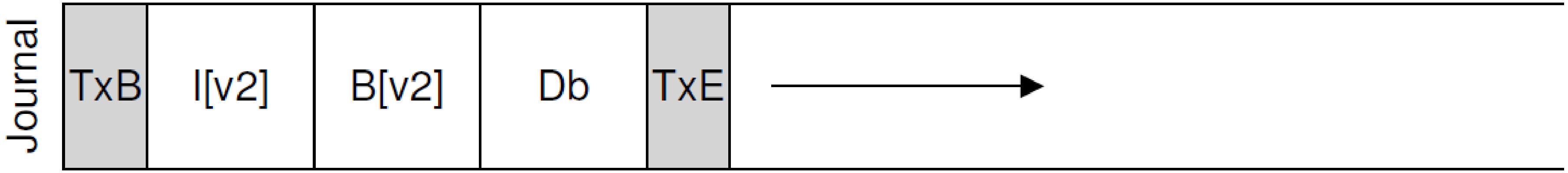
- **Journaling** (or **write-ahead logging**) is the most popular solution to the consistency problem.
  - It first writes a note to a separate **log** structure (somewhere else on the disk) **before** updating the structures in place.
  - It adds a bit of work during updates; but the log tells what to fix after a crash without scanning the entire disk.
- **Linux ext3** incorporates journaling into FS as follows:
  - The disk is divided into block groups as FFS, ext2, etc.
    - Each group has its inode/data bitmap, inodes, and data blocks.
  - Journal (log) occupies some small amount of space.

Super	Journal	Group 0	Group 1	. . .	Group N

- **Question**: What should we note in the journal?

Data Journaling (1/4)

- Consider the example of block appending with three writes: inode (`I[v2]`), bitmap (`B[v2]`), data block (`Db`).
- **Data Journaling**: Write all of them into the log as a transaction, before updating them in place.
  - The **transaction begin** (`TxB`) tells us about this update.
    - Including information about this pending update (e.g., the addresses of the three blocks), and a transaction identifier (`TID`).
  - The **middle** contains the exact contents of the three blocks.
  - The final block (`TxE`) is a marker of the **end** with the `TID`.

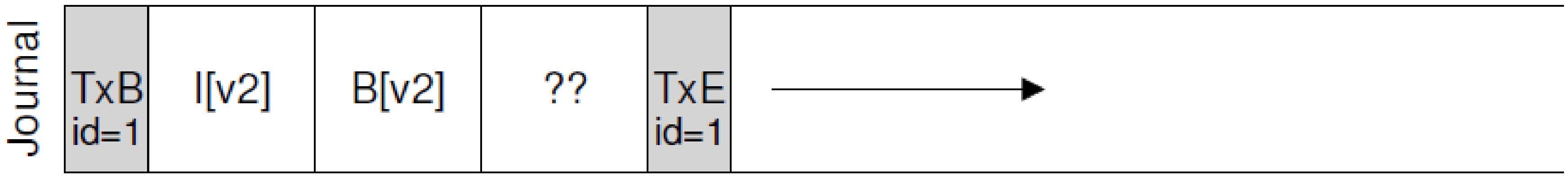


- **Checkpoint**: Write the pending data and metadata updates to the final locations in the file system.

Data Journaling (2/4)

- **Question**: How should data journaling issue the **five writes** of a transaction (`TxB`, `I[v2]`, `B[v2]`, `Db`, `TxE`)?
- Approach #1: Issue them one by one
  - It is safe, but too slow.

– Approach #2: Issue all five writes at once
  - It turns five writes into a sequential one and thus be faster.
  - It is unsafe, since the disk internally re-schedules I/Os.
    - If disk loses power before writing any of them to the journal, the wrong contents are used during replay.
    - For example, the garbage block "??" is copied to the final location of `Db` when the file system replays the transaction.

