EECE7376: Operating Systems Interface and Implementation

Persistence

Information Persistence

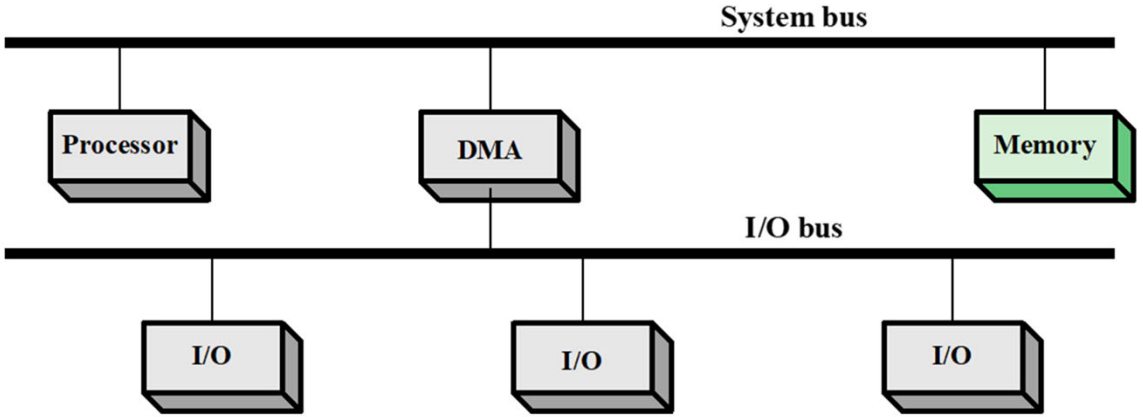
- Making information persist, despite computer crashes, disk failures, or power outages is a tough and interesting challenge.
- A **persistent-storage** device, such as a classic hard disk drive (HDD) or a more modern solid-state storage device (SSD), stores information permanently (or at least, for a long time).
- Unlike memory, more specifically the RAM, whose contents are lost when there is a power loss, a persistent-storage device keeps such data intact.
- Thus, the OS must take extra care with such a device: this is where users keep data that they really care about.

I/O Device Drivers

- HDD and SSD are I/O devices that need drivers to encapsulate within them the specifics of all interactions with the OS.
- Such encapsulation provides an **abstraction** level to the OS to handle something like the file system without the need to deal with the details of how to issue a read or write request to different types of storage devices.
- Disk accesses typically take milliseconds, **a long time** for a processor.
- The **boot loader** issues disk read commands and reads the **status bits** repeatedly until the data is ready. This **polling** approach is fine in a boot loader, which has nothing else to do.
- In regular operating system operations, it is more efficient to let another process run on the CPU and arrange to receive an **interrupt** when the disk operation has completed.

Direct Memory Access (DMA) (1/2)

- When interacting with I/O devices to transfer a large chunk of data, the CPU would be overburdened with a rather trivial task (e.g., moving a large amount of data between the main memory and the I/O device), and thus wastes a lot of time and effort that could better be spent running other processes.
- The **DMA** engine is essentially a very specific device within the computer system that orchestrates transfers between devices and main memory without much CPU intervention.



Direct Memory Access (DMA) (2/2)

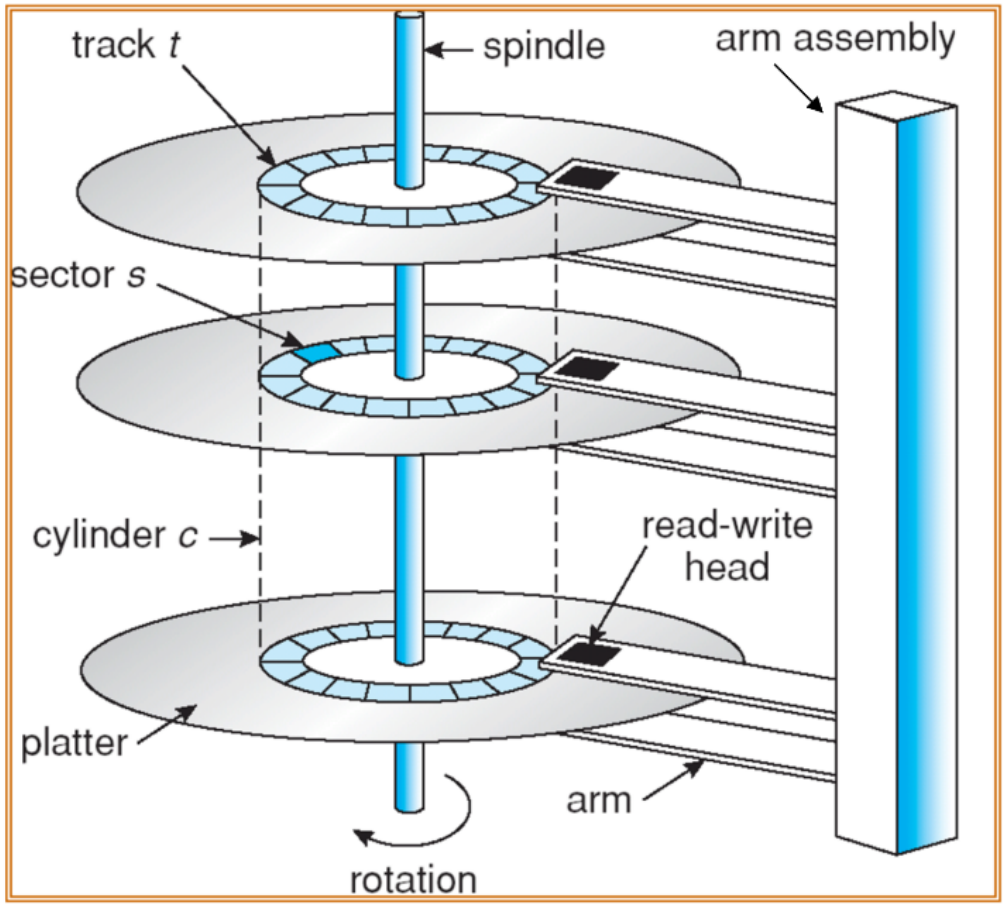
- When a process issues a system call to transfer data to a device, for example, the OS moves the process to the **"sleeping"** state and instructs the **DMA** engine by telling it where the data lives in memory, how much data to copy, and which device to send it to.
- At that point, the OS is done with **initiating the transfer process** and its scheduler instructs the CPU to proceed with other processes.
- When the DMA is done with the actual transfer, it raises a **hardware interrupt** causing the CPU to jump into the OS at a predetermined interrupt service routine (**ISR**).
- The OS **concluded the data transfer** by resuming the process (moving it to the "**runnable**" state) that initiated the data transfer request.

Hard Disk Drives

- The **hard disk drives** (HDD) have been the main form of **persistent data storage** in computer systems for decades and much of the development of **file system technology is based on their behavior**.
- While SSDs are faster, quieter, and smaller in size, HDDs are cheaper per-gigabyte, available in larger capacity, and easier data recovery if damaged.
- While personal computers rely now on SDDs, for data centers, HDDs remain essential.

Hard Disk Drives Anatomy

- **Platter**, is a circular hard surface on which data is stored.
- Each platter has two **surfaces**.
- Data is encoded on each surface in concentric circles called **tracks**. A surface contains thousands of tracks.
- Each track has multiple **sectors**.
- A sector represents a **block** of data that is the smallest unit to read/write from/to the disk.
- A typical sector contains 512 bytes. More sectors are on an outer track than on an inner track



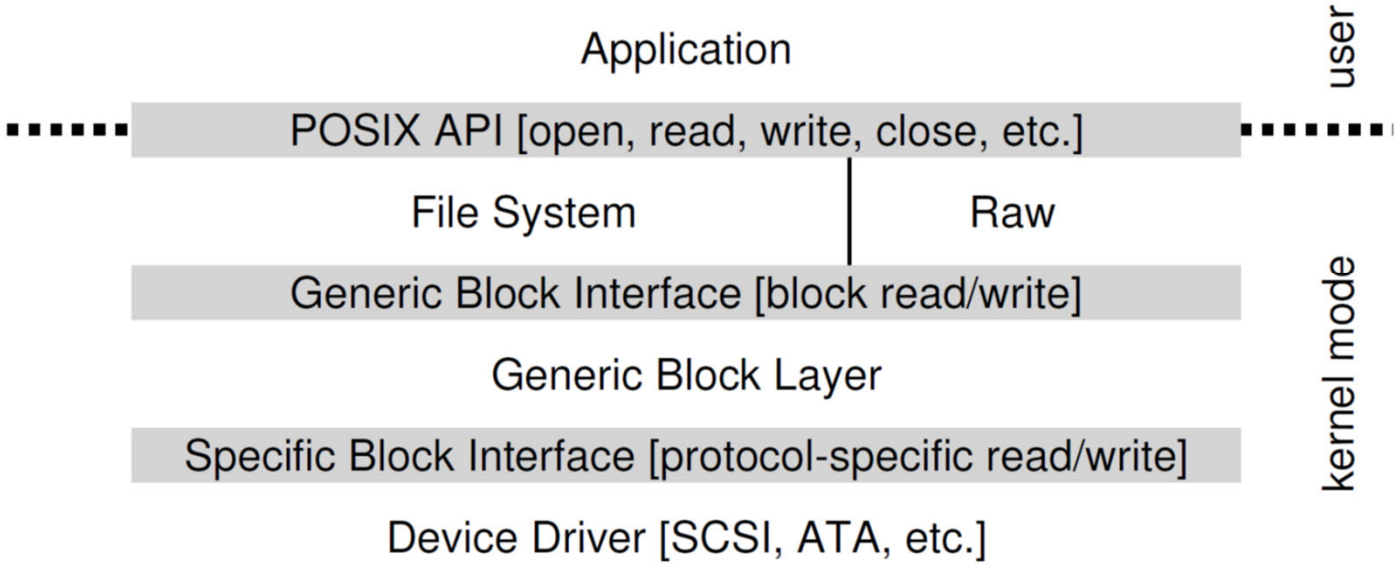
Disk Read and Write Times

- **Seek time**: Time for heads to move to appropriate track.
  - The appropriate head is enabled
- **Rotation latency time**: Time for the sector to appear under the head.
  - Assume 7,200 rotations per minute (RPM)
  - 7,200 / 60 = 120 rotations per second
  - 1/120 = ~8 msec per rotation
  - Average rotation delay is 8 ÷ 2 = ~4 msec
- **Transfer time**:
  Time to Read/write the sector.
  - 58 Mbytes/sec
  - 4-Kbyte disk blocks
  - Time to transfer a block takes:
    (4/(58*1000)) *1000 = 0.07 msec
- Disk I/O time = $T_{I/O}$ =
  seek + rotation + transfer
- Disk I/O rate = $R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}}$

File System Stack

- The figure shows the Linux file system stack.
- The application and the file system are **completely unaware** of the specifics of which disk technology they are using; they simply issue block read and write requests to the generic block layer, which routes them to the appropriate device driver.

- The shown **raw** interface enables special applications (such as a **disk defragmentation** tool) to directly read and write blocks without using the file system abstraction.

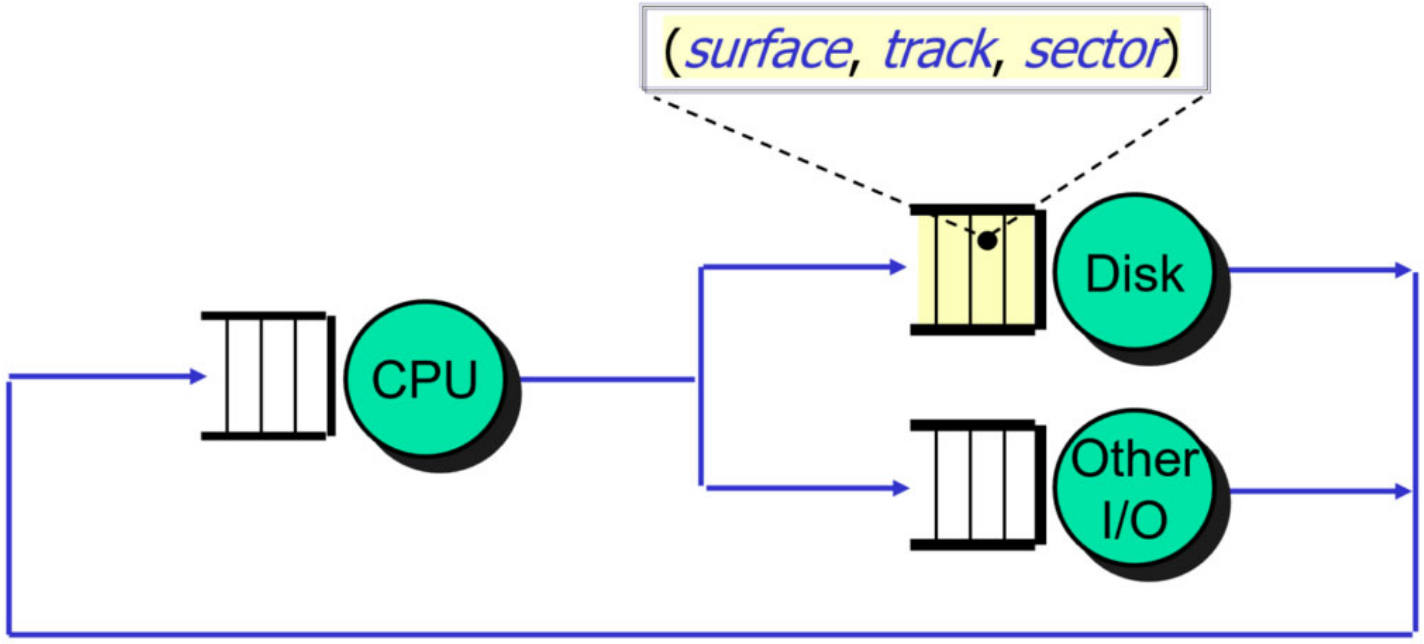


Disk Scheduling

- The OS plays a role in deciding the order of I/Os issued to the disk. More specifically, **given a set of I/O requests**, the **disk scheduler** examines the requests and decides which one to schedule next.
- Unlike job scheduling (in CPU Virtualization), where the length of each job is usually unknown, with disk scheduling, we can make a good guess at how long a "job" (i.e., disk request) will take.
- Therefore, the disk scheduler might, for example, "greedily" pick the one that will take the least time to service first.

