# Advanced Operating Systems: Three Easy Pieces

## 3. Persistence

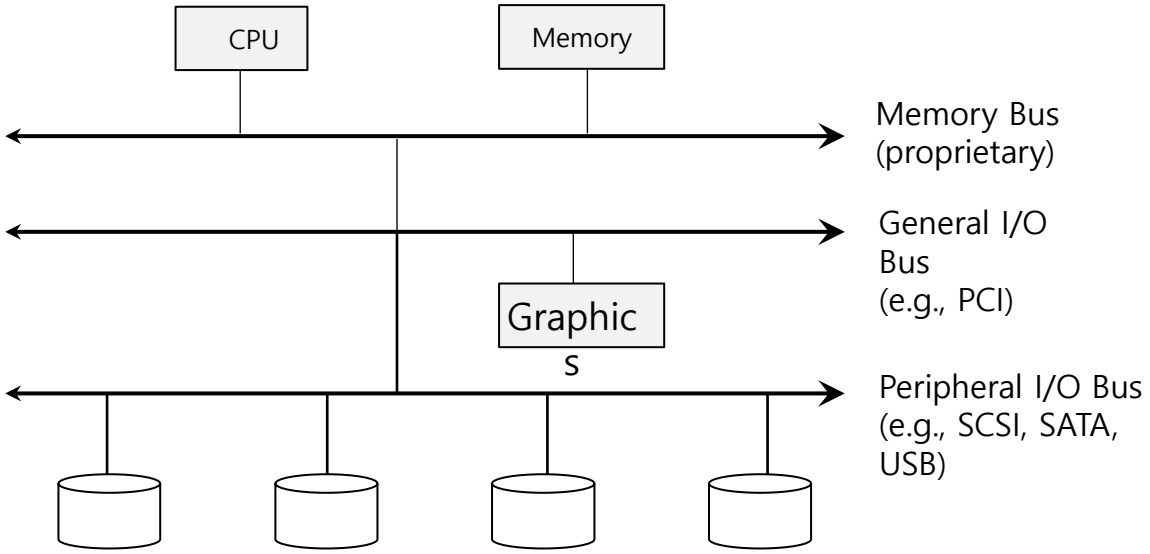
# Outline

- I/O devices
- Hard Disk Drives
- RAID
- File and Directory*
- File System Implementation
- Locality and the Fast File System*
- Cache Consistency*
- Log-structured File Systems
- Data Integrity and Protection

# Persistence:
# I/O Devices

# I/O Devices

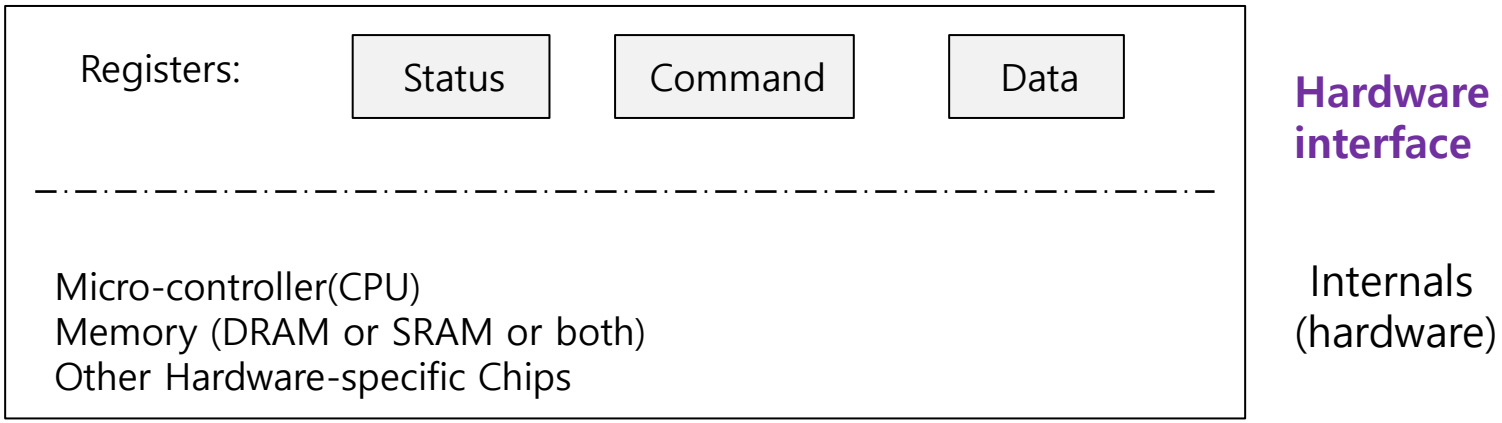
- I/O is **critical** to computer system to **interact with devices/systems**.
- **Issues:**
  - How should I/O be integrated into systems?
  - What are the general mechanisms?
  - How can we make the efficiency?




**Prototypical System Architecture**

# Canonical Device

- **Canonical Devices has two important components:**
  - **Internals** which is implementation specific.
  - **Hardware interface** allows the system software to control its operation.




**Canonical Device**

  - **Registers:** status, command, control, error and data registers control the device behavior.

# Polling

- Operating system waits until the device is ready by **repeatedly** reading the status register.
  - **Positive aspect** is simple and works.
  - **However, it wastes CPU time just waiting for the device:**
    - Switching to another ready process is better utilizing the CPU.

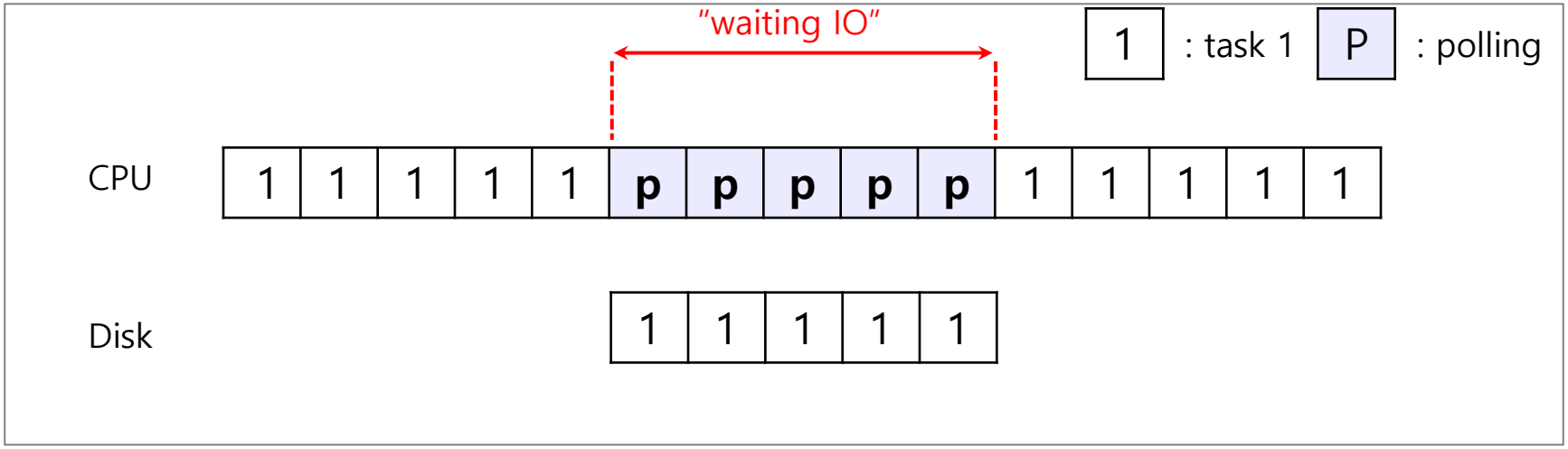



**Diagram of CPU utilization by polling**

# interrupts

- **Put the I/O request process to sleep** and context switch to another.
- When the device is finished, wake up the process waiting for the I/O by an **interrupt**.
  - Positive aspect is to allow the **CPU and the disk to be properly utilized in parallel**.

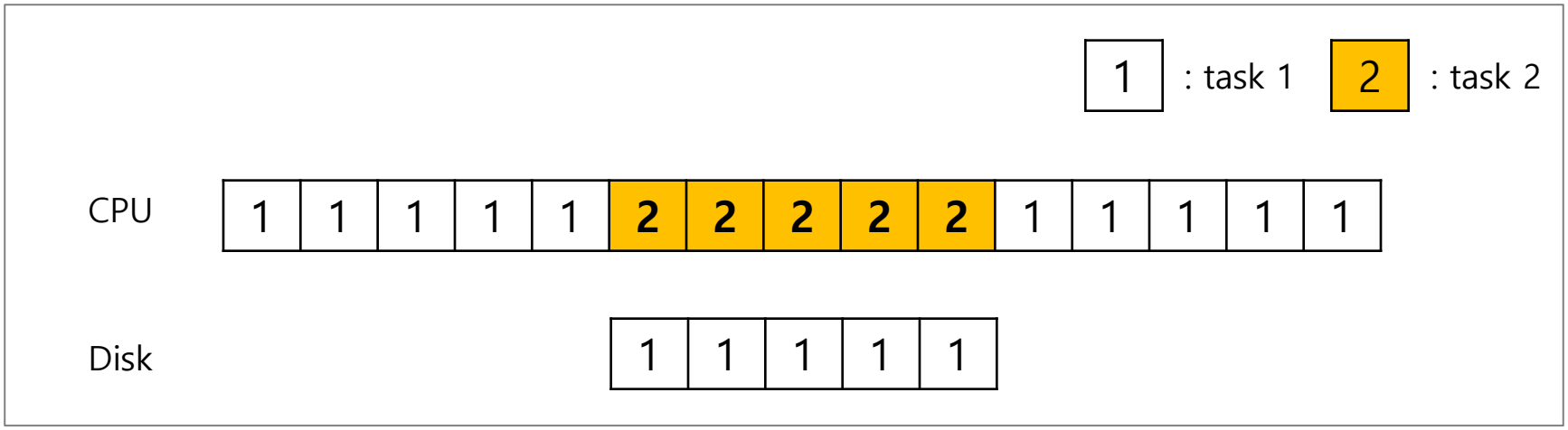



**Diagram of CPU utilization by interrupt**

# Polling vs. interrupts

- ***However***, **"interrupts is not always the best solution"**
  - **If, device performs very quickly**, interrupt will "slow down" the system (i.e., two context switch overhead).
  - **Because context switch is expensive (switching to another process)**

**If a device is fast → poll is best.**
**If it is slow → interrupts is better.**

# DMA (Direct Memory Access)

- **Write: logical copy data** from memory **to the DMA controller** by passing “where the data lives in memory, and how much data to copy”
- **Controller begins I/O to disk.** When completed, DMA raises an interrupt.