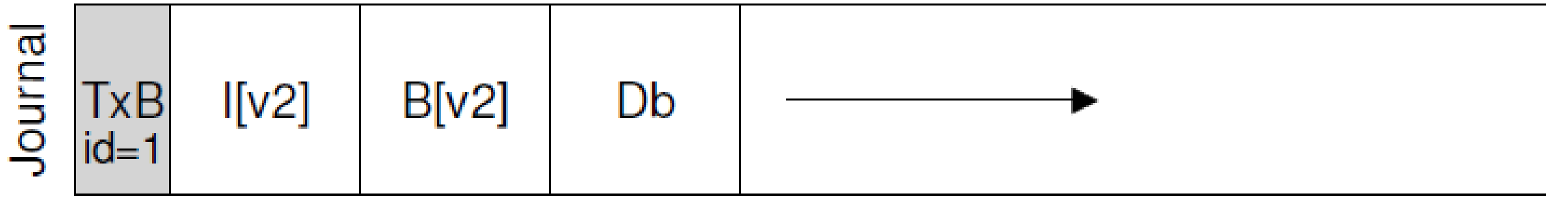


Data Journaling (3/4)

- **(Correct)** Approach #3: Issue writes in two steps

  Step 1) Issue all writes except `TxE` at once

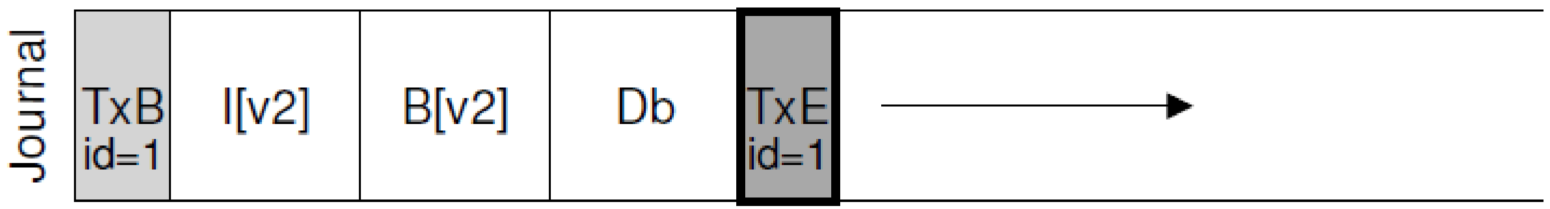
Journal Write



  Step 2) Issue the write of `TxE`, leaving journal in the safe state

  - To ensure the write of TxE is **atomic**, it must be a 512-byte write.

Journal Commit



- The sequence of data journaling:

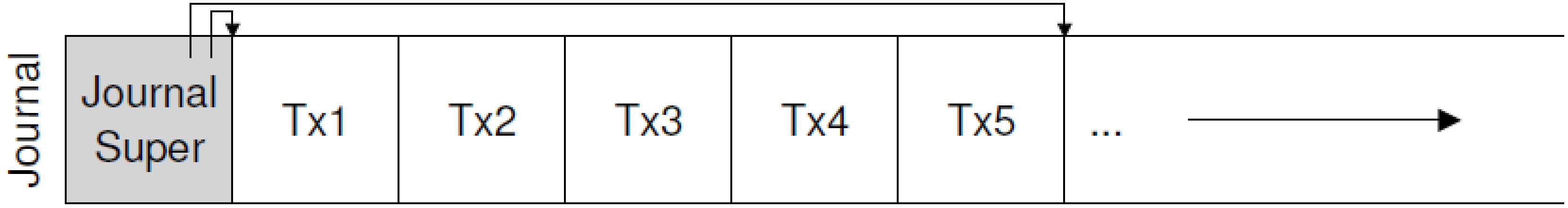
  ① **Journal Write**: Write transaction content except `TxE`

  ② **Journal Commit**: Write the transaction commit block (`TxE`)

  ③ **Checkpoint**: Write pending updates to final disk locations

Data Journaling (4/4)

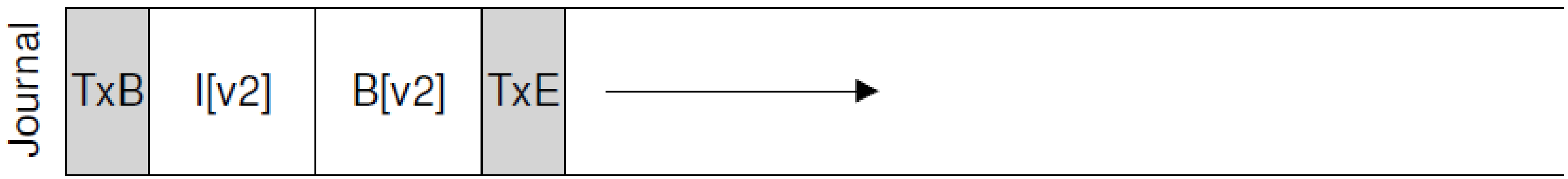
- **Recovery**
  - The file system scans the log and looks for transactions that have committed but not checkpointed yet.
  - Committed transactions are replayed in order (a.k.a., redo).
    - Redundant updates are possible, but they don't hurt consistency.
- **Optimization(s)**
  - **Batching Log Updates**
    - Some file systems (e.g., `ext3`) do not commit each update at a time, but buffer updates into a global transaction to reduce write traffic.
  - **Making The Log Finite**
    - The journal is a finite-sized circular log by marking the oldest and newest non-checkpointed transactions in the journal superblock.