Disk I/O Requests

- The figure shows a queue of disk I/O requests in the form of a 3-tuple: (*surface*, *track*, *sector*).
- The OS maximizes disk I/O throughput by minimizing head movement through **disk head scheduling**.
- The scheduler can achieve this goal by rearranging the queue requests based on the **track field**.



Disk I/O Requests Example

- Several algorithms exist to schedule the servicing of disk I/O requests.
- Suppose that a disk drive has 200 tracks, numbered 0 to 199.
- The head is currently on track 53.
- The queue of pending track requests is:

1st request

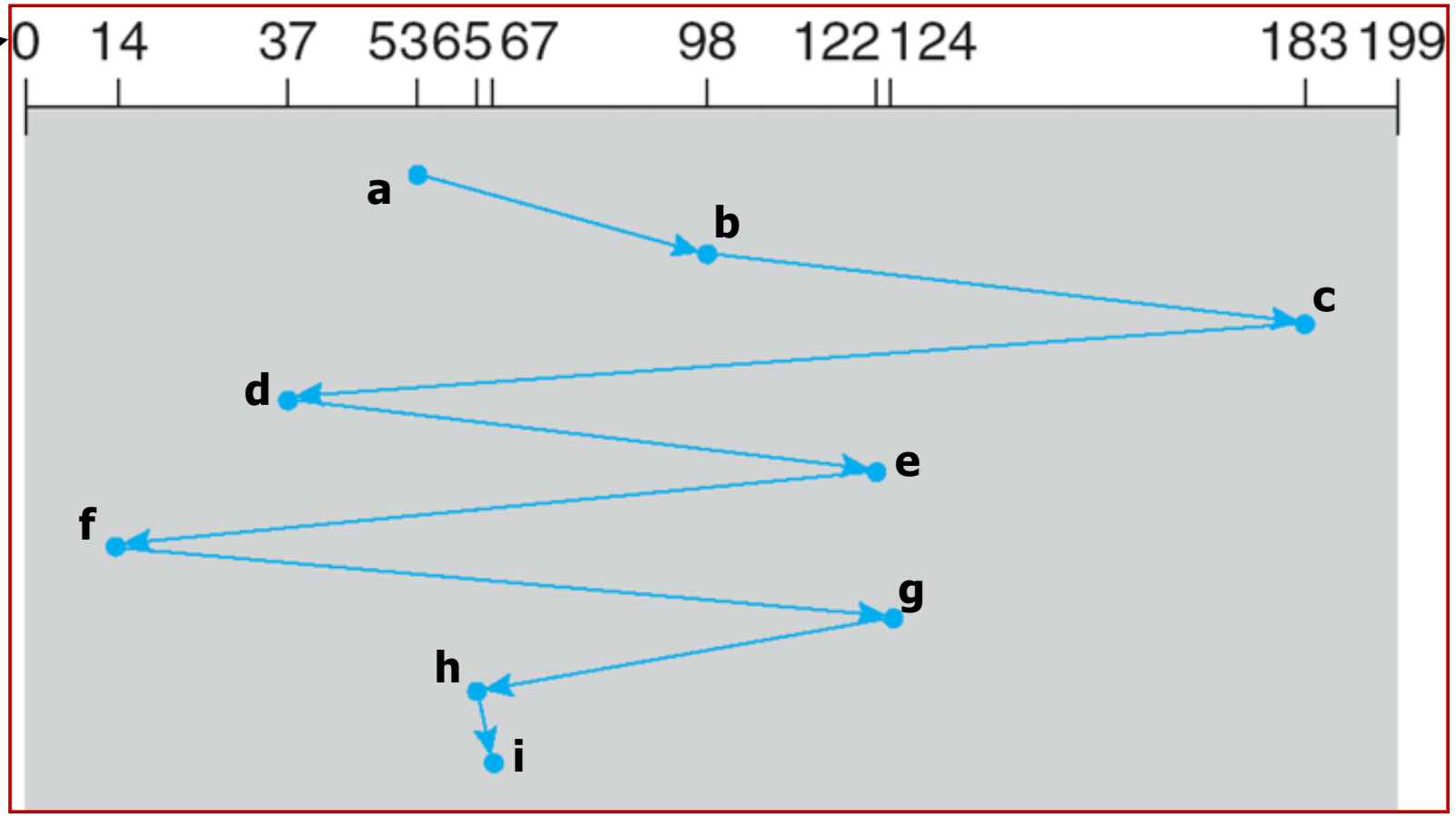
98	183	37	122	14	124	65	67	

Disk Head Scheduling: FCFS

Assume Starting at track 53

98	183	37	122	14	124	65	67	

A slice of the disk from the spindle to the outer edge



FCFS scheduling results in the head moving about 640 tracks

Can we do better?

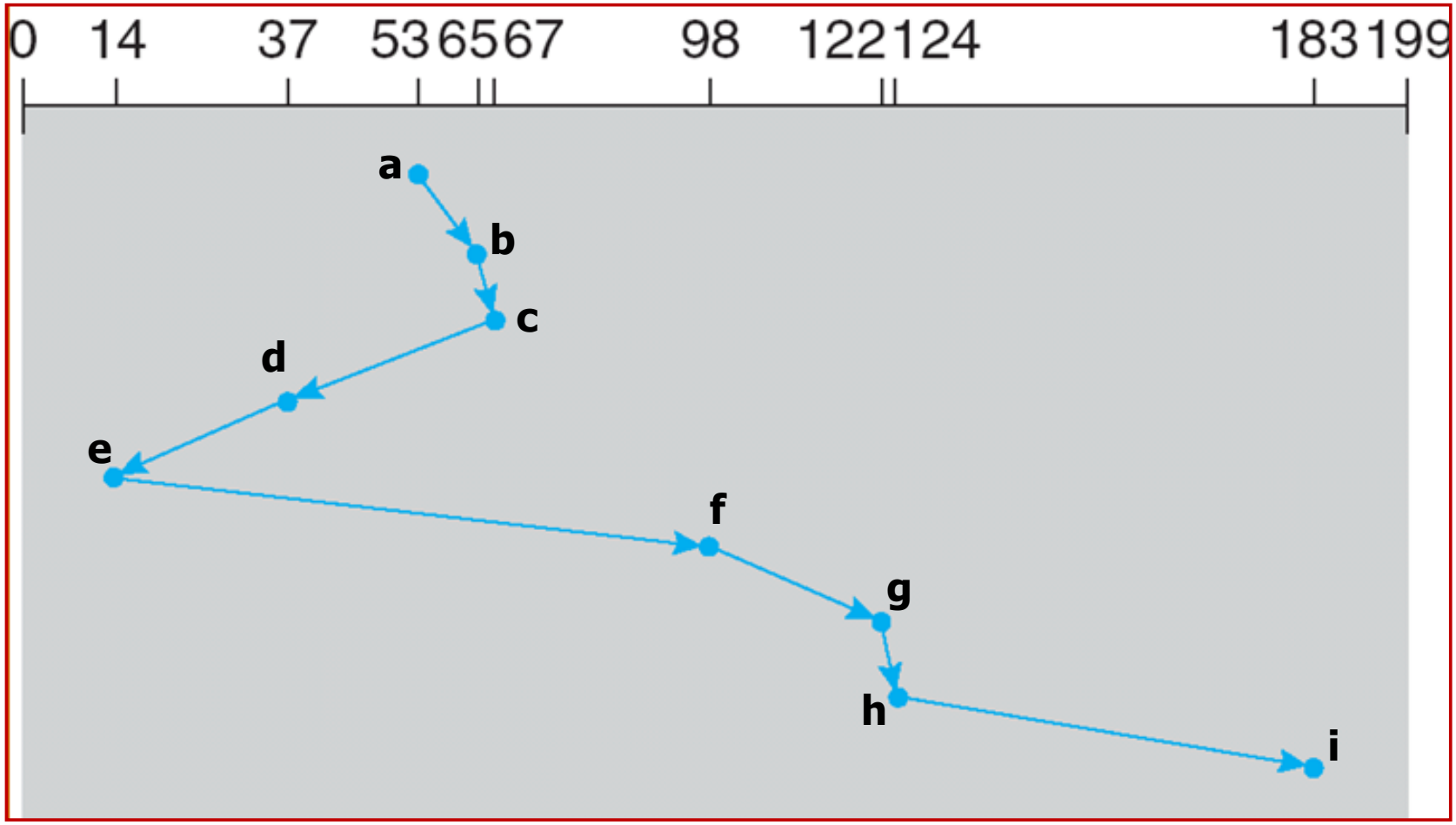
Disk Head Scheduling: SSTF

- Greedy scheduling: **shortest seek time first** (SSTF)
- Selects the request with the minimum seek time from the current head position.
- The idea is that as new requests arrive, they are inserted into the queue in SSTF order.
- SSTF scheduling is a form of SJF scheduling; may cause **starvation** of some requests.
- Imagine if there were a steady stream of requests to the inner track, where the head currently is positioned. Requests to any other tracks would then be ignored completely by a pure SSTF approach.

SSTF Example

Assume Starting at track 53

98	183	37	122	14	124	65	67	



SSTF scheduling results in the head moving about 236 tracks.
Significant improvement over the 640 tracks with FCFS!

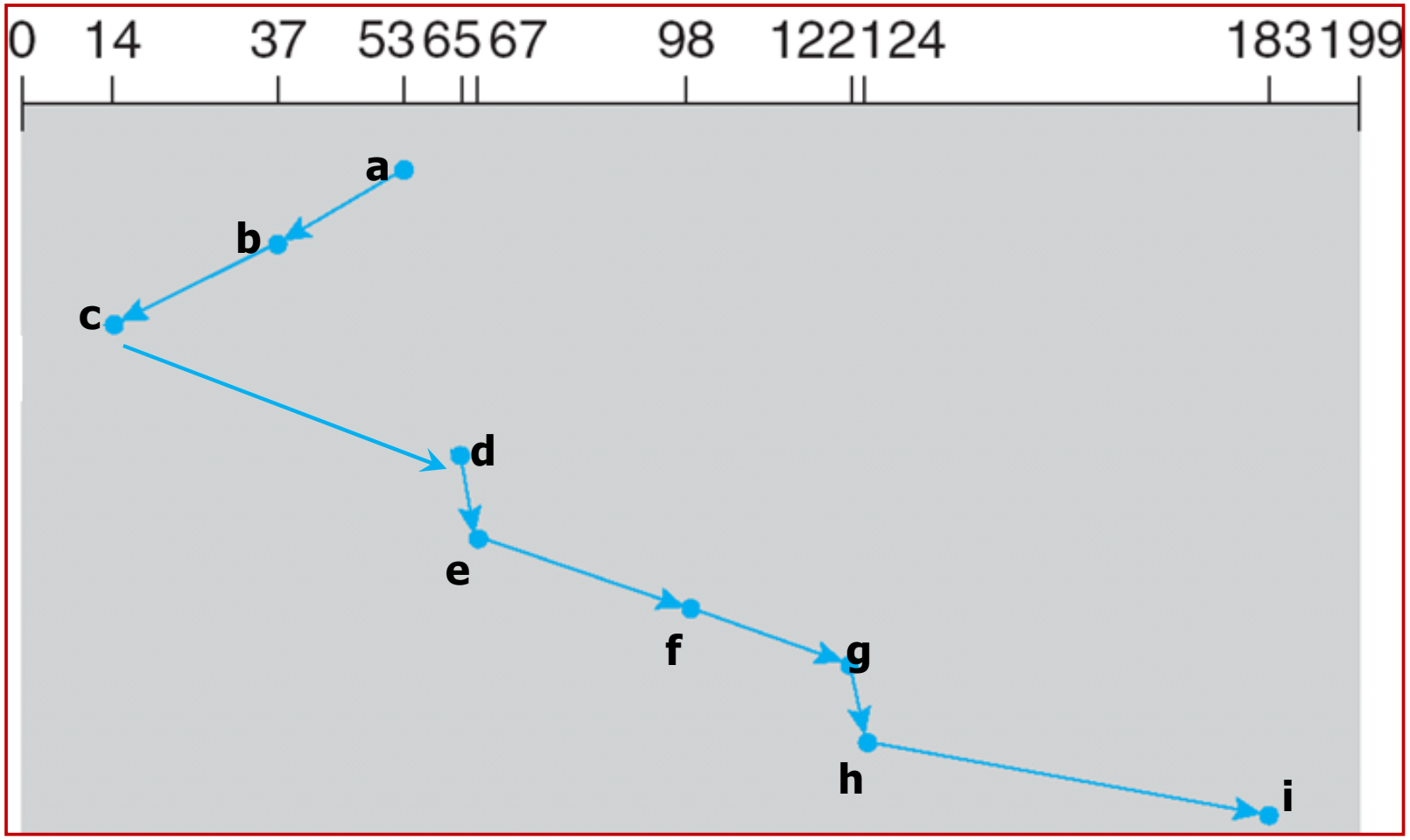
Disk Head Scheduling: SCAN

- The disk moves the head in one direction until all requests have been serviced and then reverse and servicing continues until it reaches the last request.
- Sometimes it is called the **elevator algorithm**. Imagine an elevator is using the SSTF algorithm instead?
- Let's call a single pass across the disk (from outer to inner tracks, or inner to outer) a **sweep**.
- SCAN **favors the middle tracks** as after servicing one end of the disk, SCAN passes through the middle twice before coming back to that same end again.