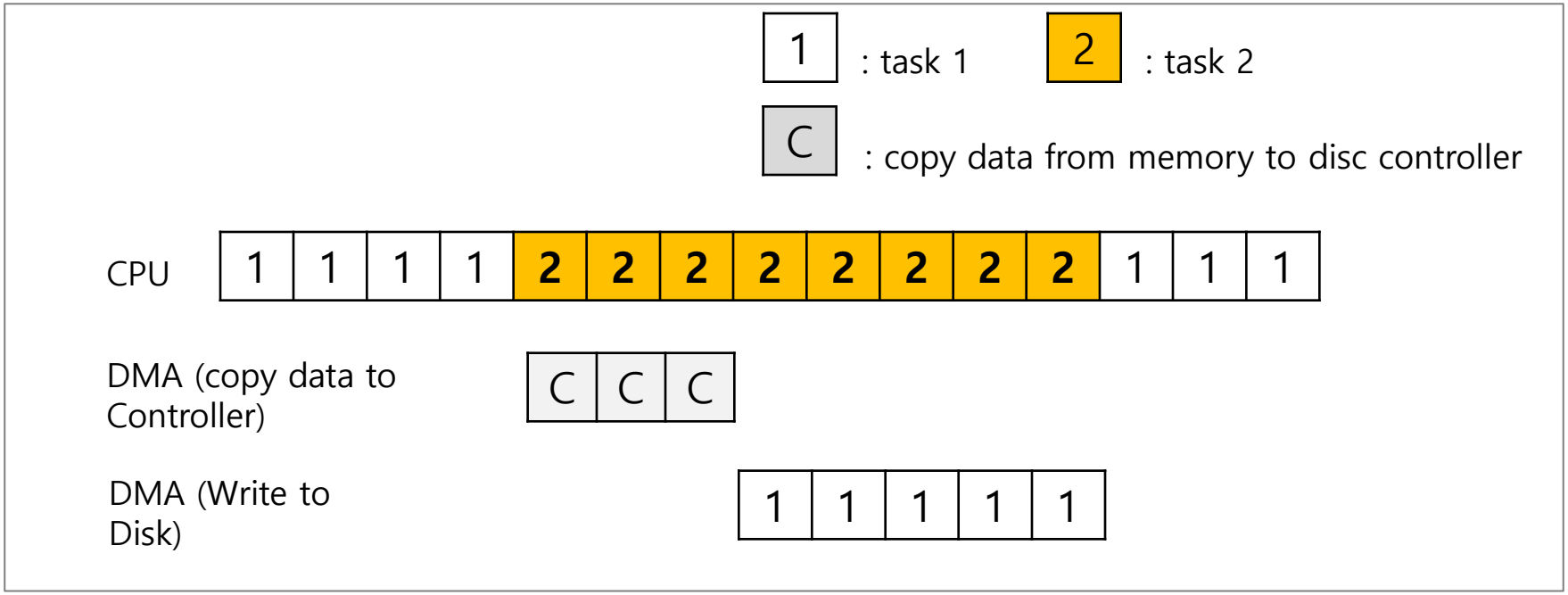


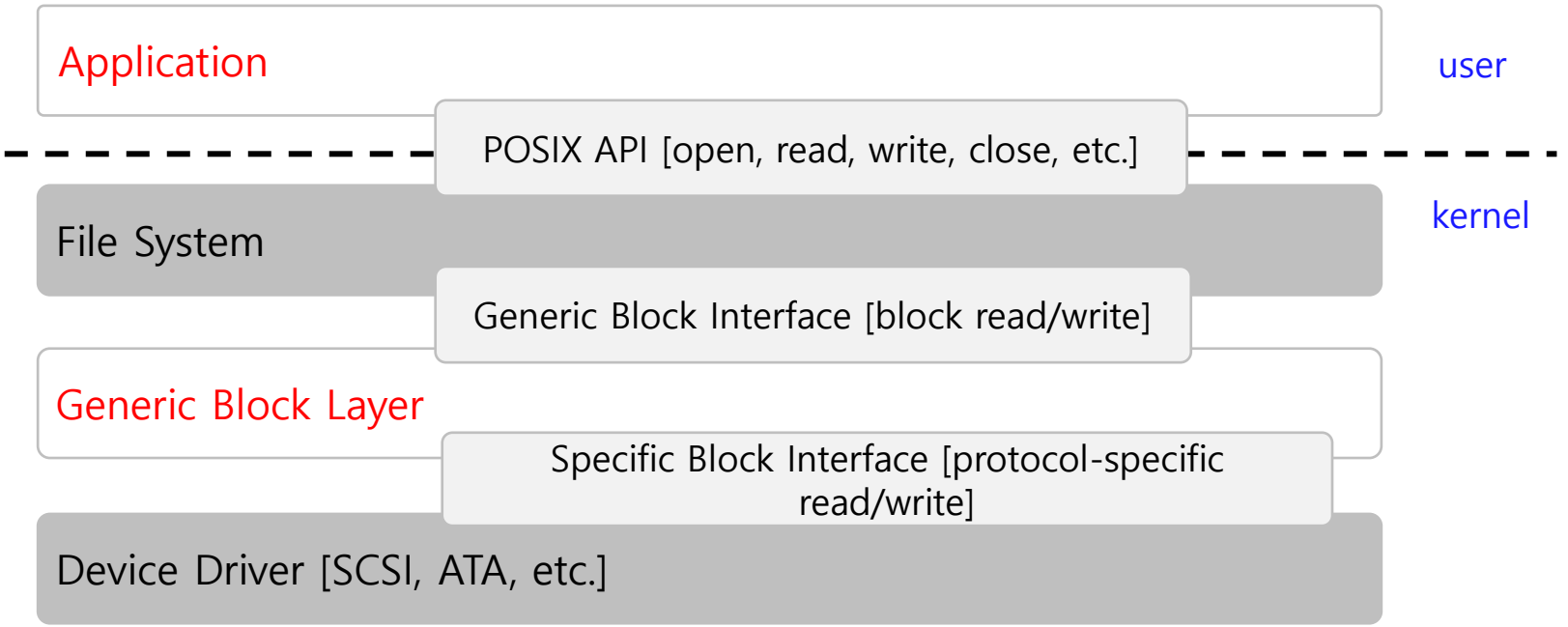

**Diagram of CPU utilization by DMA**

# Device interaction

- How the OS communicates with the **device**?
- **Solutions:**
  - **I/O instructions:** a way for the OS to send data to/from specific device registers.
    - Ex) in and out instructions on x86
  - **Memory-mapped I/O**
    - Device registers available as if they were memory locations.
    - When user R/W to the device mapped memory, the OS load (to read) or store (to write) to the device instead of main memory.
  - We'd like to build a file system that works on top of **SCSI** (Small Computer System Interface) disks, **IDE** (Integrated Drive Electronics) disks, **USB** (flash drive) keychain drivers, and so on.

# File system Abstraction

- File system **specifics** which disk class it is using.
  - Ex) It issues **block read** and **write** request to the generic block layer.




**The File System Stack**

# A Simple IDE Disk Driver

- **Five types of registers:**
  - Control, Data, command, status and error registers
  - Memory mapped IO
  - In and Out I/O instruction

- **Scenario:**
  - Wait for device to be ready
  - Write parameters to command register
  - Starts the I/O
  - Handle interrupt

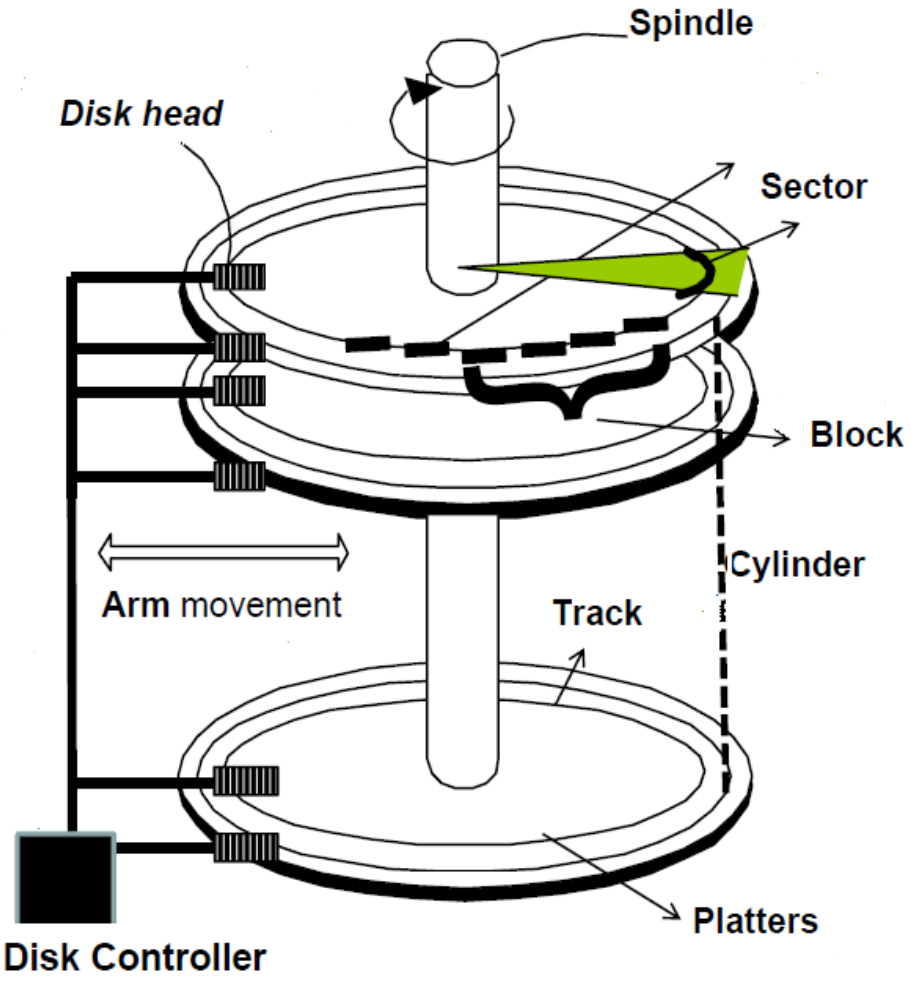
# Hard Disk Drives

# Hard Disk Driver

- Hard disk drives have been **the main form of persistent data storage** in computer systems for decades:
  - **The drive consists** of a large number of **sectors** (512-byte blocks).
  - **Address Space:**
    - We can view the disk with n sectors as an array of sectors; 0 to n-1.
  - **Terminology:** head, platter, cylinder/track, sector

# Components of a Disk

- **Platter**: several platters (base). Platter has 2 surfaces. Disks are usually 3.5" (older disks are 5.25").
- **Track**: co-centric ring on platter
- **Disk Sector:** usually 512B or 2048B for optical disks
- **Disk Block (page)**: contiguous sequence of **sectors**; unit of R/W.
  Block size = N * sector size
  Sector size ≈ 512 bytes
- **Cylinder**: set of all tracks with the same diameter
- **Disk Head**: positioned over a Block before r/w function takes place.
- **Controller**: interfaces the disk drive to the computer
- **Spindle**: used to turn the disk at steady speed (5400rpm – round per minute)



# Disk I/O Cost Components

- **Seek time:** time to move the disk head to the right track/cylinder; varies between 1 → 10 msec
- **Rotation time:** time for the head to be positioned at the beginning of the desired sector, varies between 1 → 5 msec
- **Transfer time:** time to transfer data (~10-60 µsec) either read from or written to the desired sector; around ~100-200 MB/sec
- **Key to lower I/O cost** is to reduce seek/rotation delays
- **Techniques**: Sequential scan, pre-fetching several pages, etc.