Metadata Journaling

- Data journaling doubles write traffic to the disk.

- **Metadata Journaling** (or Ordered Journaling): Log everything except the user data (i.e., `Db`).



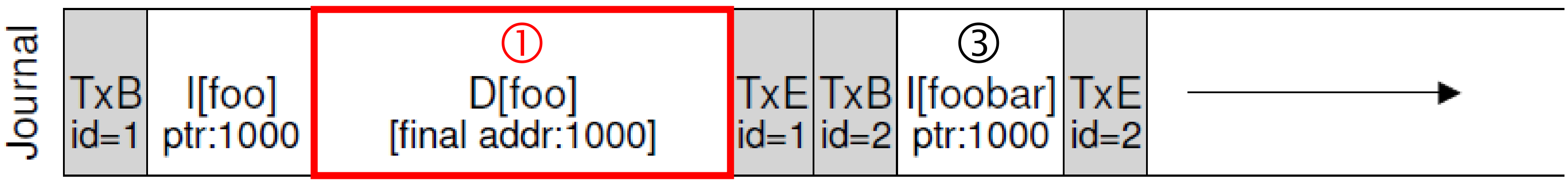
- **Key Issue**: The ordering of user data write is critical.
  - Write `Db` *after* the transaction completes:
    - The file system is consistent.
    - But inode (`I[v2]`) may point to garbage data if the write of `Db` fails.
  - Write `Db` *before* the transaction completes:
    - Both file system and data consistency can be guaranteed.

Metadata Journaling

- The sequence of metadata journaling:
  ① **Data Write**: Write data to final location
  ② **Journal Metadata Write**: Write the begin block (`TxB`) and metadata (`I[v2]`, `B[v2]`) to log
  ③ **Journal Commit**: Write the transaction commit block (`TxE`)
  ④ **Checkpoint Metadata**: Write the contents of metadata update to their final locations within the file system
  ⑤ **Free**: Mark the transaction free in the journal superblock
- Notes:
  – Forcing the data write to complete (Step 1) before issuing writes to the journal (Step 2) is not required.
  – The only real requirement is that Steps 1 and 2 complete before the issuing of the journal commit block (Step 3).

Metadata Journaling

- Tricky Case: **Block Reuse**
  - Let's say we have a directory `foo` at block 1000.
  - Suppose the user ① adds a new entry to the directory `foo`, ② deletes the directory content (**nothing logged!**), and ③ creates a new file `foobar` at block 1000 (by reusing it).



  - The directory content is metadata and should be logged.
  - What happens if we recover from a crash?
    - The recovery process simply replays everything in the log, including the write of directory data (`D[foo]`) in block 1000.
    - This overwrites the user data of new file `foobar` by directory data.
  - **Solution**: Add a revoke record to avoid re-writing old data.

Wrap-up: Journaling Timeline

Data Journaling

TxB	Metadata	Data	TxE	Metadata	Data
Issue	Issue	Issue			
Complete	Complete	Complete			
			Issue		
			Complete		
				Issue	Issue
				Complete	Complete

Metadata Journaling

TxB	Metadata	TxE	Metadata	Data
Issue	Issue			Issue
Complete	Complete			Complete
		Issue		
		Complete		
			Issue	
			Complete	

Other Approaches

- **Copy-On-Write (COW)**
  - Never overwrites files or directories in place.
  - Places new updates to previously unused locations on disk.
  - Includes the newly updated structures after a number of updates are completed.
  - Built on the design of the log-structured file system (LFS).

- **Backpointer-Based Consistency (BBC)**
  - Adds a backpointer to every data block.
  - Achieves lazy crash consistency without ordering.
    - By checking if the forward pointer (e.g., the address in the inode or direct block) points to a block that refers back to it.

Summary

- File System Organization
  - Abstraction: Files and Directories
  - Metadata Region and Data Region
- File System Interface
- File System Implementations
  - UNIX File System
    - Access Paths: Reading and Writing
    - Caching and Buffering
  - Fast File System (FFS)
    - Disk Awareness
    - Data Locality
  - Crash Consistency
    - File System Checker
    - Journaling