SCAN Example

Assume Starting at track 53

98	183	37	122	14	124	65	67	



SCAN scheduling results in the head moving over 208 tracks.
Better than the SSTF 236 tracks.

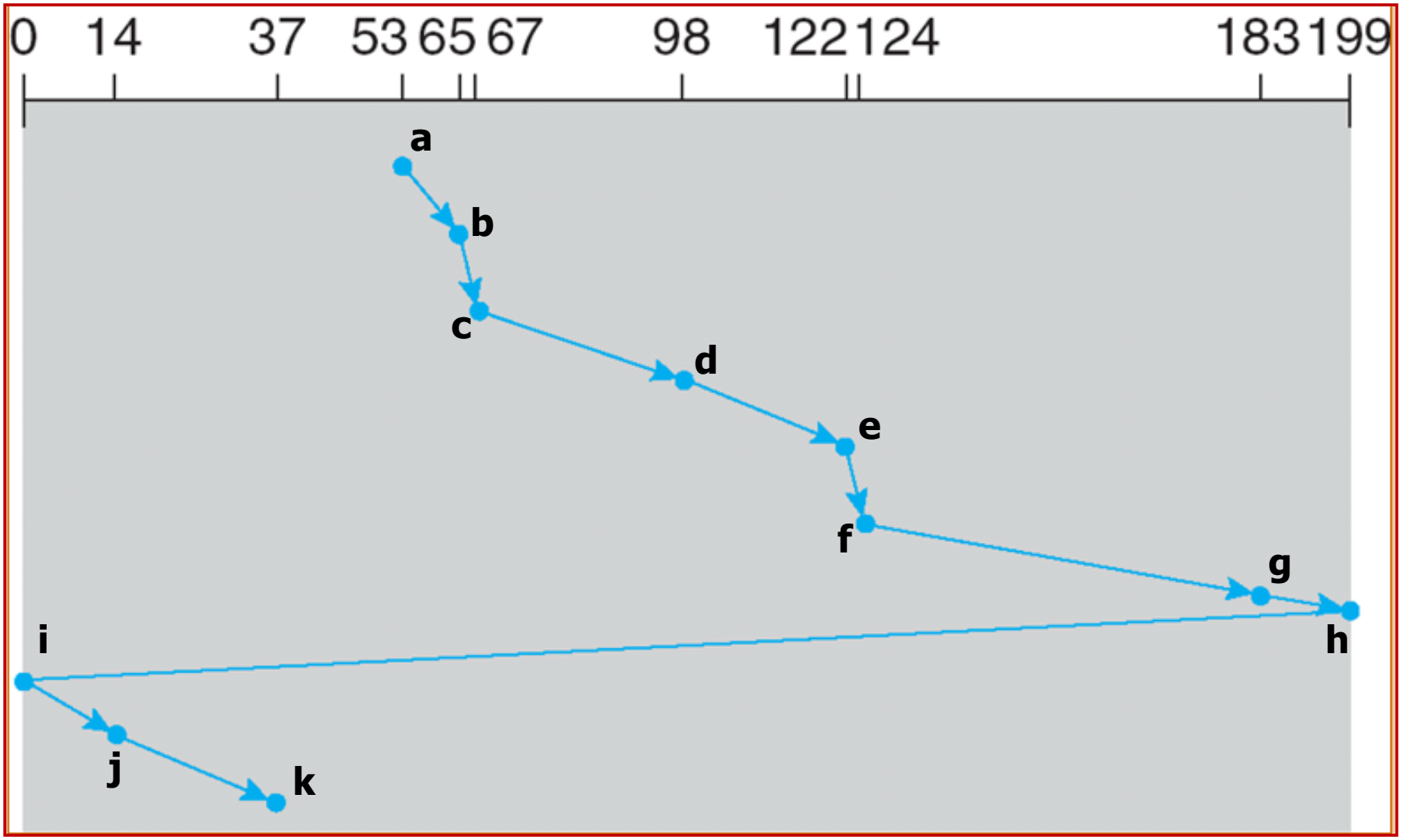
Disk Head Scheduling: C-SCAN

- The head moves from its current position towards one end of the disk servicing requests as it goes.
- When it reaches the end, it immediately returns to the other end of the disk, without servicing any requests on the return trip (i.e., it **sweeps** only on one direction).
- From that other end it continues on the original direction serving the remaining requests.
- Treats the disk as a **circular** list that wraps around from the last track to the first one.
- C-SCAN provides a **more uniform wait time** than SCAN and it is a bit fairer to inner and outer tracks.
- With C-SCAN repositioning to the beginning is faster than repositioning in small pieces because of acceleration and deceleration (inertia).

C-SCAN Example

98	183	37	122	14	124	65	67	

Assume Starting at track 53 and the head moves towards the outer end of the disk first



C-SCAN scheduling results in the head moving over 382 tracks

Where is Disk Scheduling Performed?

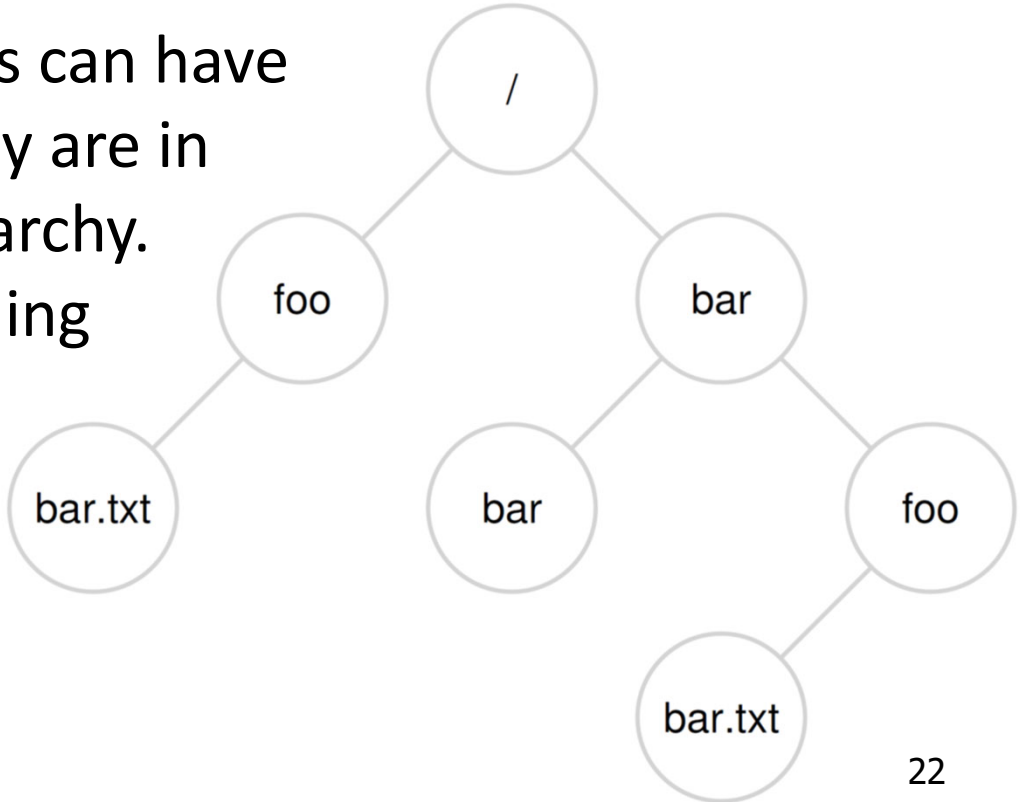
- In older systems, the operating system did all the scheduling; after looking through the set of pending requests, the OS would pick the best one (based on the used scheduling algorithm), and issue it to the disk.
- In modern systems, disks can accommodate multiple outstanding requests, and have sophisticated **internal schedulers**. Such internal schedulers can be implemented more accurately as all relevant details are available, including exact head position.
- Thus, the OS scheduler usually picks what it thinks the best few requests are (say 16) and issues them all to disk; the disk then uses its internal knowledge of head position and detailed track layout information to service said requests in the best possible order (e.g., using **shortest positioning time first SPTF**)

Files And Directories

- Files and directories are two key abstractions that have been developed over time in the **virtualization of storage**.
- A **file** is simply a linear array of bytes, each of which you can be read or written.
- Each file has an associated **inode** (index node) stored with the file in the disk.
  - The inode **data structure** contains its associated file metadata.
  - It allows any compatible system to be able to recognize the disk contents.
- A **directory** also has its own inode and it contains a list of {*user-readable name*, *inode number*} pairs.
- Each entry in a directory refers to either a file or another directory.
- By placing directories within other directories, users are able to build an arbitrary **directory tree** (or **directory hierarchy**), under which all files and directories are stored.

Directory Tree (Hierarchy)

- The directory hierarchy starts at a **root directory** (in Linux-based systems, the root directory is simply referred to as /).
- In the figure, a user created a directory `foo` in the root directory `/`, and then created a file `bar.txt` in the directory `foo`, we could refer to the file by its **absolute pathname**, which in this case would be `/foo/bar.txt`.
- As shown, directories and files can have the same name as long as they are in different locations in the hierarchy.
- Thus, we can see one great thing provided by the OS:
  a convenient way to name all the files we are interested in.



Creating Files

- This code creates a file named "foo" in the current directory:

```
#include <fcntl.h>
...
int fd;
...
fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

- The second parameter has the following flags:
  - O_CREAT to create the file if it does not exist.
  - O_WRONLY to ensure that the file can only be written to.
  - O_TRUNC to truncate the file to a size of zero bytes (removing any existing content if the file already exists).
- The third parameter specifies permissions by making the file readable (S_IRUSR) and writable (S_IWUSR) by the owner.

Recall the xv6 Process Structure

- The **`struct proc`** (in **`proc.h`**) is the per process data structure:

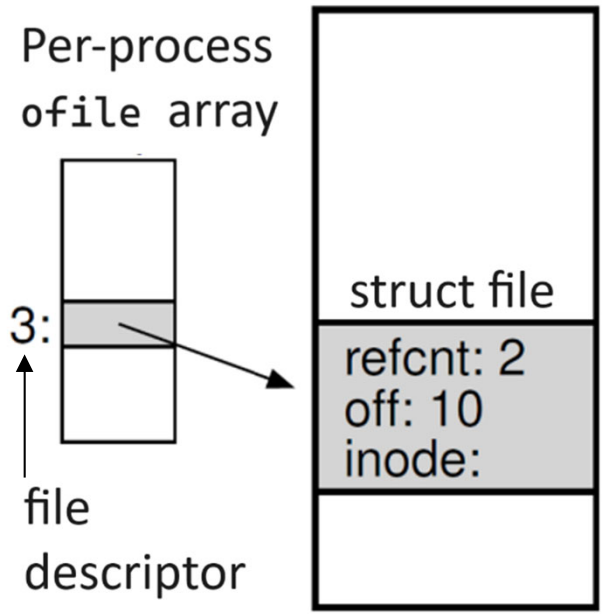
```
enum procstate // all possible process states
        { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
// Per-process data structure
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page directory
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // defined in proc.h with registers needed for swtch()
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files associated with the process
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

File Descriptor

- Function `open()` returns a **file descriptor**, which is an integer used in Linux systems to access the file. It is used in calling other file related "methods" like `read()` and `write()`.
- File descriptors are managed by the operating system on a per-process basis by keeping some kind of an array in the Linux systems **proc** structure like the following in xv6:

```
struct file *ofile[NOFILE];  // an array (table) of max NOFILE open files
```

- Each entry of the table is just a **pointer to a struct file**, which will be used to track information about the file being read or written.
- The **file descriptor** is the **index** to the corresponding file pointer in this table. It is private per process.