# Cache (disk Buffer in RAM)

- **Cache:** Holds in the cache the data read from disk or to be written to the disk
  - Allow the drive to quickly respond to requests.
  - Small amount of memory (usually around 8 or 16 MB)

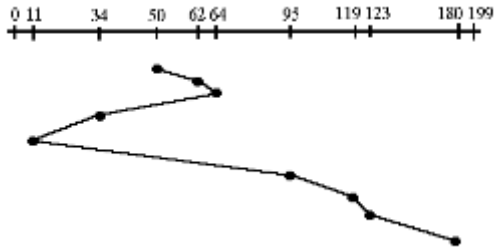
# Writeback / Delayed Write:
## Write to cache

- **Writeback** (Immediate reporting to client before data lands on disk – data in the cache):
  - **Acknowledge** a write has completed when it has **put the data only in its memory cache**.
  - **Faster** but dangerous – you might lose data in case of power failure before cache is written to disk.

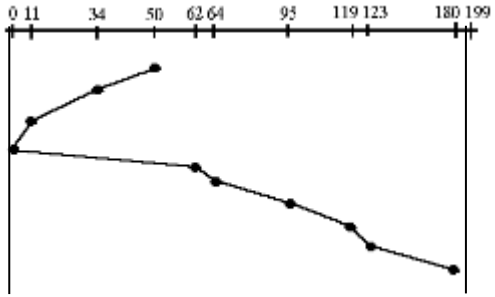
- **Write through**
  - **Acknowledge** a write has completed after the write has **actually been written to disk**.

# Disk Scheduling

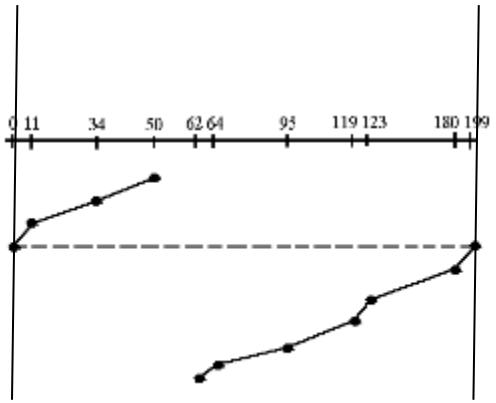
- **Disk Scheduler** decides which I/O request to schedule next:
- **SSTF** (Shortest Seek Time First):
  - Order the queue of I/O request by track/cylinder
  - Pick requests on the nearest track to complete first along the head movement.



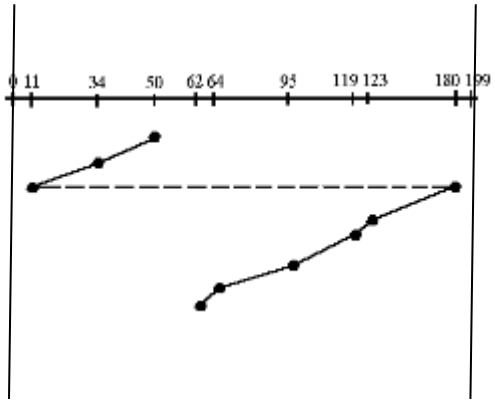

- **Elevator (SCAN):**
  - Head will move to one end then reverse direction to the other end. Always go to the two inner and outer cylinders



# Disk Scheduling




- **Circular SCAN (C-SCAN):**
  Similar to elevator but it serves requests only in one direction
- **LOOK:**
  Similar to SCAN but it does not go to the boundary cylinder unless needed to serve I/O
- **C-LOOK:**
  Similar to LOOK but it serves only in one direction



# I/O Merging (Delayed Write)

- **Reduce the number of I/O requests** sent to the disk and lowers overhead by using writeback (or delayed write):
  - E.g., read blocks 33, then 8, then 34:
    - Instead of performing 3 I/Os, the scheduler merges the request for blocks 33 and 34 *into a single two-block request and wind up performing only 2 I/Os.*

# RAID

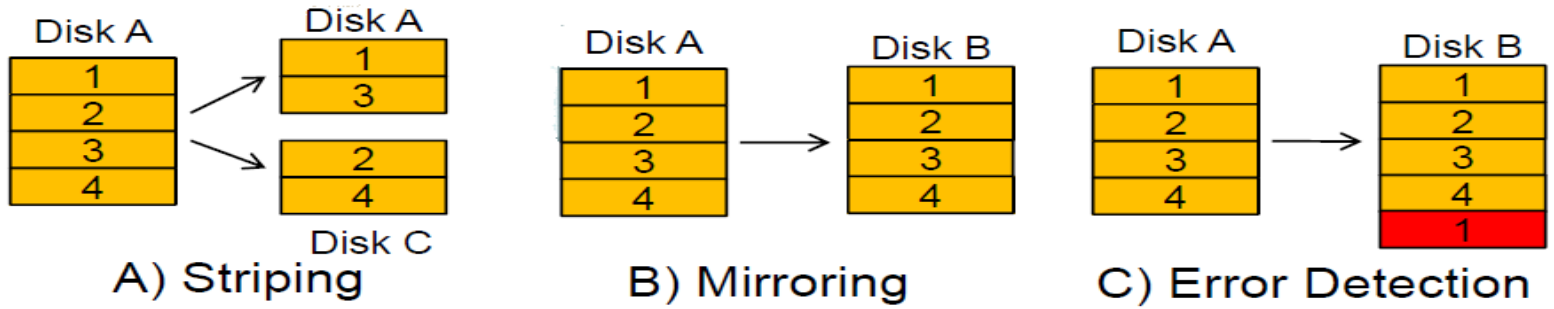
# RAID (Redundant Array of Inexpensive Disks)

- **Use multiple inexpensive disks** in concert to build a **faster**, **bigger**, and more **reliable** disk system.
  - RAID just looks like a one big disk to the host system.
- **Advantages:**
  - **Performance** & **Capacity**: Using multiple disks that can function in parallel
  - **Reliability**: RAID can tolerate the loss of a disk, i.e., replication.

**RAIDs provide these advantages transparently to systems that use them.**

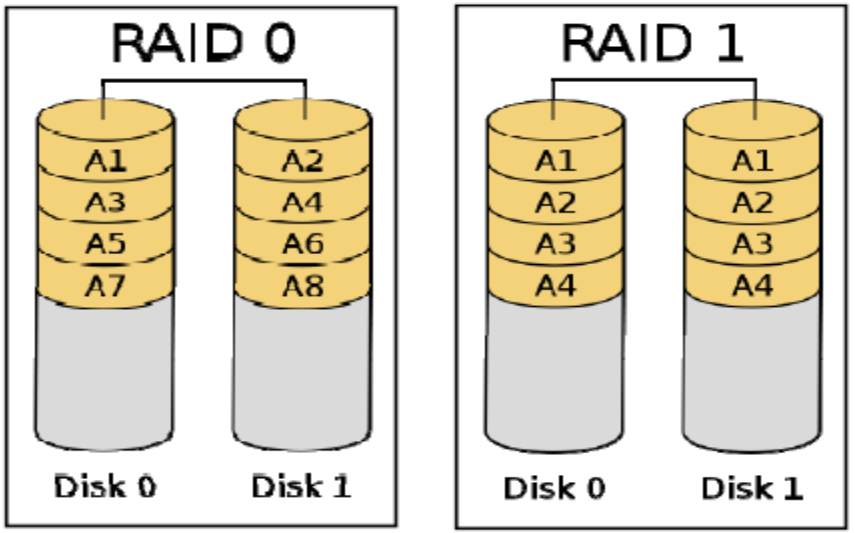
# Accessing a Disk Block

- **Disk Arrays:** arrangement of several disks that gives Abstraction of a single large disk:
  - **Increase performance:** build based on “inexpensive” and redundant components – multiple disks function in parallel
  - **Increase Reliability:** because of redundancy it can survive mechanical failures

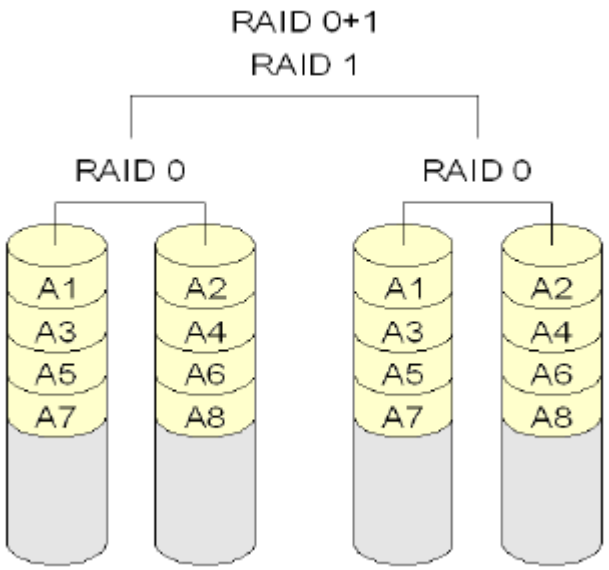


# Disk-RAID

- **RAID 0**: stripping
- **RAID 1**: Mirroring

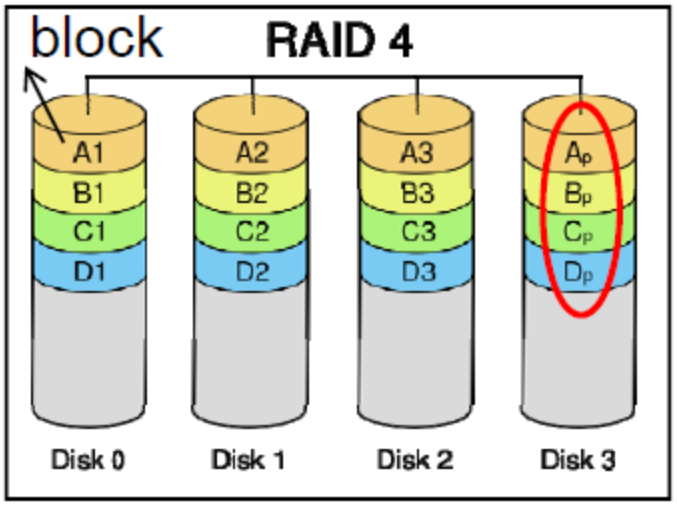



- **RAID 0 +1**: Striping
  +
  Mirroring

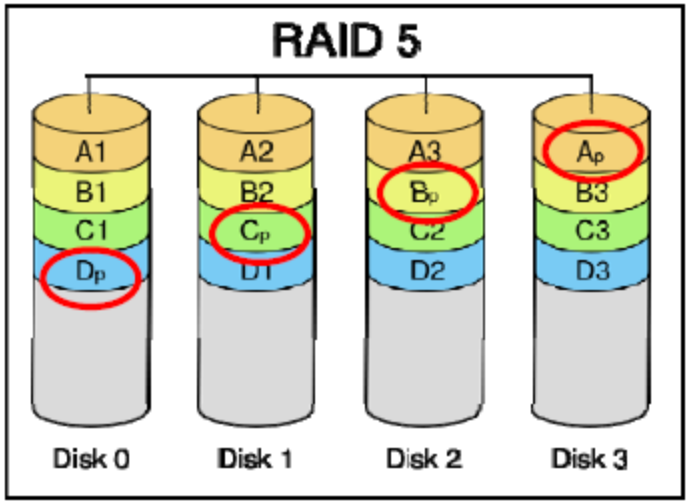


# Disk-RAID

- **RAID 4:** Block-Interleaved Parity. Designated drive for parity blocks




- **RAID 5**: Block-Interleaved Distributed Parity




- …..