The `file` Structure

- The **`struct file`** is a kernel structure that represents an *open file* and is associated with the process that opens the file
- It is **created by the kernel** when a process issues an *open* command and is passed to any function that operates on the file (indirectly through the file descriptor).
- It is cleared after last ***close*** call on the file.
- The following are examples of the `struct file` fields:
  - `fmode_t f_mode`: The file mode identifies the file as either readable or writable (or both).
  - `loff_t f_pos`: The file current reading or writing position. Updated only on read and write operations.
  - `struct inode *f_inode`: to get to the inode of the file .

Reading from a File

- Reading a file copies bytes from the **current file position** to memory, and **then updates file position**.

```
char buf [512];
int fd ; /* file descriptor */
int nbytes ; /* number of bytes read*/
/* Open file fd ... */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd , buf , sizeof(buf))) < 0){
 perror ("read");
 exit(1);
}
```

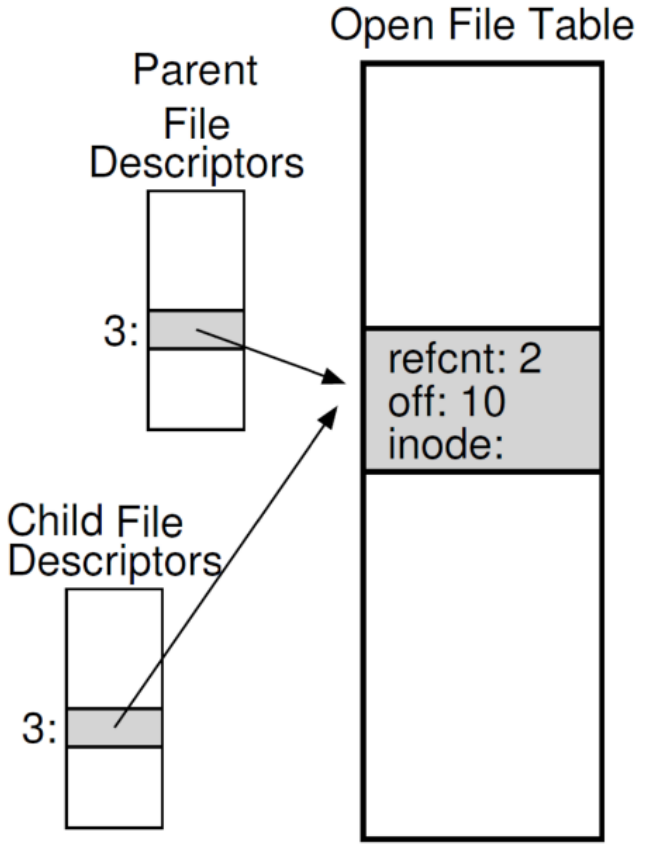
Writing to a File

- Writing a file copies bytes from memory to the **current file position**, and **then updates file position**.

```
char buf [512];
int fd ; /* file descriptor */
int nbytes ; /* number of bytes read */
/* Open the file fd ... */
/* Fill buf with what you want to write and
then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd , buf , sizeof(buf))) < 0) {
   perror("write");
   exit(1);
}
```

Shared Files

- Each process has **its own open file table**, which contains pointers to the **`struct file`** structures of the opened files.
- If **another process** opens the same file at the same time, each process will have, in its open file table, its own entry of the **`struct file`** containing that file info . This way, **each reading or writing of a file is independent**.
  - However, an entry in the open file table is **shared** when a parent process creates a child process with `fork()`. This way both the parent and the child can **cooperate** working on the file (e.g., **parent continues reading from where the child stopped reading**).
    - As shown, when a file table entry is shared, its **reference count** is incremented.
    - **inode** refers to a data structure that contains the file metadata and it is **the same for all processes accessing the same file**.



File Reading and Writing Commands

- Using output redirection, ">", the following Linux `echo` command line program is used to create a file `foo` and write "`hello`" in it. The program `cat` dumps the contents of the file to the screen.

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

- What system calls does the `cat` program use to display the content of a file?
- On Linux, use the **`strace`** tool to trace every system call made by a program while it runs, and dump the trace to the screen.

Tracing `cat` (1/2)

- Here is an example of using `strace` to figure out what cat is doing (some calls removed for readability):

```
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
...
prompt>
```

- It opens the file `foo` for reading and function `open()` returned 3 as the file descriptor. Why 3?
  - Each running process starts with three opened files: **standard input**, **standard output**, and **standard error** (in which the process writes error messages). These have file descriptors 0, 1, and 2, respectively.

Tracing `cat` (2/2)

- After the open succeeds, cat uses the `read()` system call to repeatedly read some bytes from a file. In the example, it reads from file descriptor 3 (i.e., file `foo`) and stores the result in a buffer that has a size of 4 KB.
  - The first `read()` call returns 6 (as it read 6 characters) while the second `read()` call returns 0.
- The `write()` system call, writes the buffer returned from `read()` to file descriptor 1 (i.e., the standard output).
- Once `read()` returns 0, the program knows that it has read the entire file. Thus, the program calls `close()` to indicate that it is done with the file "foo".

Files Permissions (1/2)

- The file system presents a virtual view of a disk. However, the abstraction is notably different from that of the CPU and memory, in that files are commonly **shared** among different users and processes and **are not (always) private**.
- Thus, a more comprehensive set of mechanisms for enabling various degrees of sharing are usually present within file systems.
- In Linux, to see permissions for the current directory contents:

```
prompt> ls -l
total 8
drwxr-xr-x 2 emad emad 4096 Apr  3 10:21 Documents
-rw-r--r-- 1 emad emad    6 Apr  3 10:14 f1.txt
```

  - 1st character distinguishes files, directories, and links
    "**-**" normal file "**d**" directory "**l**" symbolic link
  - Remaining characters are for permission levels as follows:
    "**r**" means **read** permission
    "**w**" means **write** permission (for **d**, it allows creating files within)
    "**x**" means **execute** permission (for **d**, it allows changing directory)

Files Permissions (2/2)

```
drwxr-xr-x 2 emad emad 4096 Apr  3 10:21 Documents
-rw-r--r-- 1 emad emad    6 Apr  3 10:14 f1.txt
```

Other (anyone) "o"

Group (Faculty) "g"

owner (jsmith) "u"

- If you own the file, you can change its permissions with the chmod command. Syntax:

  `chmod [user/group/others/all]=[permission] [file(s)]`

- Example:

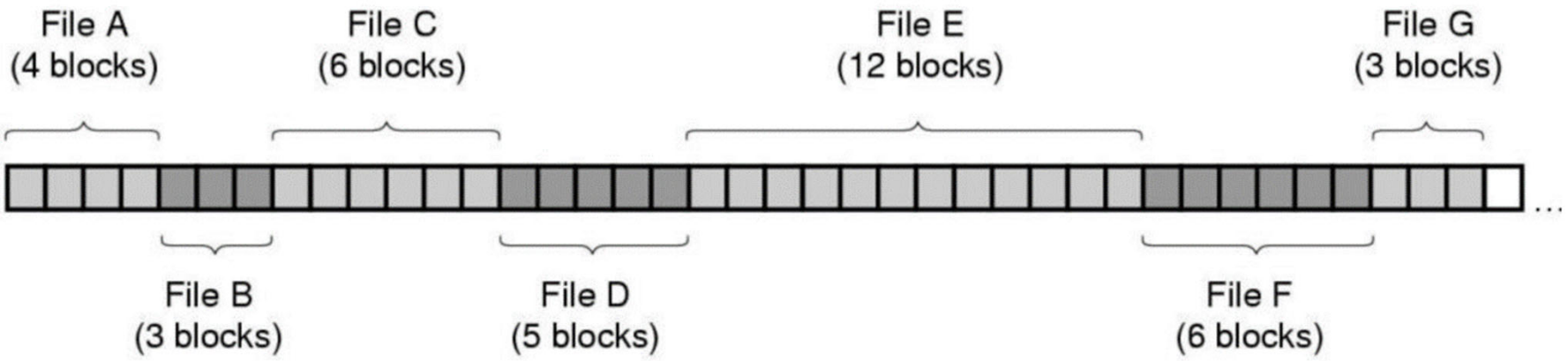
Before:	`-rw-r--r--  f1.txt`
Command:	`chmod og=rw f1.txt`
After:	`-rw-rw-rw-  f1.txt`

File System Implementation

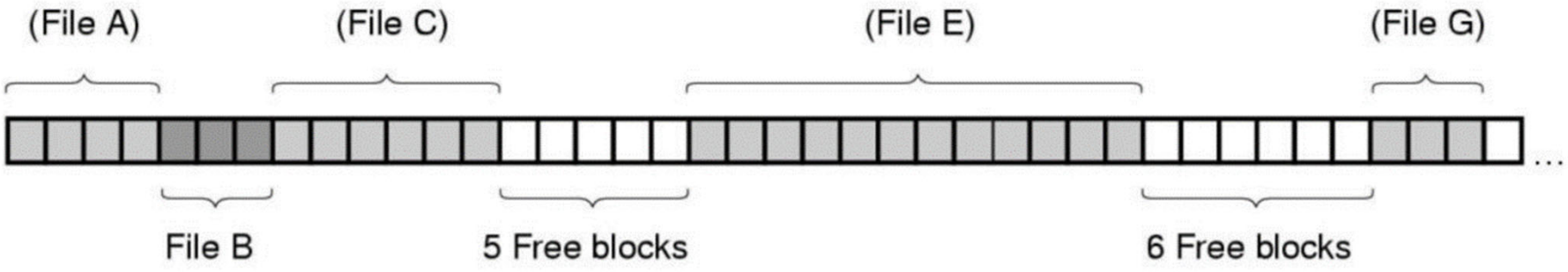
- The file system is handled **solely by software**; unlike our development of CPU and memory virtualization, we will not be adding hardware features to make some aspect of the file system work better.
- There are two aspects that make the file system work: **data structures** and **access methods**.
- The **data structures** aspect of the file system is about the types of on-disk structures (e.g., arrays, trees) that are utilized by the file system to organize its data and metadata.
- The **access methods** aspect of the file system is about the methods used to map the calls made by a process, such as `open()`, `read()`, `write()`, etc., onto its structures.
  - Which structures are read during the execution of a particular system call? Which are written? How efficiently are all these steps performed?

Contiguous Allocation Approach

- The following diagram shows a contiguous allocation of disk space for 7 files:

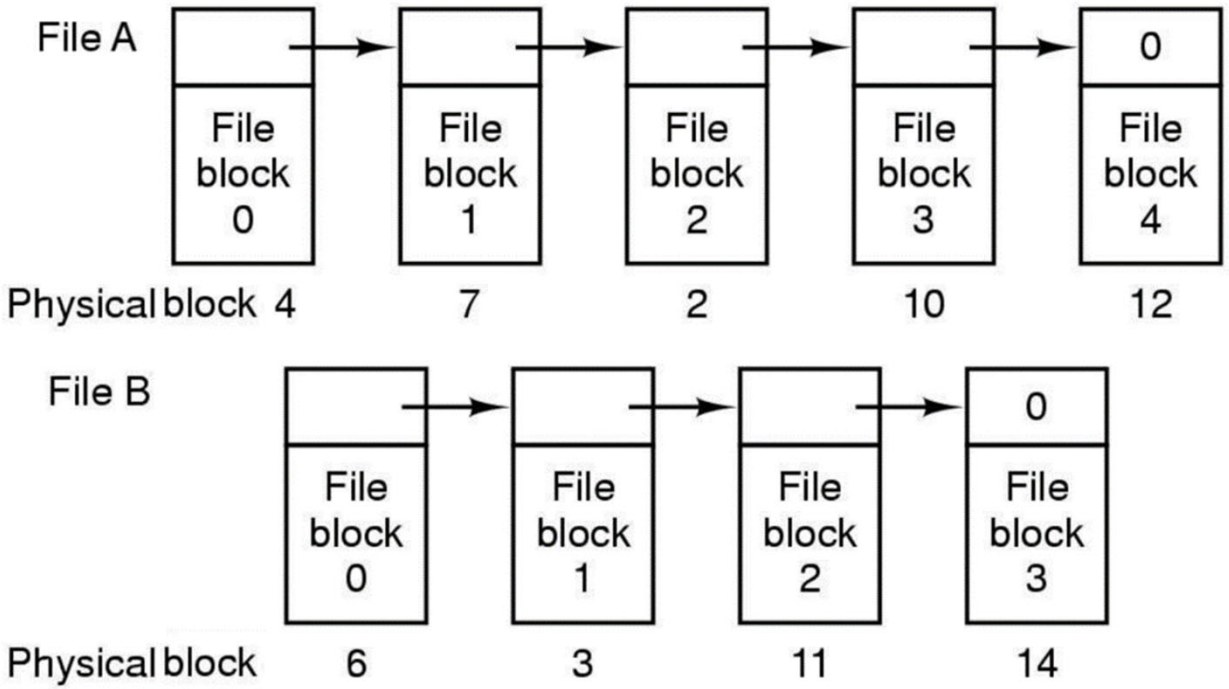


- The following is the state of the disk after files **D** and **F** have been removed. What is the problem?



Linked Allocation Approach

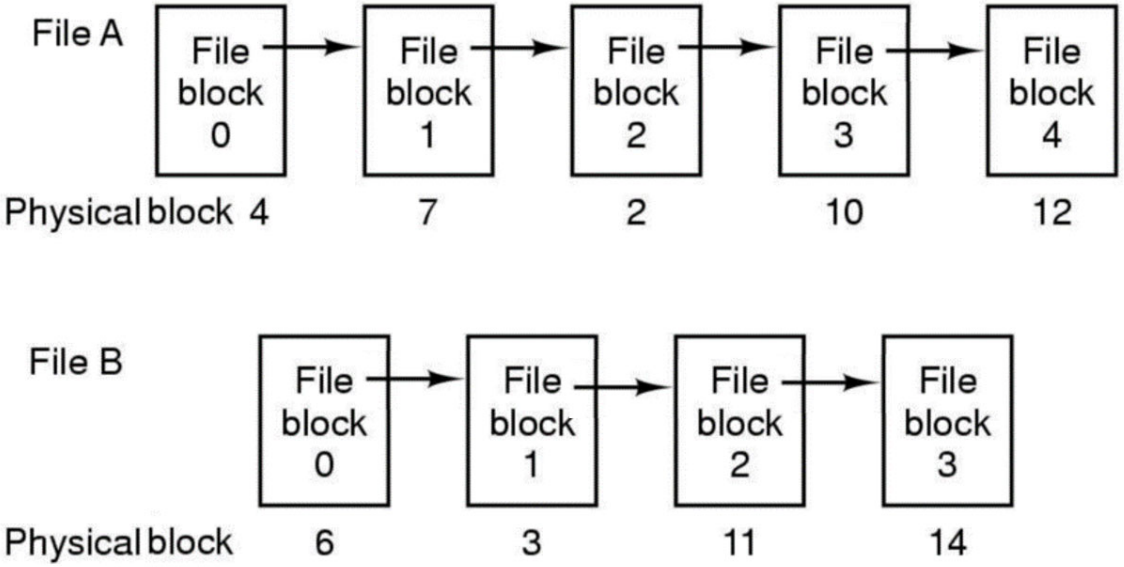
- The following diagram shows the linked list of blocks used for each file.

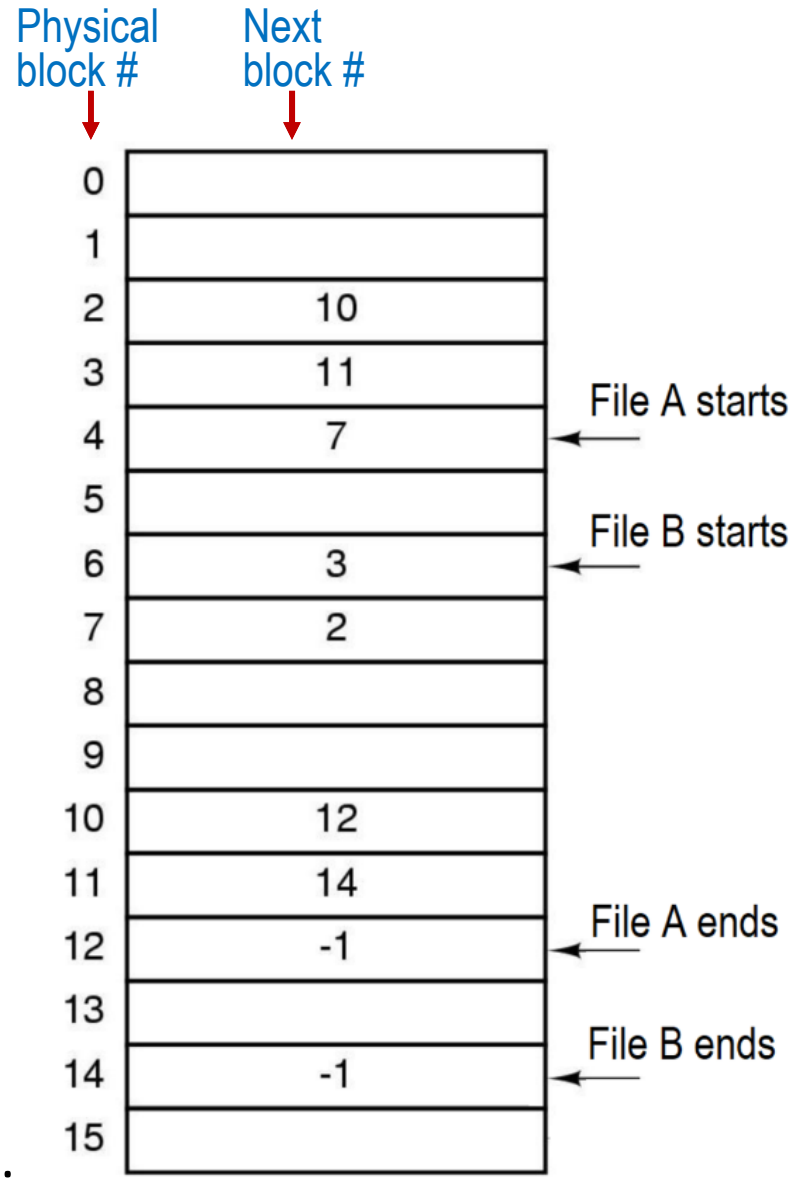


- **Free blocks** have their own linked list.
- *Advantage*: Logically contiguous blocks can be discontinuous on disk
- *Disadvantages*:
  1. Space for pointers;
  2. Random seeks are expensive. Requires list traversal from the start.

File Allocation Table Approach

- The linked list allocation is stored in a file allocation table (**FAT**) that can be moved to the RAM.

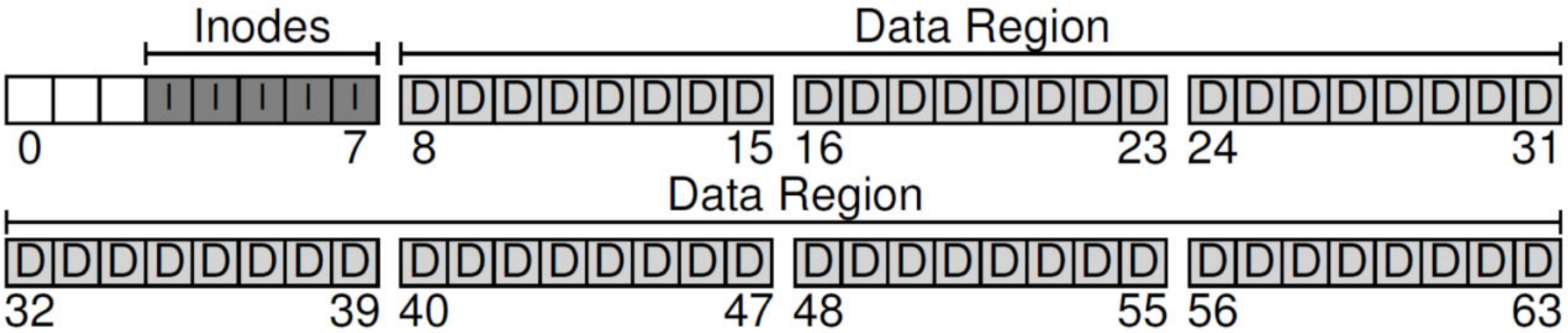




- *Advantage*: Doesn't need a separate "next" pointer within each physical block.
- *Disadvantages*: Random seeks are still expensive.

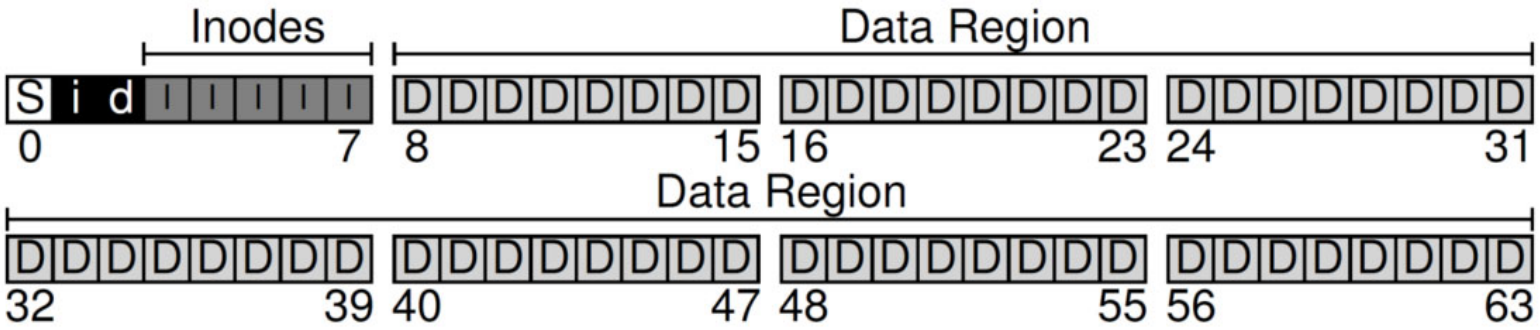
Index Nodes Approach

- Assume we need to build a simple file system on disk partition that has **64 blocks**, each of **size 4 KB**. The blocks are addressed from **0 to 63**.
- Let's call the region of the disk we use for user data "***the data region***", and, for simplicity, reserve a fixed portion of the disk for these blocks, say the last **56 blocks** (block 8 to block 63) on the disk.
- A portion in the disk is needed for the **inodes** (the structures that contain the files **metadata**, which includes the locations of the files' data blocks). Let's call this portion of the disk "***the inode table***" where 5 of our 64 blocks (blocks 3 to 7 denoted by I's in the diagram) are allocated.



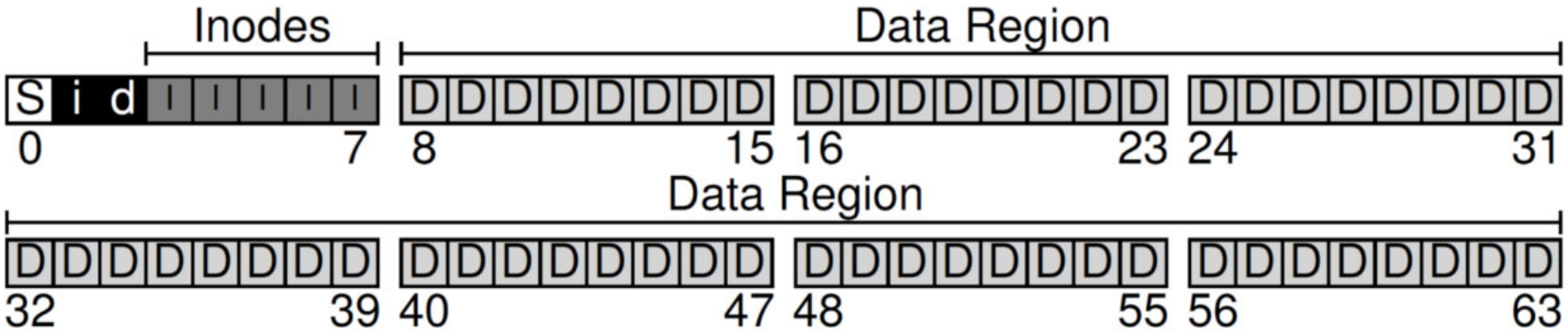
Data and inode Bitmap

- One primary component that is still needed to track whether inodes or data blocks are free or allocated. Such **allocation structures** are thus a requisite element in any file system.
  - A simple allocation structures is known as a **bitmap**, one block for the data region, the **data bitmap (d)**, and one block for the inode table, the **inode bitmap (i)**.
  - **Each bit** in the structure is used to indicate whether the corresponding object/block is free (0) or in-use (1).
  - Too many mapping bits (32K bits per block) for this simple system. This is just for simplicity.