# Fault Model

- **RAIDs** are designed to **detect** and **recover** from certain kinds of disk faults.

- **Fail-stop** fault model
  - **A disk can be in one of two states only:** *Working* or *Failed*.
    - **Working:** all blocks can be read or written.
    - **Failed:** the disk is permanently lost.
  - **RAID controller** can immediately observe when a disk has failed.

# RAID Comparison: A Summary

*N*: the total number of disks
*D:* the time that a request to a single disk take.

| | RAID-0 | RAID-1 | RAID-4 | RAID-5 |
|---|---|---|---|---|
| **Capacity** | N | N/2 | N-1 | N-1 |
| **Reliability (can survive # failures)** | 0 | 1 (for sure)<br>$\frac{N}{2}$ (if lucky) | 1 | 1 |
| **Throughput** | | | | |
| Sequential Read | N · S | (N/2) · S | (N-1) · S | (N-1) · S |
| Sequential Write | N · S | (N/2) · S | (N-1) · S | (N-1) · S |
| Random Read | N · R | N · R | (N-1) · R | N · R |
| Random Write | N · R | (N/2) · R | $\frac{1}{2}$R | $\frac{N}{4}$ R |
| **Latency** | | | | |
| Read | D | D | D | D |
| Write | D | D | 2D | 2D |

**RAID Capacity, Reliability, and Performance**

# RAID Comparison: A Summary

- **Care for Performance** and do not care about reliability → RAID-0 (Striping)
- **Random I/O performance** and **Reliability** → RAID-1 (Mirroring)
- **Capacity** and **Reliability (parity block)** → RAID-5
- **Sequential I/O** and Maximize **Capacity** → RAID-5

# Files and Directories

# Persistent Storage

- **Keep the data intact** even if there is a power loss.
  - Hard disk drive
  - Solid-state storage device

- **Two key abstractions** in the virtualization of storage
  - File
  - Directory

# File (Linux)

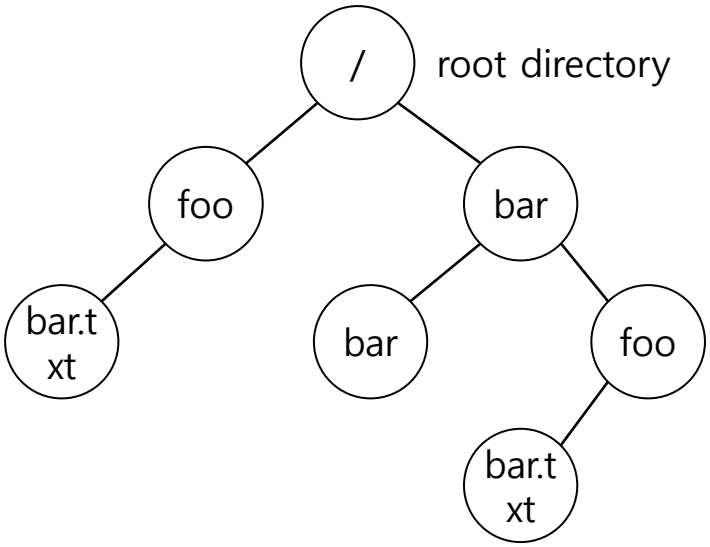
- **Linux / Unix File** is a linear array of bytes
- **Each file has 2 names**: high-level name that user uses, and low-level name (OS level) as **inode number**
  - The user is not aware of the low-level name.
- **File system** has a responsibility to store/load data persistently to/from disk.

# Directory

- **Directory is like a file**, also has a low-level name (i.e., inode).
  - **It contains** a list of **<**user-readable name (file/dir name), low-level name (inode #**>** pairs.
  - **Each entry** in a directory refers to either *files* or other *directories*.

- **Example:**
  - **A directory** has an entry (“foo”, “10”)
    - A file “foo” with the low-level name such as i-node # = “10”

# Directory Tree (Directory Hierarchy)




**An Example Directory Tree**

**Valid files (absolute pathname) :**
/foo/bar.txt
/bar/foo/bar.txt

**Valid directory :**
/
/foo
/bar
/bar/bar
/bar/foo/

Sub-directories

# Creating Files

- Use **open**() system call with O_CREAT flag:

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

  - O_CREAT : create file if it did not exist.
  - O_WRONLY : only write to that file while opened.
  - O_TRUNC : make the file size zero (remove any existing content).

- **open()** system call returns **file descriptor**.
  - *File descriptor* is a +ve integer and is used to access files.

# Reading and Writing Files

- **An Example** of reading and writing 'foo' file

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

  - **">"** : redirect the output of echo to the file foo
  - **cat:** dump the contents of a file to the screen

**How does the `cat` program access the file foo?**

**We can use `strace` (trace a command and record system calls called and signals received) to trace the system calls made by a program.**

# Reading and Writing Files (Cont.)

```
prompt> strace cat foo
#  The result of strace to figure out cat is doing.
fd = open("foo", O_RDONLY|O_LARGEFILE) = 3 // return fd = 3
# C supports char array (no strings) - "hello\n" is char[] of length = 6
read(3, "hello\n", 4096)               = 6 // read into char[] hello = 6 characters
write(1, "hello\n", 6)                 = 6 // file descriptor 1: standard out
hello
read(3, "", 4096)                      = 0 // 0: no bytes in the char[] to read into
close(3)                               = 0
…
prompt>
```

- **open**(file descriptor, flags)
  - Return file descriptor (3 in example)
  - File descriptor 0, 1, 2, is for standard input/ output/ error.
- **read**(file descriptor, buffer pointer, the size of the buffer)
  - Return the number of bytes it read
- **write**(file descriptor, buffer pointer, the size of the buffer)
  - Return the number of bytes it write

# Reading And Writing, But Not Sequentially (i.e., randomly)

- **An open file** has a **current offset**:
  - Offset determines **where** the next read or write will begin reading from or writing to within the file.

- **Update the current offset:**
  - **Implicitly**: A read or write of N bytes takes place, and N is added to the current offset.
  - **Explicitly**: lseek()

# Reading And Writing, But Not Sequentially (Cont.)

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

- **fildes**: File descriptor
- **offset**: Position the file offset to a particular location within the file
- **whence**: Determine how the seek is performed

**From the man page:**

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
the file plus offset bytes.
```

# Writing Immediately with fsync()

- **The file system** will **buffer** writes in memory (kernel buffer cache) for some time:
  - Ex) 5 seconds, or 30 seconds
  - For performance reasons

- **At that later point in time**, the write(s) will **actually be issued** (flush the buffer cache) to the storage device:
  - Write ops look to the user as completed quickly.
  - Data can be lost (e.g., the machine crashes).

# Writing Immediately with fsync() (Cont.)

- **However, some applications** require more than eventual guarantee:
  - **Ex) RDBMS** requires forced writes to disk at the time of issuing the write() system call.

- **off_t fsync(int fd)**
  - **File system forces all dirty** (i.e., not yet written) data buffers to disk for the file referred to by the file description.
  - **fsync**() returns once all of these writes are complete.

# Renaming Files

- `rename(char *old_pathname, char *new_pathname)`
  - Rename a file to different name; does not change the indoe.
  - It is implemented as an **atomic call**.
    - Ex) Change from foo to bar atomically:

```
prompt> mv foo bar        // mv uses the system call rename()
```

    - Ex) How to update a file atomically:

```
int  fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

# Getting Information About Files

- **stat**(), **fstat**(): Show the file metadata
  - **Metadata** is information about each file.
  - **Ex)** Size, Low-level name (inode #) , owner, Permission, …
  - **stat** structure is Info extracted from the i-node:

```
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 */
};
```

# Removing Files

- **rm** is Linux command to remove/delete a file
  - **rm** calls unlink() to remove a file.

```
prompt> strace rm foo
…
unlink("foo")                = 0      // return 0 upon success
…
prompt>
```

**Why it calls `unlink()`? not "`remove` or `delete`"**
**We can get the answer later (reference count)**

# Making Directories

- **mkdir():** Make a directory

```
prompt> strace mkdir foo
…
mkdir("foo", 0777)                    = 0
prompt>
```

  - When a directory is created, it is empty.
  - Empty directory has two entries: . (itself), .. (parent)

```
prompt> ls -a
./       ../
prompt> ls -al
total 8
drwxr-x---  2 remzi    6 Apr 30 16:17 ./
drwxr-x--- 26 remzi 4096 Apr 30 16:17 ../
```

# Reading Directories

- A sample code to read directory entries (like ls):

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");                    // open current directory pointer
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL)          // read one directory entry
    {
        // print out the name and inode number of each file
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp);                              // close current directory
    return 0;
}
```

  - The information available within struct dirent

```
struct dirent {
    char            d_name[256];       /* filename or subdirectory name */
    ino_t           d_ino;             /* inode number */
    off_t           d_off;             /* offset to the next dirent */
    unsigned short  d_reclen;          /* length of this record */
    unsigned char   d_type;            /* type of entry, i.e., file/dir */
}
```

# Deleting Directories

- **rmdir():** Delete a directory:
  - **Requires** that the directory be **empty**.
  - If you call rmdir() to a non-empty directory, it will fail.
    - I.e., it should have only “.” and “..” entries.

# Hard Links

- **link**(old pathname, new-one)
  - **Link** a new file name to an old/existing one
  - **Create another file** (alias) to refer to *the same existing file*
  - **The command-line link program**: ln

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2                    // create a hard link file2,
link file2 to file
prompt> cat file2
hello
```

  - Create another name in the directory that points to the same i-node as the original file, i.e., single physical copy of the file
  - Increment the reference count by 1 in the single inode

# Hard Links (Cont.)

- **Thus**, to remove a file, we call unlink().

```
prompt> ln file file2      // existing file has "hello"
prompt> rm file            // delete the original file
removed 'file'
prompt> cat file2          // Still able to access file2
hello
```

  - ***reference count***
    - **Track** how many different file names have been linked to this inode.
    - **When unlink() is called**, the reference count in the inode data structure is decremented by one.
    - **If the reference count** reaches zero, the file system frees the inode and related data blocks → truly "delete" the file

# Symbolic Links (Soft Link)

- **Symbolic link** is more **useful** than Hard link:
  - **Hard Link cannot be created** to an existing directory, i.e., to prevent linking directory to its parent directory resulting in infinite depth (i.e., "." And ".." Are pointing to the same inode).
  - **Cannot create hard link** to a file in another disk partition.
    - Because inode numbers are only unique within a file system.
- **Create a symbolic link**: ln -s

```
prompt> echo hello > file
prompt> ln -s   file  file2        /* option -s : create a symbolic link */
prompt> cat file2
Hello
Prompt> rm file    /* remove the original file; file2 is a dangling reference */
Prompt> cat file2                  /* file2 is invalid */
cat: file2: No such file or directory
```

# Symbolic Links (Cont.)

- **Dangling reference**
  - **When removing the original file**, symbolic link points to nothing (i.e., different semantics from hard link).