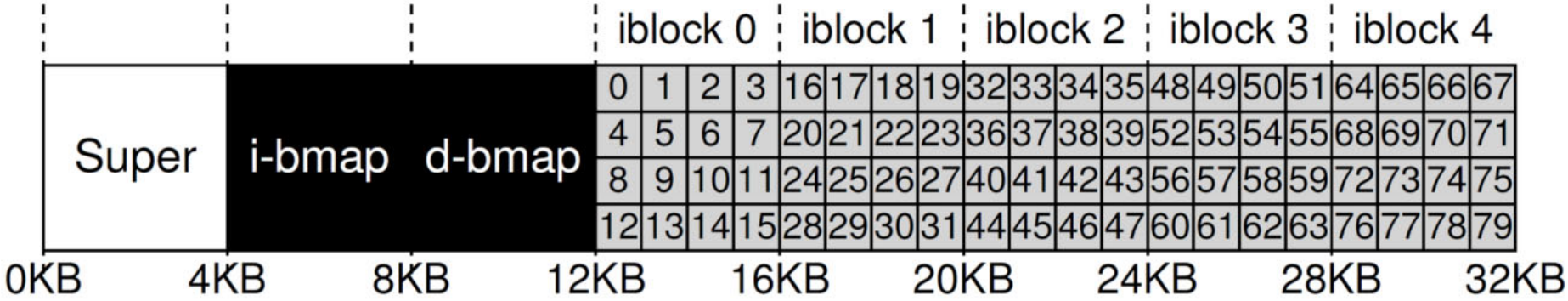
The Superblock

- The first block is the **superblock**, denoted by an **S** in the diagram. It contains information about this particular file system, such as:
  - **how many inodes and data blocks** are in the system,
  - where the **inode table begins** (block 3), and
  - a keyword to **identify the file system type** (in this case, **vsfs**).



The Inode Table

- Each **inode** is referred to by a number (the i-number), which we called the **low-level name** of the file. This number is used to calculate where on the disk the corresponding inode is located.
- The following is a close-up view of the beginning of the file system partition given the following:
  - The inode table is 20 KB (i.e., five 4KB blocks, iblock 0 to iblock 4).
  - inode is ¼KB=256 bytes → 16 inodes per block → 80 inodes per table.
  - Given the i-number (in the range 0 to 79) of an inode, the system can locate the 256-byte of the required inode. What is the maximum number of files+directories this system can have?



An inode Structure Contents

- Inside each inode (the 256 bytes) is virtually all of the information you need about a file (file **metadata**) as shown for the Ext2 system.
- The file **metadata** is the data kept by the system about each file.
  Note: the inode does not contain **the file name**!!

Size(in bytes)	**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 for 4-byte disk addresses)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Inode Direct Pointers

- To locate the file user data, an inode is to a file what a page table is to a virtual address space:
  - Page table maps **VPN** in an address space to **PFN** in DRAM
  - Inode maps **logical block numbers** (usually sequential numbers) in a file to **physical block locations** on disk using the **disk pointers** in the **`block`** field.
- A disk pointer can be a **direct pointer** where it refers to one disk block that belongs to the file. Such approach does not support big files (as the inode has only 15 such pointers).
- To support large files, **indirect pointers** are needed.

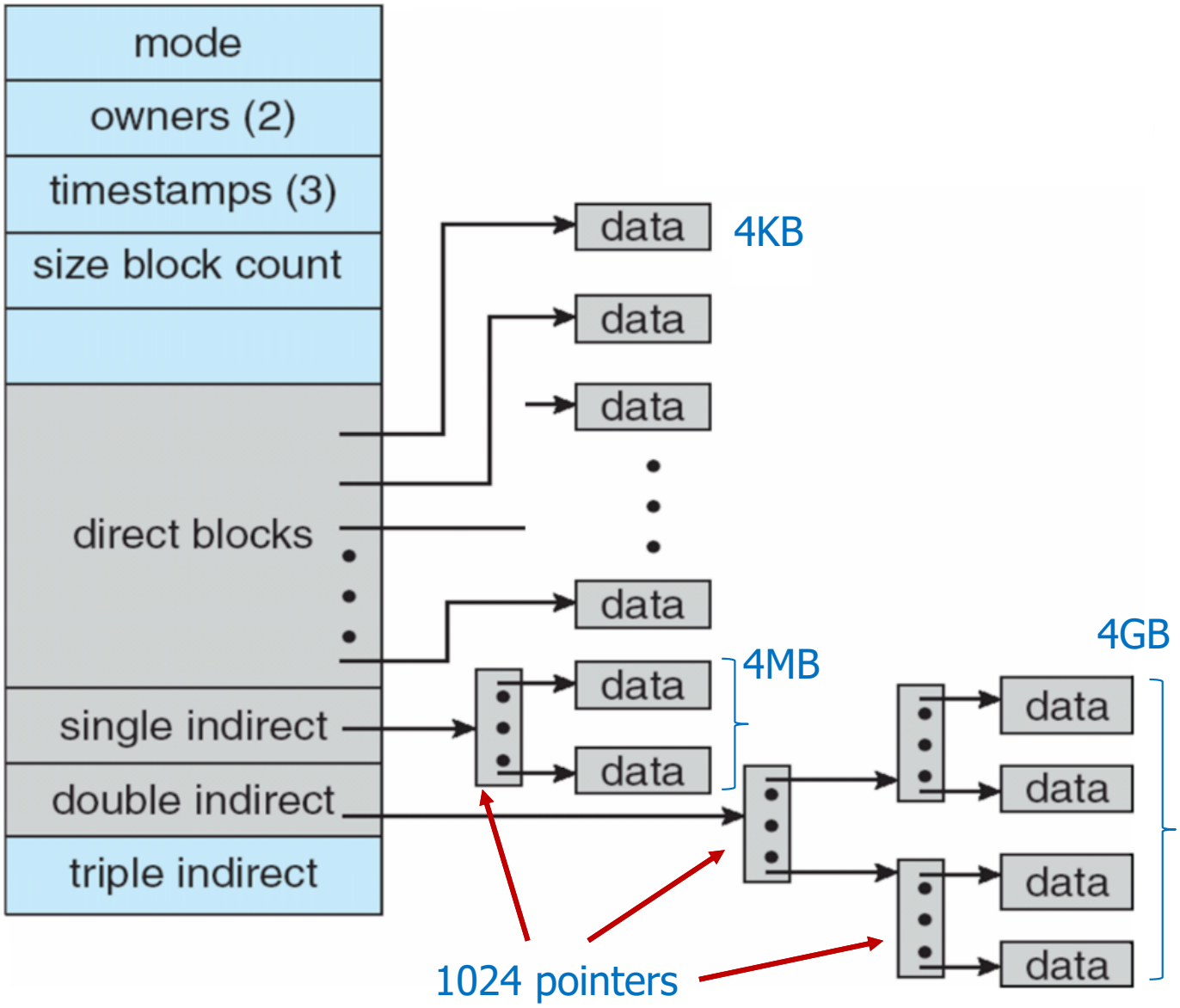
Inode Indirect Pointers

- **Indirect pointer** can be used to support bigger files . Instead of pointing to a block that contains user data, it **points to a block that contains more pointers**, each of which point to user data.
  - If a file grows large enough, an indirect block is allocated (from the data-block region of the disk), and the inode's slot for an indirect pointer is set to point to it.

- To support files that are over 4MB in size, **double,** or even **triple, indirect pointer** can be used. This approach is called the **multi-level index** approach.

Inode Pointers Example

- Each inode has a total of **15 block** pointers.
- For each block of size 4KB and a block address of 4 Bytes → A whole block can store up to **1K indirect pointers**.
- One **indirect block** points to data of size up to 1K × 4KB = 4MB
- One **double indirect** block points to data of size 1K × 4MB = 4GB
- One **triple indirect** block points to 4TB



Getting File Information

- To see the metadata for a certain file, use the `stat()` system call, which is also called by the Linux command line **`stat`**.
- `stat()` takes a pathname (or file descriptor) to a file and fill in the following `stat` structure. Each file `stat` structure is filled from the **inode** of the file.

```
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
};
```

stat Example

- The following example shows a sample run of the Linux command `stat` on a file that is just created.
- Each file system keeps this type of information in the **inode** structure.
- Recall that inode is a **persistent data structure** kept by the file system, which means **all inodes reside on disk**.
- A copy of active ones are usually **cached in memory** to speed up access.

Note: Blocks in the `stat` output is calculated as multiple of 512 bytes. In this case it is 4k/512=8

```
prompt> echo hello > file
prompt> stat file
  File: 'file'
  Size: 6   Blocks: 8   IO Block: 4096   regular file
Device: 811h/2065d Inode: 67158084    Links: 1
Access: (0640/-rw-r-----) Uid: (30686/remzi)
  Gid: (30686/remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

Directory Organization

- Often, file systems treat directories as **a special type of file**. Thus, a **directory has its own inode**, in the inode table (with the type field of the inode marked as "directory" instead of "regular file").
- That inode **points to the data blocks** where the directory data is stored.
- The directory data contains entries of the directory content.
- Each entry in the ext2 file system has the following structure:

```
struct ext2_dir_entry_2 {
  __u32  inode;      // Inode # of the directory entry (a file or another directory)
  __u16  rec_len;    // Directory entry length is multiple of 4 for efficiency
  __u8   name_len;   // Number of characters in name (without null characters)
  __u8   file_type;  // For example: 1 for Regular File and 2 for Directory
  char   name[EXT2_NAME_LEN];  // File name padded with null characters (\0)
                               // until rec_len is multiple of 4
};
```

Directory Data Example

- The following is an example of a **directory data**. The inode of this director is 5. It has three files in it (foo, bar, and foobar), with inode numbers 12, 13, and 24 respectively.
- Each directory has two extra entries, . "dot" for the current directory, and .. "dot-dot" for the parent directory.
- Initially `rec_len` stores the total size each entry and **is used to locate the next entry in the list**. It needs to be **multiple of 4** for efficiency and hence null characters are added to the name.
  - The first four fields in each entry requires 8 bytes (4+2+1+1).

    Therefore, `rec_len` = 8 + `name_len` + number of null characters

$inode_4$	rec_len_2	$name\_len_1$	$file\_type_1$	name
5	12	1	2	.\0\0\0
2	12	2	2	..\0\0
12	12	3	1	foo\0
13	12	3	1	bar\0
18	16	6	1	foobar\0\0

Deleting a Directory Entry

- The `rec_len` field may be interpreted as **a pointer to the next valid directory entry** as it is the offset to be added to the starting address of the **current directory** entry to get the starting address of the **next valid directory entry**.
- To delete a directory entry, it is sufficient to set its `inode` field to 0 and add its `rec_len` value to the value of the `rec_len` field of the **previous valid entry**. (this allows for **undelete** if needed)
- In the following example the entry of file `bar` is deleted, and the updates are shown in red.

$inode_4$	rec_len_2	$name\_len_1$	$file\_type_1$	name
5	12	1	2	.\0\0\0
2	12	2	2	..\0\0
12	24	3	1	foo\0
0	12	3	1	bar\0
18	16	6	1	foobar\0\0

Hard Links

- One field in the file `stat` information is `nlink_t`, which is the number of hard links (read from the inode **links count** field).
- The system call `link()` takes two arguments, a current file pathname and a new one; and creates another way to refer to the same file. The command-line program **`ln`** is used to do this as shown in the example (two linked files have the same inode number)

```
$ echo Hello > h1.txt
$ ln h1.txt h2.txt
$ ls -i
11700 h1.txt  11700 h2.txt
$ cat h2.txt
Hello
```

- `link()` simply creates another name in the directory you are creating the link to and refers it to the same inode number of the original file (as shown above it is 11700 for both). **The file is not copied in any way**; rather, you now just have two human-readable names that both refer to the same file.

Unlink

- When you create a file, you are really doing two things:
  1. You are **making a structure (the inode)** that will track virtually all relevant information about the file, including its size, where its blocks are on disk, and so forth.
  2. You are linking a **human-readable name** to that file and putting that link **into a directory**.
- To remove a file, the system call **`unlink()`** is used. In the previous example, if we remove h1.txt, we will still have access to h2.txt as shown:

```
$ rm h1.txt
$ cat h2.txt
Hello
```

- When the file system unlinks file, it **decrements the link count** value in the **inode structure** and, from its directory, **it removes the "link"** between the human-readable name to the given inode number.
- Only when the **link count reaches zero** does the file system also free the inode and related data blocks, and thus truly "delete" the file.

Symbolic (Soft) Links

- Hard links are limited as **you can't hard link to files in other disk partitions** (because inode numbers are only unique within a particular file system, not across file systems)
- A symbolic, or soft, link **is itself a file** whose content is a **reference** to another file or directory in the form of an absolute or relative path.
- To create a soft link, use the same program `ln`, but with the `-s` flag as shown in the example below.
- A file is denoted as a symbolic link through a dedicated bit in the **mode** field of its inode structure. Running `ls` reveals this fact as shown below (first character is `l`, not `-` as for regular files).

```
$ echo Hello > h.txt
$ ln -s h.txt hs.txt
$ ls -l
-rw-r--r-- 1 emad emad 6 Apr  2 17:57 h.txt
lrwxrwxrwx 1 emad emad 5 Apr  2 18:00 hs.txt -> h.txt
$ cat hs.txt
Hello
```

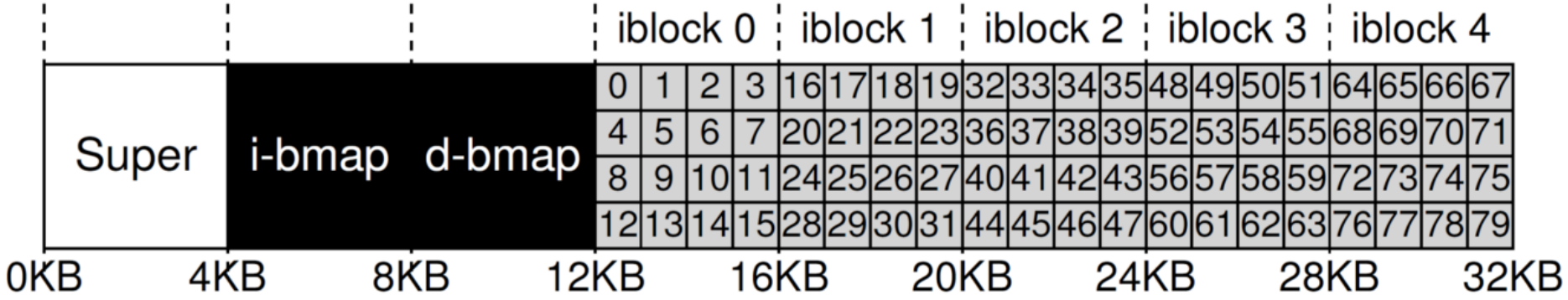
Dangling Symlink

- A dangling symlink occurs when the file system object pointed to by it is deleted, while the symlink itself still exists.
- As you see in the following example the files have different inode numbers and after deleting the original file, the symlink link file is left dangling.

```
$ls -i
11700 h.txt  11699 hs.txt
$rm h.txt
$ls -l
total 0
lrwxrwxrwx 1 emad emad 5 Apr  2 18:00 hs.txt -> h.txt
$cat hs.txt
cat: hs.txt: No such file or directory
```

Free Space Management

- The **free space management** is important for all file systems. A file system must track which inodes and data blocks are free so that when a new file or directory is allocated, it can find space for it.
- First the file system will search through the **i-bmap** for an inode that is free and allocate it to the file or directory; the file system will have to mark the inode bitmap as used (with a 1).
- A similar set of activities take place when a data block is allocated and this time with the **d-bmap**.

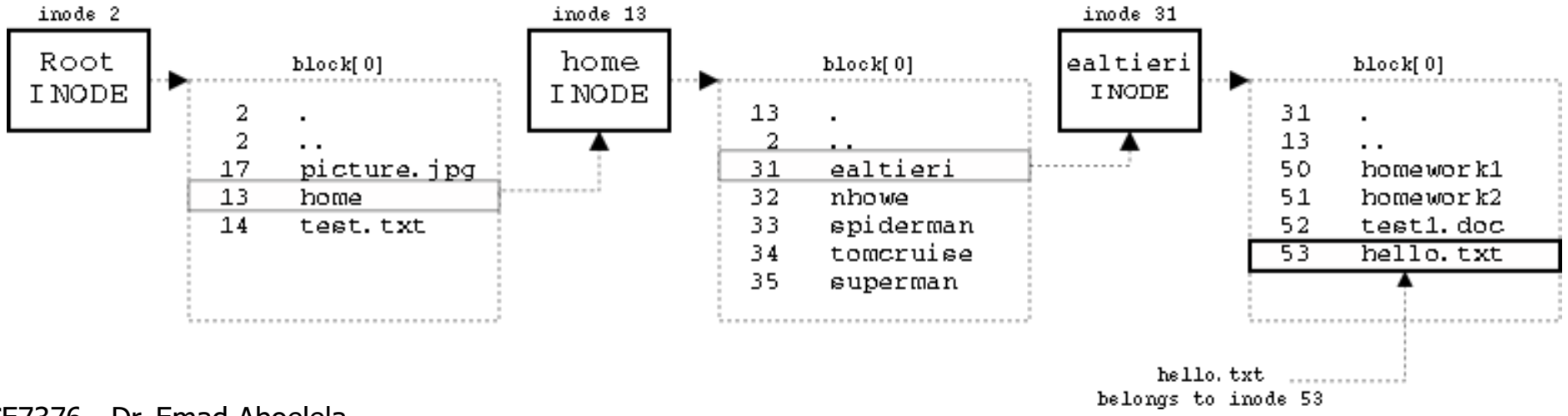


Locating a File

- Locating the data blocks belonging to a file implies locating its inode in the inode table first. The inode of the desired file is generally not known at the time the open operation is issued. What we know is the path of the file as in the following example:

```
int fd = open("/home/ealtieri/hello.txt", O_RDONLY);
```

- To find out the inode belonging to the file we first need to descend through its path, starting from the root directory. **The root directory is always located at inode 2**.
- Once the inode of the file is known, the data blocks belonging to the hello.txt are specified by the inode.block[] array.



File Open and Read Tracing

- The file system (FS) follows these steps to open file **`/foo/bar`** and then read from it:

1. Read the **inode** of the **root directory /** as such inode number is "well known". In most UNIX file systems, this **number is 2**.
2. From the root inode, follow the pointer to the **root data block**, which contains information about the root directory content. From this content, find the **inode number** of directory **`foo`**.
3. From `foo` inode, follow the pointer to `foo` data block, which contains information about the `foo` directory content. From this content, find the **inode number** of file **`bar`**.
4. From **`bar` inode**, the FS does a final **permissions check**, allocates a **file descriptor** for this process in the **per-process open-file table**, and returns it to the user.
5. Finally, the FS **reads bar data blocks** and update the inode with a new last **accessed time**.
6. The file descriptor should be de-allocated when the file is closed.

File Read Tracing Example (1/2)

- Consider the following sequence of system calls, where a file named `/foo/bar` containing 12KB of data is opened with read-only access, and its contents are read 4KB at a time.

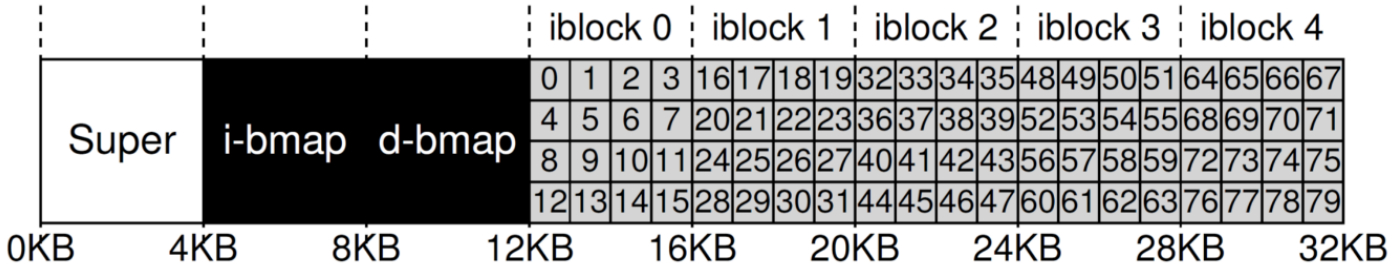
```
int fd = open("/foo/bar", O_RDONLY);
read(fd, *buffer, 4096);
read(fd, *buffer, 4096);
read(fd, *buffer, 4096);
```

- The following table summarizes all operations on disk for the above system calls.
- Each column in the table represents a data structure on disk. Time increases downwards on the table.
- Notice that the data bitmap and inode bitmap are not accessed at all during read operations

File Read Tracing Example (2/2)

/foo/bar	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data 1	bar data 2	bar data 3
open()			read	read	read	read	read			
First read()					read write			read		
Second read()					read write				read	
Third read()					read write					read

Why is this column empty?



File Create/Write Tracing

- Writing to a file is a similar process like read. First, the file must be opened. Then, the application can issue `write()` calls to update the file with new contents. Finally, the file is closed.
- Unlike reading, **writing to the file** may also **allocate** a block (unless the block is being overwritten, for example).
- When writing out a new file, each write not only has to write data to disk but also must perform the following steps:

1. Read the data bitmap and mark the newly-allocated block as used.
2. Write back the updated data bitmap to the disk.
3. Read the inode (which is updated with the new block's location).
4. Write back the updated inode.
5. Write the actual file data block itself.

- **Creating a new file** involves more I/O steps to create its inode, update the inode bitmap, allocate space within the directory containing the new file, and update the directory inode.

File Create/Write Tracing Example (1/2)

- Consider the following sequence of system calls. The `open()` system call opens file `/foo/bar` with write-only access. Flag `O_CREAT` specifies that the file should be created if it does not exist. The set of `write()` system calls write 12KB of data into the file, 4KB at a time.

```
int fd = open("/foo/bar", O_CREAT | O_WRONLY);
write(fd, buffer, 4096);
write(fd, buffer, 4096);
write(fd, buffer, 4096);
```

- The following table summarizes all operations on disk for the above system calls. Each column in the table represents a data structure on disk. Time increases downwards on the table.

File Create/Write Tracing Example (2/2)

/foo/bar

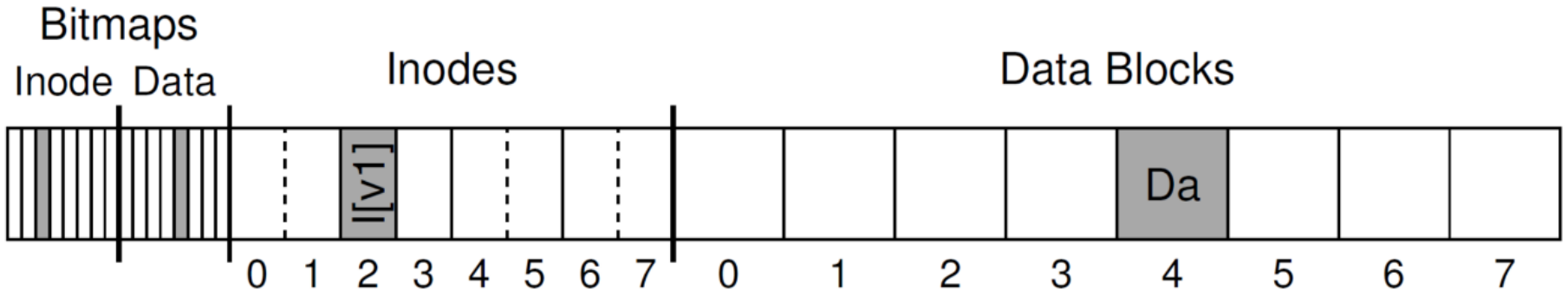
	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data 1	bar data 2	bar data 3
`open()`		read write	read	read write	write	read	read write			
First `write()`	read write				read write			write		
Second `write()`	read write				read write				write	
Third `write()`	read write				read write					write

Crash-Consistency Problem

- One major challenge faced by a file system is how to update persistent data structures despite the presence of a power loss or system crash.
- This leads to an interesting problem in file system implementation, known as the **crash-consistency problem**.
- To understand this problem, imagine that in order to complete a write operation, the system issues two requests to update on disk structures, A and B. Because the disk services only a single request at a time, one of these requests will reach the disk first (either A or B). If the system crashes or loses power after one structure update completes, the on-disk structure will be left in an **inconsistent** state.

Crash Example (1/4)

- Assume the shown tiny disk that has only 8 inodes and 8 data blocks.
  - An inode bitmap (with just 8 bits, one per inode)
  - A data bitmap (also 8 bits, one per data block).
  - Inodes (8 total, numbered 0 to 7, and spread across four blocks).
  - Data blocks (8 total, numbered 0 to 7).
- You can see that a single inode is allocated (inode number 2), which is marked in the inode bitmap, and a single allocated data block (data block 4), also marked in the data bitmap.
- The inode is denoted **I[v1]**, as it is **the first version of this inode**.



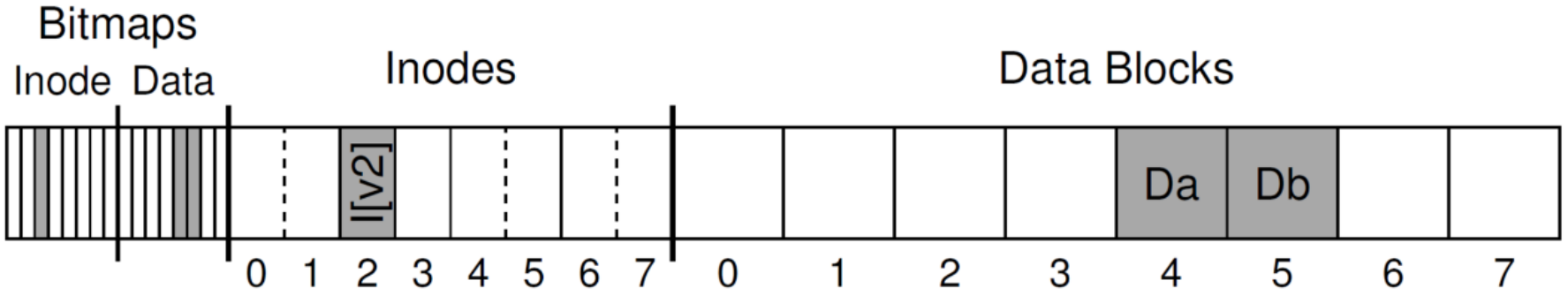
Crash Example (2/4)

- The following is a simplified content of I[v1] (left) and I[v2] (right).
- I[v2] reflects the update shown in the figure where we appended to the file a new block Db.