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file            // remove the original file
prompt> cat file2
cat: file2: No such file or directory
```

# Making and Mounting a File System

- **mkfs tool:** Make a file system

  $ mkfs -t <fs type> <device>  # -t is for type

  $ mkfs -t ext3 /dev/sda1  ← example

  - Write an empty file system, starting with *a root directory*, on to a disk partition.
  - **Input:**
    - A device (such as a disk partition, e.g., /dev/sda1)
    - A file system type (e.g., ext3)

# Making and Mounting a File System (Cont.)

- **mount():**
  - Take an existing directory as a target **mount point**.
  - Essentially paste a new file system onto the directory that I mounted the file system on. Original content of the existing directory will be no longer visible to the user.
  - **Example:**

File system type File system directory

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

    - The pathname /home/users/ now refers to the root of the newly-mounted directory.

# Making and Mounting a File System (Cont.)

- **mount program**: show **what is mounted** on a system.

```
/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)
```

  - **ext3:** A standard disk-based file system
  - **proc:** A file system for accessing information about current processes
  - **tmpfs:** A file system just for temporary files
  - **AFS:** A distributed file system (Andrew FS)

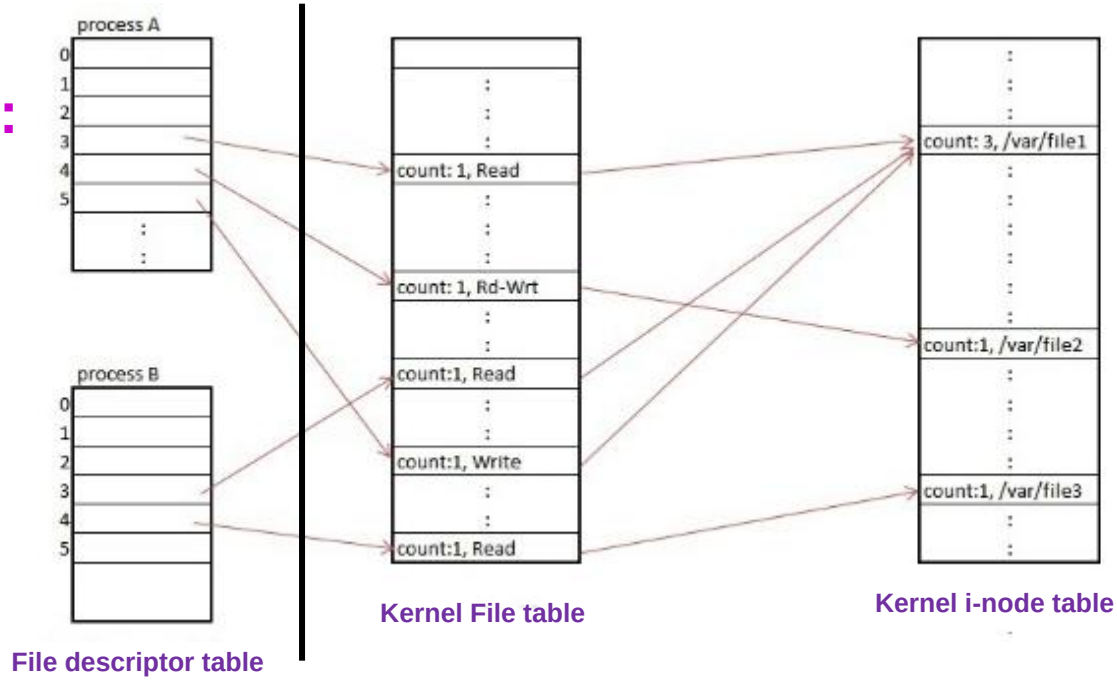
# File System Implementation

# The Way To Think

- **There are two different aspects to implement file system**
  - **Data structures**
    - What types of on-disk data structures are utilized by the file system to organize its data and metadata?
  - **Access methods**
    - How does the file system map the calls made by a process as open(), read(), write(), etc. to internal calls that access the above data structures
    - Which structures are read during the execution of a particular system call?

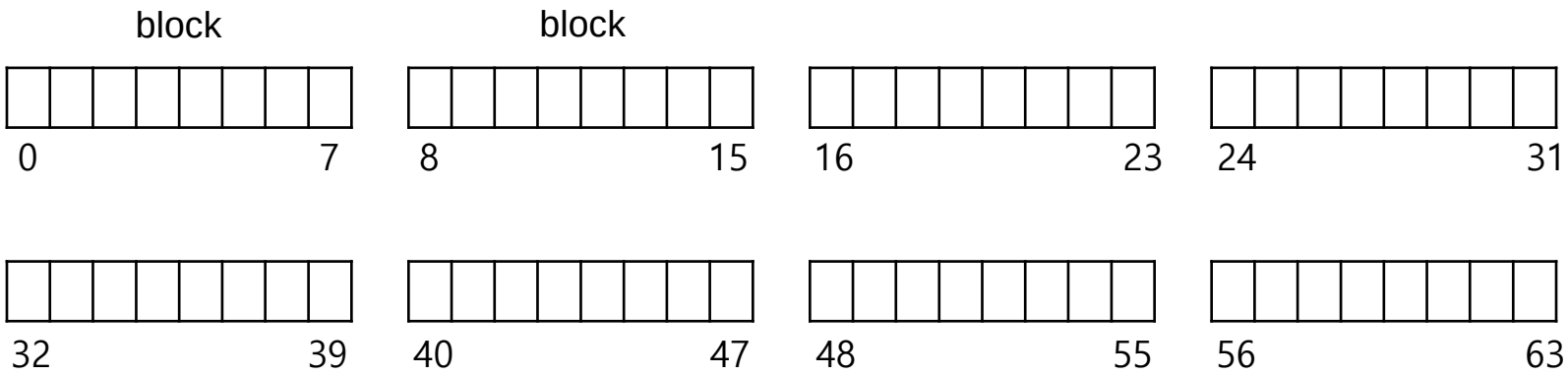
# Overall Organization

- **Let's develop** the overall organization of the file system data structure.
- **Overall Architecture:**
  - **File descriptor array** is in the process' User area. Returned FD from open() is the index in this array.
  - **Kernel File table** is shared by all processes and it allows file (offset) sharing between processes
  - **Kernel Inode table** is accessible by all processes, an inode per file



# Overall Organization (Contd.)

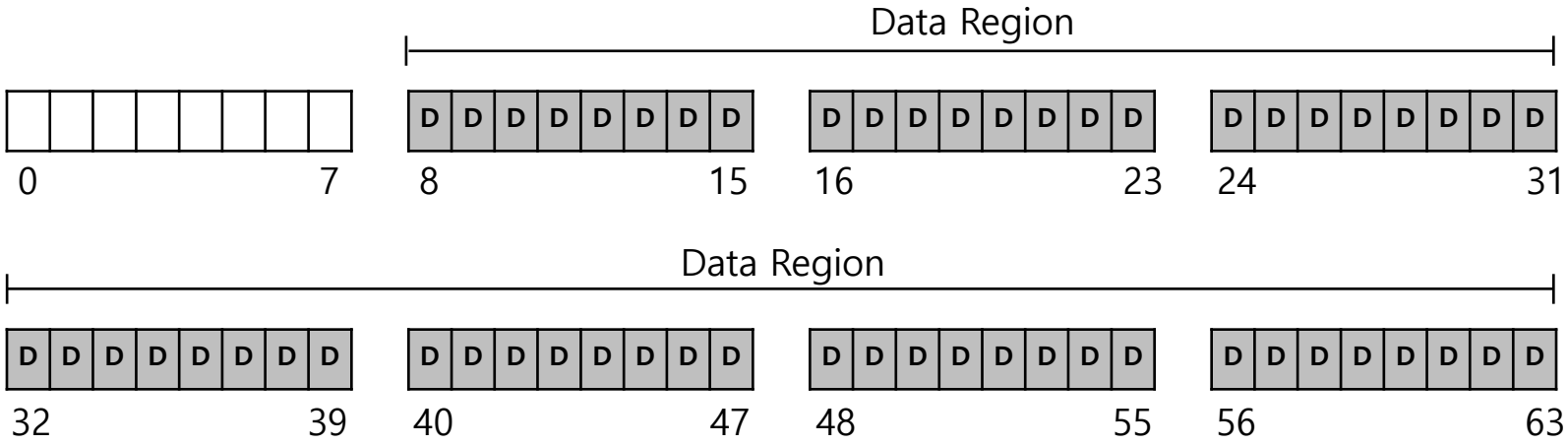
- **Disk layout:**
  - **Divide the disk** into **blocks (block is sequence of contiguous sectors and one sector = 512 bytes). Block size needs to be binary multiple of sectors**:
    - Block size is 4 KB (8 contiguous sectors).
    - The blocks are addressed from sector *0 to N -1.*




Bit-map Block Table: 0 – 63

# Data region in file system

- **Reserve data region** to store user data

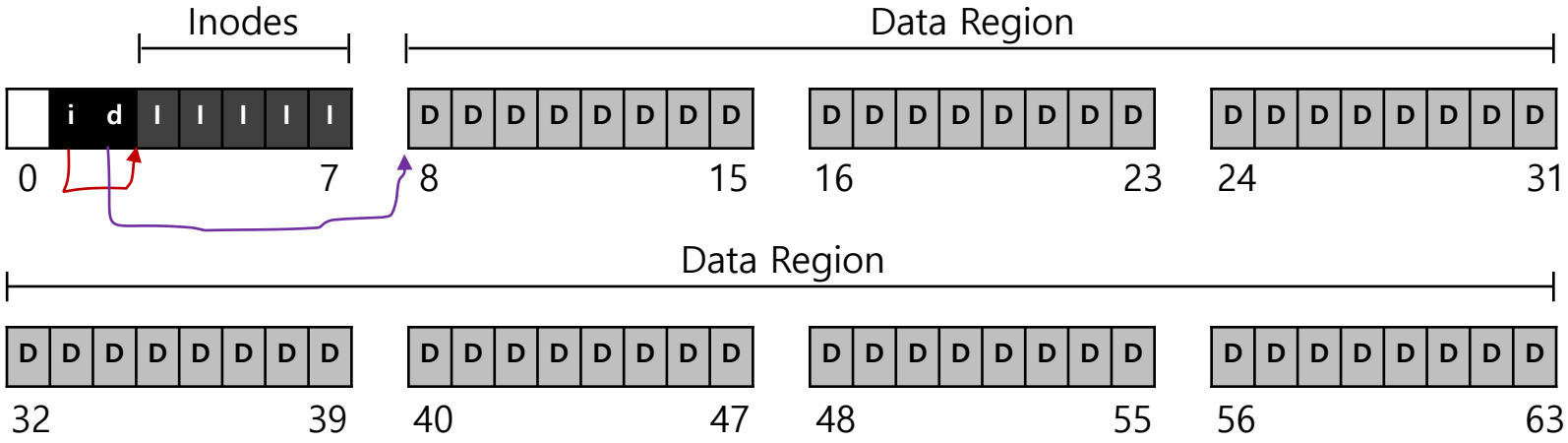



  - **File system** (i-node) has to track which data block (1 block is n sequential sectors) comprise a file, the size of the file, its owner, etc.

**How we store these inodes in file system?**

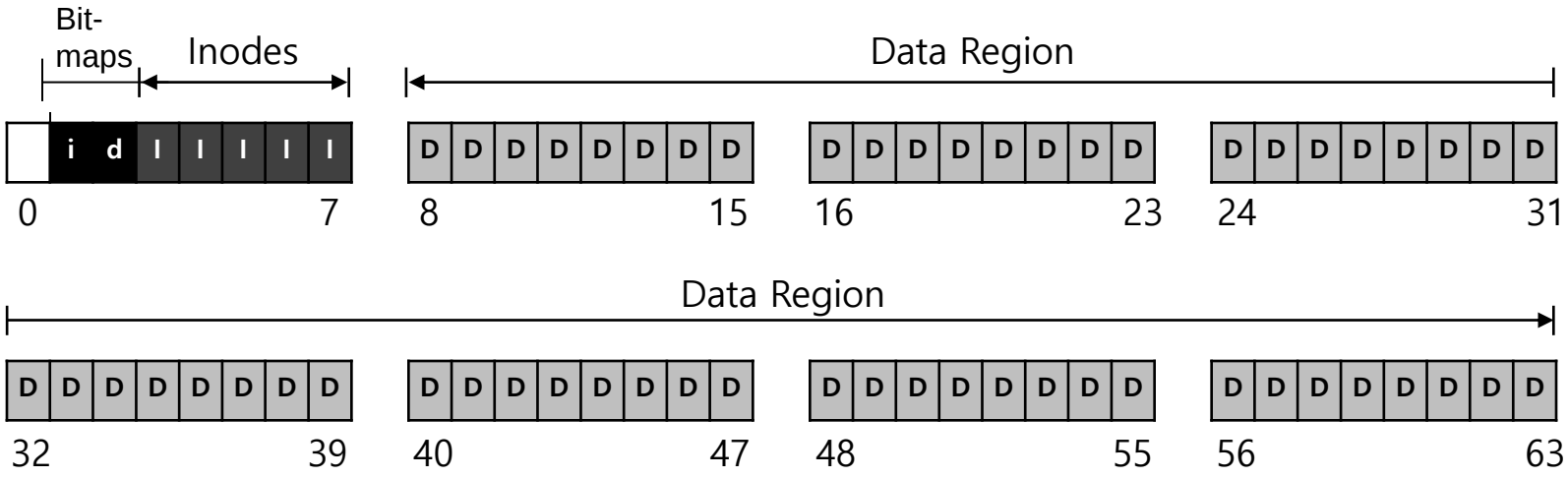
# Inode table in file system

- **Reserve some space** for **inode table:**
  - **This holds** an array of on-disk inodes.
  - Ex) **inode tables:** 3 ~ 7 blocks, inode size : 256 bytes
    - 4-KB block can hold 16 inodes.
    - The filesystem contains 80 inodes (5 blocks) - maximum number of files.



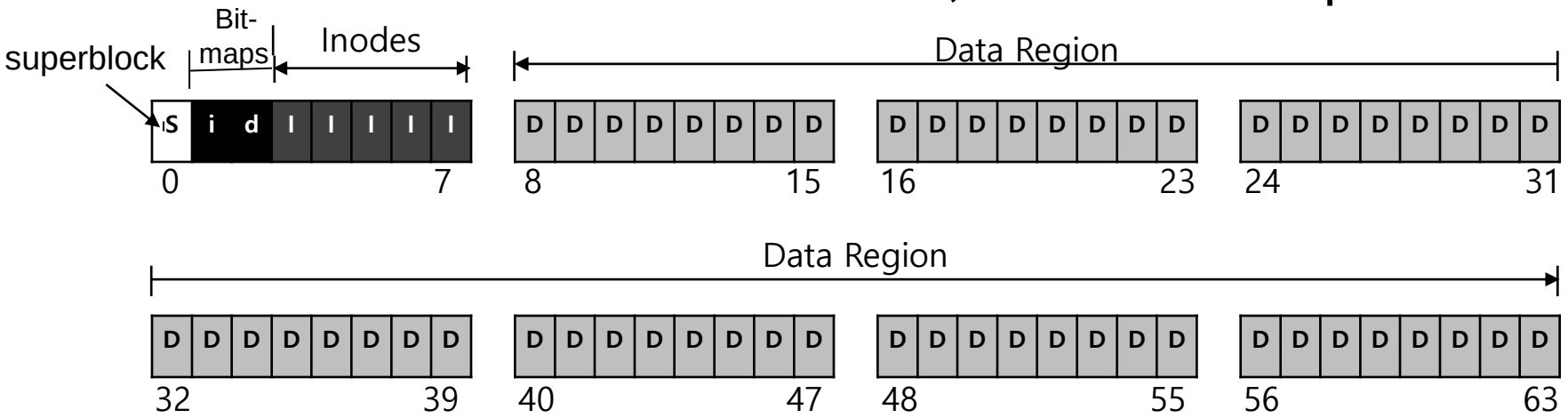
# Allocation structures

- We need to track whether inodes or data blocks are free or allocated.