```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

- In addition to the inode update, we also need to **update the data bitmap** as shown.

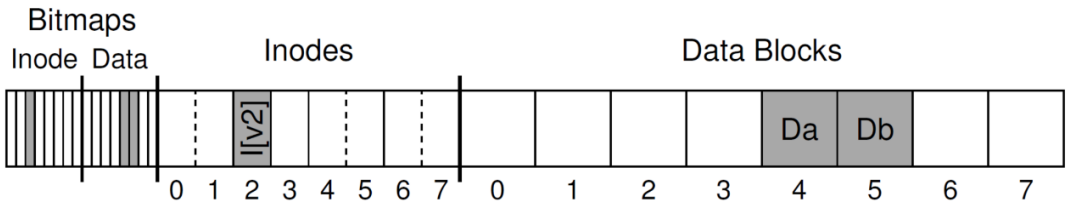


Crash Example (3/4)

- The file system must perform **three separate writes** to the disk, one each for the inode (I[v2]), bitmap (B[v2]), and data block (Db).
- These writes usually don't happen immediately when the user issues a write() system call; rather, the changes will sit in main memory (in the page cache or buffer cache) for some time first; then, when the file system finally decides to write them to disk, the file system will issue the requisite write requests to the disk.
- Unfortunately, a crash may occur and thus interfere with these updates to the disk.
- In particular, if a crash happens after one or two of these writes have taken place, but not all three, the file system could be left in an **inconsistent** state.

Crash Example (4/4)

- The following are some of the crash scenarios:
  - **Just the data block (Db) is written to disk**. In this case, the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred. This case is not a problem, from the perspective of file-system crash consistency.
  - **Just the updated inode (I[v2]) is written to disk**. In this case, the inode points to the disk address (5) where Db was about to be written, but Db has not yet been written there. Thus, if we trust that pointer, we will read garbage data from the disk (the old contents of disk address 5). The disagreement between the bitmap and the inode is an **inconsistency** in the data structures of the file system.
  - **Just the updated bitmap (B[v2]) is written to disk**. In this case, the bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus, the file system is inconsistent again; if left unresolved, this write would result in a **space leak**, as block 5 would never be used by the file system.



Solution #1: The File System Checker

- The **fsck** is a UNIX tool for finding disk structure inconsistencies and repairing them.
  - Such an approach can't fix all problems. For example, the problem where the file system looks consistent but the inode points to garbage data.
- fsck tasks:
  - **Superblock**: fsck first does sanity checks such as making sure the file system size is greater than the number of blocks that have been allocated. The goal of these sanity checks is to find a suspect (corrupt) superblock; in this case, the system (or administrator) may decide to use an alternate copy of the superblock.
  - **Free blocks**: fsck scans the inodes to produce a correct version of the allocation in both the data and inodes bitmaps; thus, if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes.
  - **Duplicates**: fsck also checks for duplicate pointers where two different inodes refer to the same block. If one inode is obviously bad, it may be cleared.

fsck Tasks (Cont'd)

- **Bad blocks**: A pointer is considered "bad" if it obviously points to something outside its valid range, e.g., it has an address that refers to a block greater than the partition size. In this case, fsck removes (clears) the pointer from the inode or indirect block.
- **Directory checks**: fsck does not understand the contents of user files; however, directories hold specifically formatted information created by the file system itself. Thus, fsck performs additional integrity checks on the contents of each directory, making sure that "." and ".." are the first entries, that each inode referred to in a directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy.

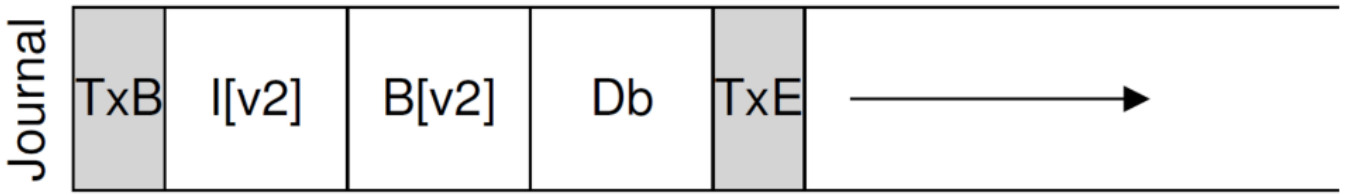
- fsck (and similar approaches) have a bigger and perhaps more fundamental problem: they are **too slow**. With a very large disk volume, scanning the entire disk to find all the allocated blocks and read the entire directory tree may take many minutes or hours

Solution #2: Journaling

- **Journaling** (or **Write-Ahead Logging**) is used by many modern file systems use the idea, including Linux ext3 and ext4, and Windows NTFS.
- Its basic idea is as follows. When updating the disk, before overwriting the structures in place, first "write ahead" a little note (somewhere that well-known location on the disk) describing what you are about to do (i.e., a log).
- By writing the note to disk, you are guaranteeing that if a crash takes places during the update (overwrite) of the structures you are updating, you can go back and look at the note you made and try again; thus, you will know exactly what to fix (and how to fix it) after a crash, **instead of having to scan the entire disk**.
- By design, journaling thus adds a bit of work during updates to greatly reduce the amount of work required during recovery.

Data Journaling Example

- Recall our simple crash example where we wish to write the inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk. Before writing them to their final disk locations, we are now first going to write them to the log (a.k.a. journal). This is what this will look like in the log:



- The log starts with **TxB** that contains the **transaction identifier** (**TID**) and something like the final addresses of the blocks I[v2], B[v2], and Db.
- In **physical logging**, the exact physical contents of the update are stored in the journal right after **TxB**.
- The final block (**TxE**) is a marker of the end of this transaction, and it also contains the **TID**.

Data Journaling Sequence (Ver 1)

- Once this transaction is safely on disk, we are ready to overwrite the old structures in the file system; this process is called **checkpointing**. Therefore, our operations sequence first version is:
  1. **Journal write**: Write the transaction, including a transaction-begin block, all pending data and metadata updates, and a transaction end block, to the log; wait for these writes to complete.
  2. **Checkpoint**: Write the pending metadata and data updates to their final locations in the file system.
- The problem with this version is if the writing to disk during the "journal write" step is not done in sequence (e.g., TxB and TxE are written, and a crash happens before writing one or more of the other journal blocks).

Data Journaling Sequence (Ver 2)

- To avoid the problem with Ver 1, the file system first writes all blocks except the TxE block to the journal. When those writes complete, the file system issues the write of the TxE block, thus leaving the journal in a safe state. Therefore, our operations sequence second version is:
  1. **Journal write**: Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete.
  2. **Journal commit**: Write the transaction commit block (containing TxE) to the log; wait for write to complete
  3. **Checkpoint**: Write the contents of the update (metadata and data) to their final on-disk locations.

Data Journaling Recovery

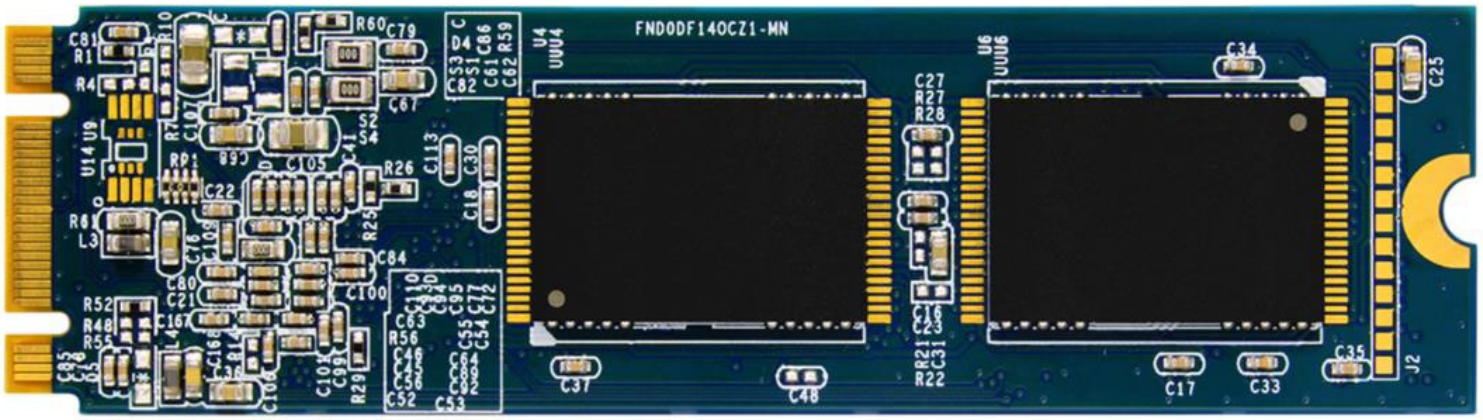
- If the crash happens after the transaction has committed to the log, but before the checkpoint is complete, the file system can recover the update as follows.
  - When the system boots, the file system recovery process will scan the log and look for transactions that have committed to the disk.
  - These transactions are thus replayed (in order), with the file system again attempting to write out the blocks in the transaction to their final on-disk locations.
  - By recovering the committed transactions in the journal, the file system ensures that the on-disk structures are consistent.

Data Journaling Sequence (Ver 3)

- After recovery is done, or if more recovery is needed, the file system needs to mark the transaction free in the journal as it is not needed any more.
- Because of the high cost of writing every data block to disk twice, some file systems **write data blocks (of regular files) to the disk first** without the need to include them in the journal log. Therefore, our operations sequence third version is:
  1. **Data write**: Write data to final location; wait for completion.
  2. **Journal metadata write**: Write the begin block and metadata to the log; wait for writes to complete.
  3. **Journal commit**: Write the transaction commit block (containing TxE) to the log; wait for the write to complete.
  4. **Checkpoint metadata**: Write the contents of the metadata update to their final locations within the file system.
  5. **Free**: Later, mark the transaction free in the journal.

Flash-Based SSDs

- **Solid-state storage device** is a persistent storage device that has recently gained significance in the world. SSDs have **no mechanical or moving parts** like hard drives; rather, they are simply built out of transistors, much like the RAM.
- However, unlike typical RAM, an SSD retains information despite power loss.



SSDs Technologies

- **NAND-based flash** is a technology used to implement SSDs. This technology has two challenges:
  1. To write to a given chunk of it (i.e., a **flash page**), you first have to erase a bigger chunk (i.e., a **flash block**).
  2. Writing too often to a page will cause it to **wear out**.

- Flash chips are organized into **banks**. Within each bank there are a number of **blocks** (typical size 128 or 256 KB); each block has a number of **pages** (typical size 4KB).

Basic Flash Operations

- **Read (a page)**: A client of the flash chip can read any page simply by specifying the page number to the device. This operation, compared to a disk, is **quicker** (**10s of microseconds**) and **uniform** (i.e., it takes same time regardless of the page location on the device and the location of the previous request). This means the device is a random-access device.
- **Erase (a block)**: Before writing to a page within a flash, the technology requires erasing the entire block the page lies within. Therefore, any important data in the block is copied elsewhere (e.g., to the RAM) before executing the erase. Once finished (**taking a few milliseconds**), the entire block is reset, and each page is ready to be programmed.
- **Program (a page)**: Once a block has been erased, the program command can be used to write the desired contents of a page to the flash. Programming a page takes around **100s of microseconds**.

Flash Reliability

- Unlike mechanical disks, which can fail for a wide variety of reasons, flash chips are pure silicon and hence have fewer reliability issues to worry about.
- The primary concern is **wearing out**; when a flash block is erased and programmed, it slowly accumulates a little bit of extra charge. Over time, as that extra charge builds up, it becomes increasingly difficult to differentiate between a 0 and a 1.
- The typical lifetime of a block is currently not well known. Some SSD are rated by manufacturers as having between 10,000 and 100,000 Program/Erase cycle lifetime. However, recent research has shown that lifetimes are much longer than expected.

SSD Performance

- The following table shows some performance data for three different SSDs and one top-of-the-line hard drive.

Device	Random Reads (MB/s)	Random Writes (MB/s)	Sequential Reads (MB/s)	Sequential Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

- SSDs outperform HDD especially in the random operations.
- SSD random read performance is not as good as SSD random write performance. The reason is due to the log-structured design of many SSDs, which **transforms random writes into sequential ones and improves performance**.

Readings and Resources List

- From Arpaci-Dusseau textbook:
  - Chapter 39: 39.17 Making And Mounting A File System
  - Chapter 44: sections 44.7, 44.8, and 44.9.