- Use **bitmap**, each bit indicates free(0) or in-use(1)
  - **data bitmap**: for data blocks region
  - **inode bitmap**: for inode table



# Superblock

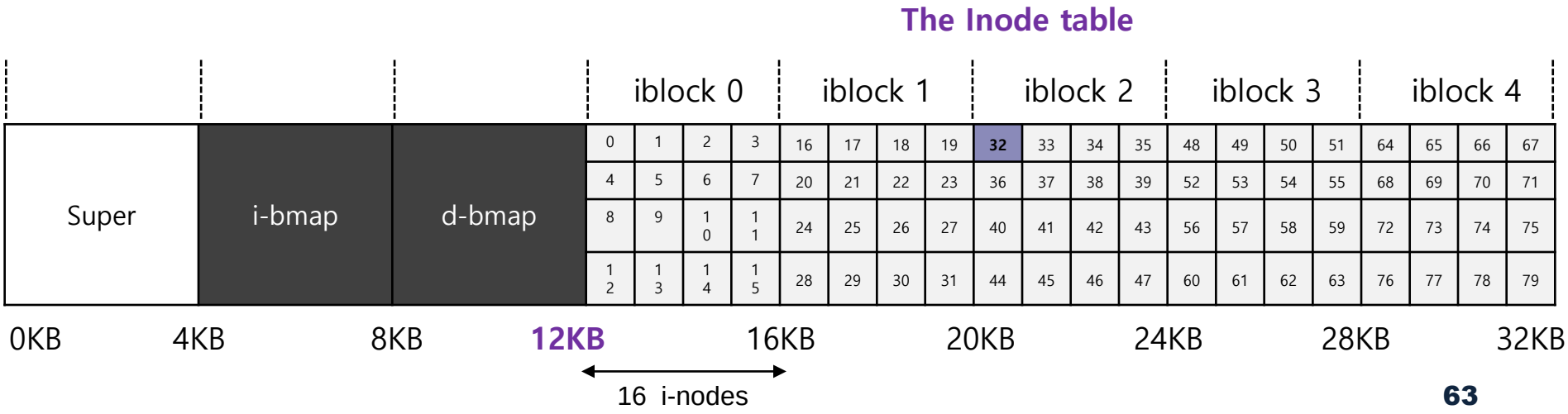
- **Super block contains** the following **information** for a **particular file system**
  - Ex) file system size, block size, # of used/free blocks, size & location of i-node table, disk block map…




  - Thus, when mounting a file system, OS will read the superblock first, to initialize various information.

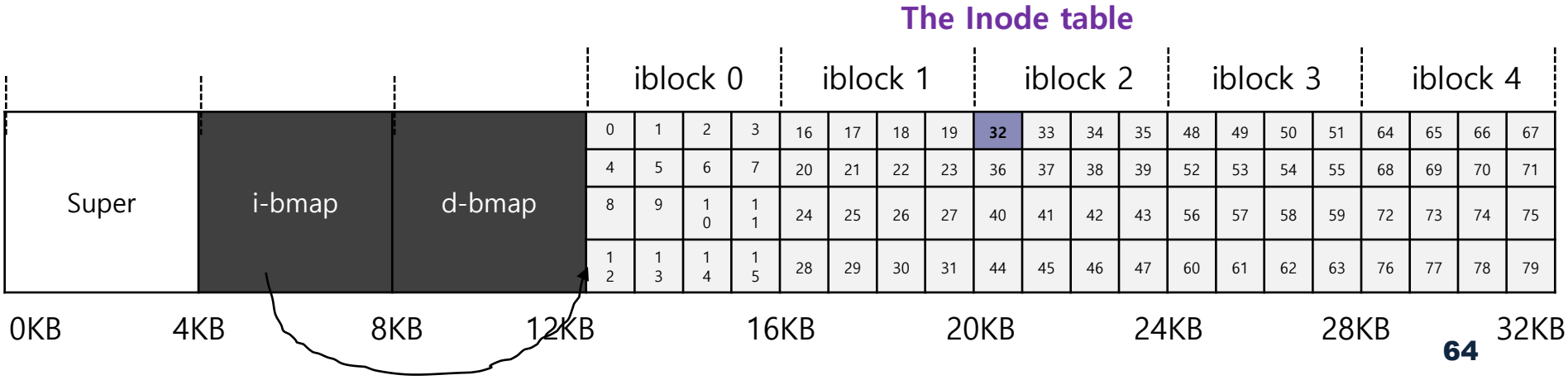
# File Organization: The inode

- **Each inode is referred to by inode number:**
  - **by inode number**, File system calculate where the inode is on the disk.
  - Ex) **inode number:** 32
    - Calculate the offset into the inode region (32 x sizeof (inode = 256 bytes) = 8,192 = 8 KB
    - Add start address of the inode table(i.e., 12 KB) + inode region offset (i.e., 8 KB) = 20 KB



# File Organization: The inode (Cont.)

- **Disk are not byte addressable**, sector addressable.
- **Disk consist of** a large number of **addressable sectors**, (512 bytes)
  - Ex) **Fetch the block of inode** (for inode number: 32)
    - **Sector address**: addr of the inode block:
    - **Block** has 16 inodes
    - **Blk # for inode:** (inumber * sizeof(inode)) / blocksize
      Blk # for inode 32 = (32 * (1/4)) / 4 = 2 ← block #
    - **Sector # for blk 2:** ((blk # * blocksize) + inodeStartAddr) / sectorsize
      Sector # = {(2 * 4) + 12} / (1/8) = 20 * 8 = 160 ← sector # for begin blk 2

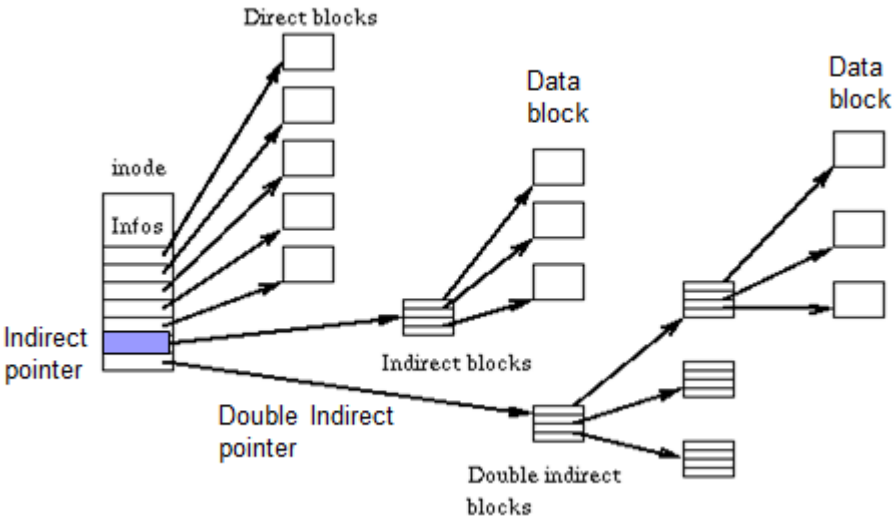


# File Organization: The inode (Cont.)

- **inode** have all of the information about a file:
  - **File type** (regular file, directory, etc.),
  - **Size**, the number of data blocks allocated to it.
  - **List of addresses** of the file's data blocks: https://en.wikipedia.org/wiki/Inode_pointer_structure
  - **Protection information** (who owns the file, who can access, etc).
  - **Time information**.
  - **Etc**.

# The Multi-Level Index

- **To support bigger files**, we use multi-level index.
- **Indirect pointer** points to a block that contains more pointers.
  - **inode have fixed** number of direct pointers (**12**) and a single indirect pointer.
  - **If a file grows** large enough, an indirect block is allocated (1024 pointers), inode's slot for an indirect pointer is set to point to it.
    - File size (1-indirect) = (12 + 1024) x 4 K = 4,144 KB ≈ 4MB
    - File size (2-indirect) = (12 + 1024 + $1024^2$) x 4K ≈ 4 GB



# The Multi-Level Index (Cont.)

- **Double indirect pointer** points to a block that contains indirect blocks:
  - Allow file to grow with an additional 1024 x 1024 or 1 million 4KB blocks.
- **Triple indirect pointer** points to a block that contains double indirect blocks.
- Multi-Level Index approach to pointing to file blocks:
  - Ex) twelve direct pointers, a single and a double indirect block.
    - over 4GB in size ← $(12+1024+1024^2)$ x 4KB
- Many file system use a multi-level index:
  - Linux EXT2, EXT3, NetApp's WAFL, Unix file system.
  - Linux EXT4 use extents instead of simple pointers.

# Directory Organization

- **Directory contains** a list of entries, each is <file name, inode number> pairs.
- **Each directory** has two extra files . ”dot” for current directory and .. ”dot-dot” for parent directory
  - **For example**, dir has three files (foo, bar, foobar)

| inum | reclen | strlen | name |
|---|---|---|---|
| 5 | 4 | 2 | . |
| 2 | 4 | 3 | .. |
| 12 | 4 | 4 | foo |
| 13 | 4 | 4 | bar |
| 24 | 8 | 7 | foobar |

**Directory (dir) on-disk**

**strlen**: length of the file name
**reclen**: length of this direct entry – must be multiple of 4 (for alignment reasons)

# Free Space Management

- **File system** tracks which inode and data block are free or not.

- **In order to manage free space, we have two simple bitmaps:**
  - **When file is newly created,** it allocates an inode by searching the inode bitmap and update on-disk bitmap.
  - **Pre-allocation policy** for data blocks is commonly used for allocating contiguous blocks for near future usage.

# Access Paths: Reading a File From Disk

- **Issue an open("/foo/bar", O_RDONLY): Traverse the pathname** and thus locate the desired indoe.
  - **Begin at the root** of the file system (/)
    - In most Unix file systems, the root inode number is **2**
  - **Filesystem reads** in the block that contains inode number 2.
  - **Look inside** of it to find pointer to data blocks (contents of the root).
  - **By reading in** one or more directory data blocks, It will find "foo" directory.
  - **Traverse recursively** the path name until the desired inode ("bar")
  - **Check final permissions**, allocate a file descriptor for this process and returns file descriptor to user.

# Access Paths: Reading a File From Disk (Cont.)

- **Issue read() to read from the file:**
  - **Read in the first block of the file**, consulting the inode to find the location of such a block:
    - Update the inode with a new last accessed time.
    - Update in-memory open file table for file descriptor, and the file offset.
- **When file is closed:**
  - **File descriptor** should be deallocated, but for now, that is all the file system really needs to do. No disk I/Os take place (i.e., delayed write).

# Caching and Buffering

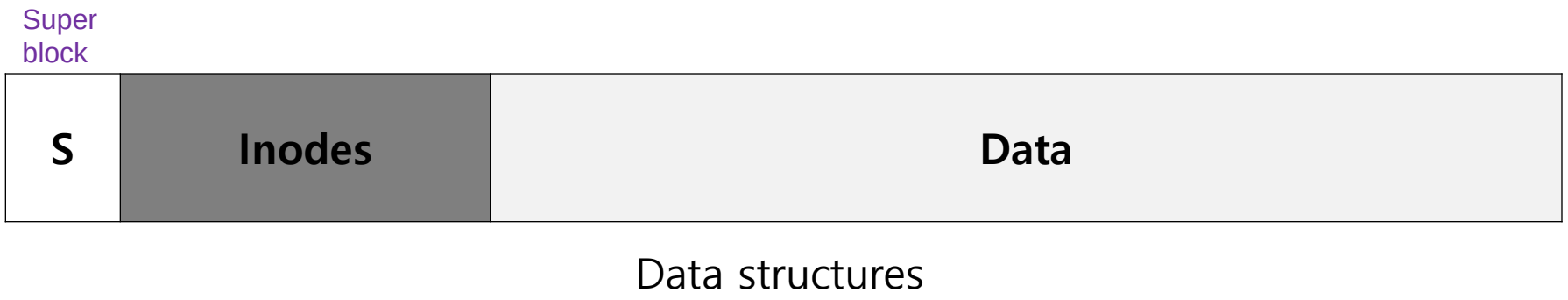
- **Reading and writing files are expensive**, incurring many I/Os.
  - **For example**, long pathname(/1/2/3/…./100/file.txt)
    - One to read the inode of the directory and at least one to read its data.
    - Literally perform tens of reads just to open the file.
- **In order to reduce I/O traffic**, file systems aggressively use system memory (DRAM) to cache.
  - **Early file system** used fixed-size cache to hold popular blocks.
    - **Static partitioning** of memory can be wasteful!
  - **Modern systems** use **dynamic partitioning** approach, unified page cache.
- **Read I/O** can be avoided by using large cache.

# Caching and Buffering (Cont.)

- **Write traffic** has to go to disk for persistence, Thus, cache does not reduce write I/Os.
- **File system use write buffering (delayed write)** for write performance benefits:
  - **delaying writes** (file system batch some updates into a smaller set of I/Os).
  - **By buffering** a number of writes in memory, the file system can then schedule the sequential/subsequent I/Os together + reorder writes to minimize seek/rotation time.
  - **By avoiding** frequent writes.
- **At some point** the file system needs to update i-node on disk and the data bit-map on disk.
- **Some application force flush data to disk (RDBMS)** by calling **fsync**() or direct I/O.

# Locality and the Berkeley Fast File System (FFS)

# Unix Operating System




- **The Good Thing:**
  - Simple and supports the basic abstractions.
  - Easy to use file system.
- **The Problem:**
  - Terrible performance

# Problem of Unix Operating System

- **Unix file system** treated the disk as a **random-access memory – not optimized mechanical movement.**
  - **Example of random-access blocks with Four files:**
    - Data blocks for each file can be accessed by going back and forth the disk, because they are not **contiguous.**

| A1 | A2 | B1 | B2 | C1 | C2 | D1 | D2 |
|---|---|---|---|---|---|---|---|

    - File b and d are deleted.

| A1 | A2 | | | C1 | C2 | | |
|---|---|---|---|---|---|---|---|

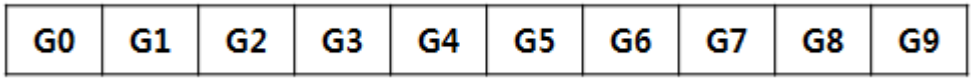
    - **File E** is created with free blocks. (**spread across** the drive)

| A1 | A2 | E1 | E2 | C1 | C2 | E3 | E4 |
|---|---|---|---|---|---|---|---|

- **Other Problem** is the original block size was too small (512 bytes)

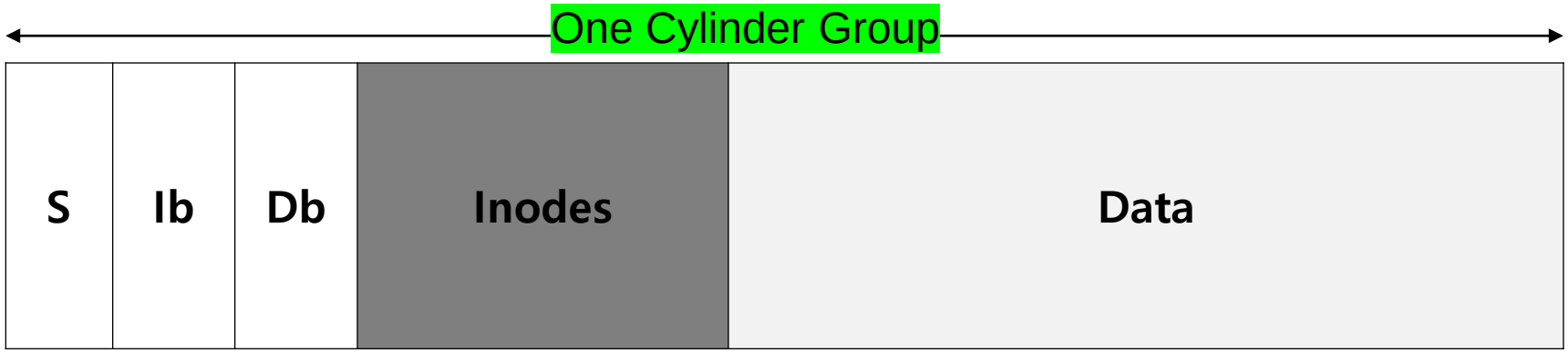
# FFS: Disk Awareness is the solution

- **FFS** is **Fast File system** designed by a group at Berkeley.
- **The design of FFS** is that file system structures and allocation polices to be “disk aware” to improve performance:
  - **Keep same API** with file system (open(), read(), write(), etc.).
  - **Changing** the internal implementation.
- **FFS** divides the disk into a bunch of groups **(Cylinder Group)**
  - **Modern file system** call cylinder group as block group.




  - **These groups** are used to improve seek performance.

# Organizing Structure: The Cylinder Group (Cont.)



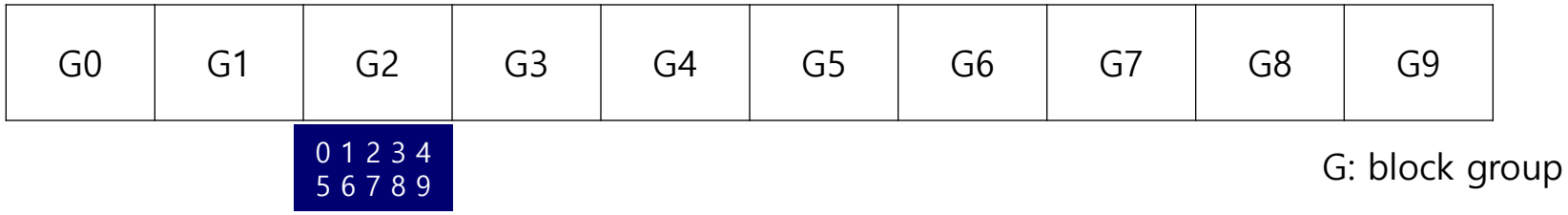

- **Data structure for each cylinder group:**
  - **A copy of the super block(S)** per cylinder group for reliability reason.
  - **inode bitmap(ib)** and **data bitmap(db)** to track free inode and data block within a cylinder group.
  - **Inode blocks** and **data blocks** are same as the previous very-simple file system(VSFS).

# How To Allocate Files and Directories?

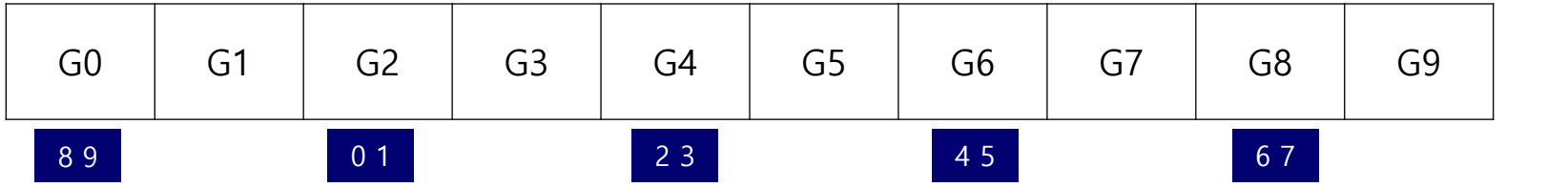
- **Policy** is "**keep related stuff together**"
- **The placement of directories:**
  - **Find the cylinder group** with a low number of allocated directories and a high number of free inodes.
  - **Put the directory data and inode** in that group.
- **The placement of files:**
  - **Allocate** data blocks of a file in the same group as its inode
  - **It places** all files in the same cylinder group as their directory (i.e., files and directory in the same group)

# The Large-File Exception

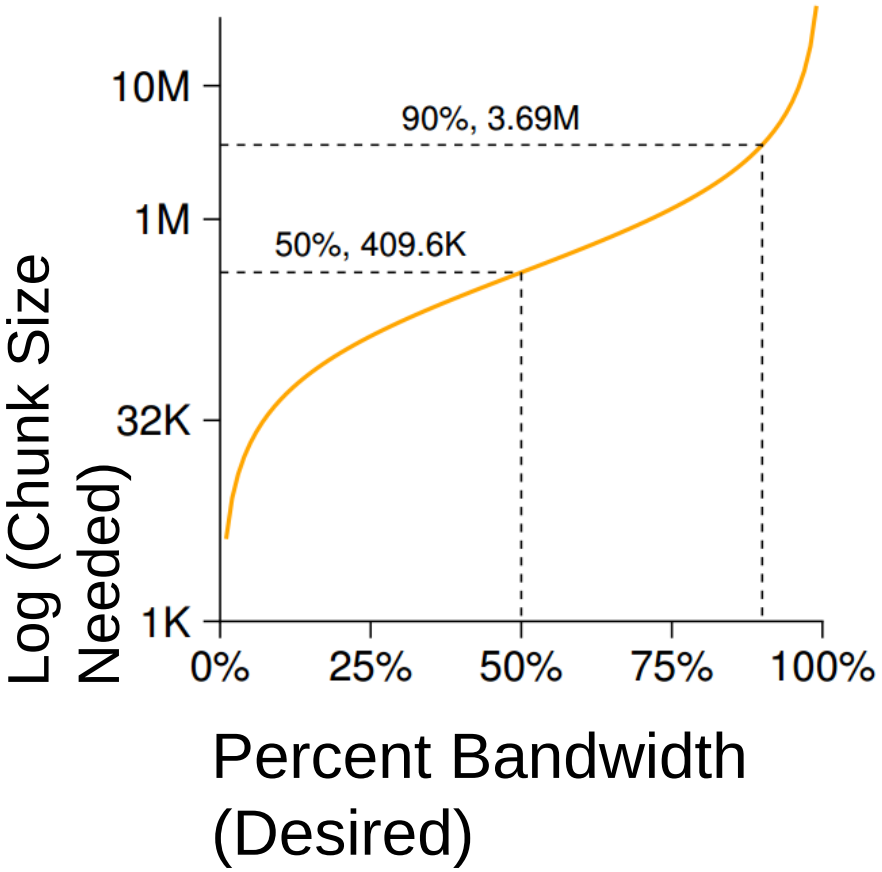
- **General policy of file placement:**
  - **Entirely fill the block group** the file is first placed within
  - **Hurt file-access locality** from “related” files being placed in different block groups!




- **For large files, chunks are spread across the disk:**
  - **Hurt performance**, but it can be addressed by choosing chunk size
  - **Amortization**: reducing overhead by doing more work in each IO



# Amortization: How Big Do Chunks Have To Be?




- Computation of the size of chunk
  - Desire 50% of peak disk performance
    - half of time seeking and half of time transferring
  - Disk bandwidth: 40 MB/s
  - Positioning time: 10ms
  - Data transfer in one seek =

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

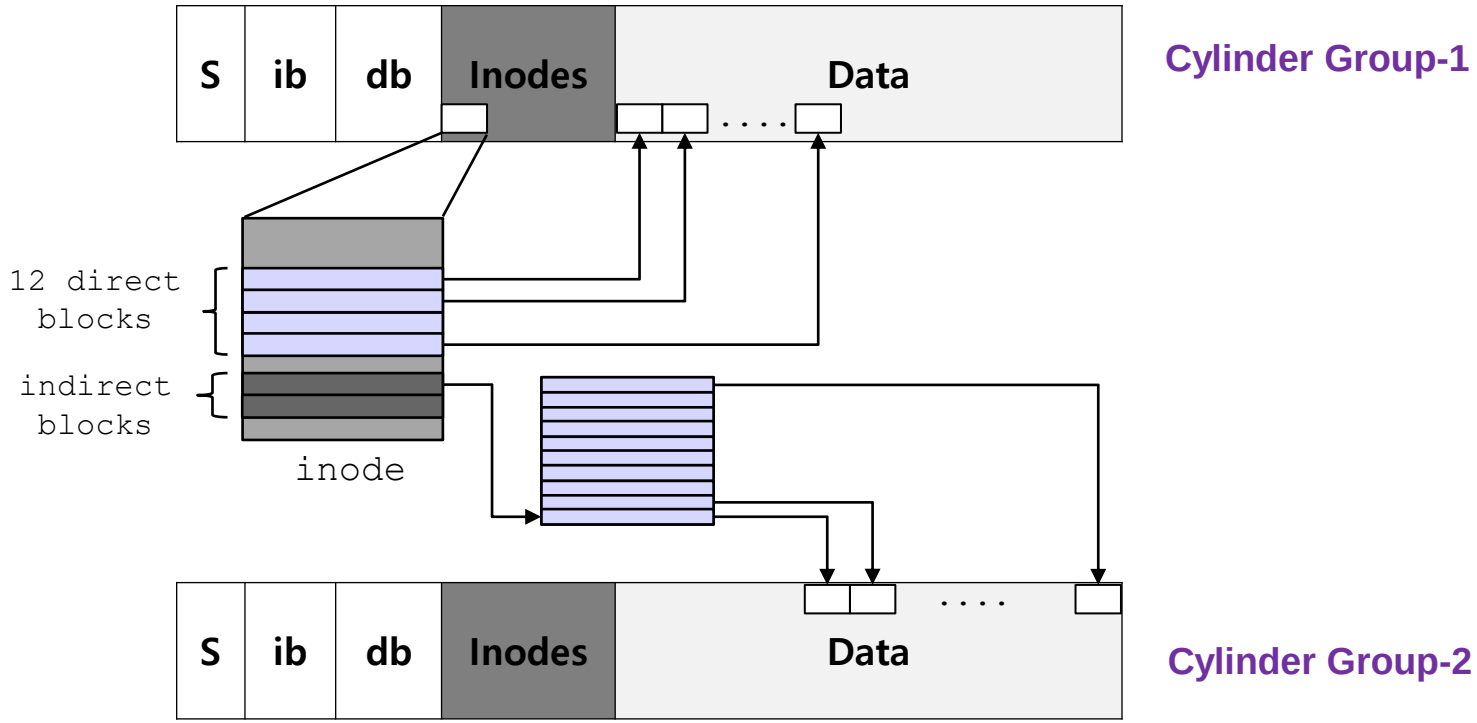
40.96 KB/msec . 10 msec =

$409.6\ KB = 0.4\ MB$

    - Transfer only 409.6 KB every time seeking
  - 99% of peak performance on 3.69MB chunk size

# The Large-File Exception in FSS*

- **A simple approach based on the structure of inode:**
  - Each subsequent **indirect block(s)**, and all the **blocks it pointes to**, are placed in **a different block group.**
  - Every **1024 blocks (4MB)** of the **file in a separate group**






* **FSS**: File Storage Service: https://www.usenix.org/system/files/atc19-kuszmaul.pdf

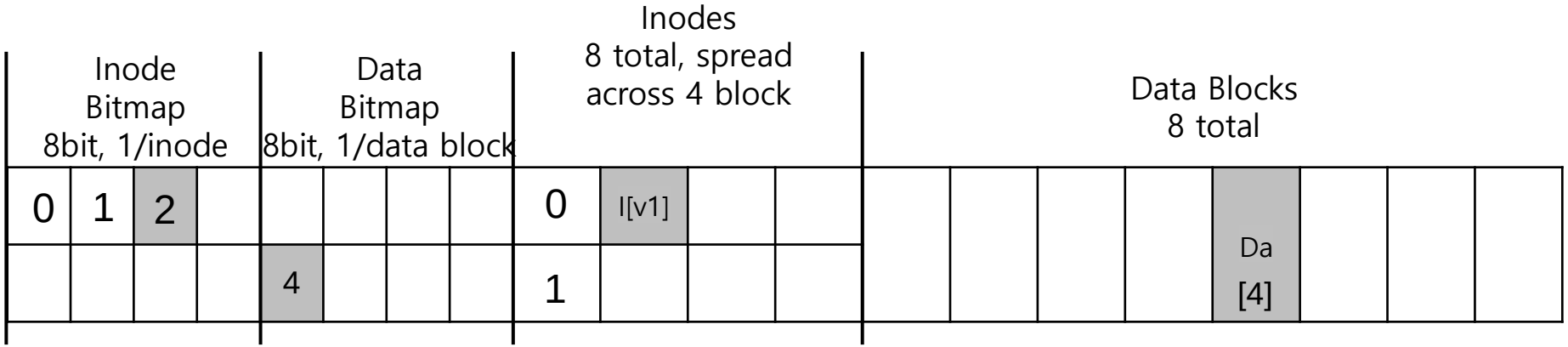
# Crash Consistency

# Crash Consistency

- **Unlike most data structure**, file system data structures must **persist:**
  - They must survive over the long haul, stored on devices that retain data despite power loss (e.g., disk).
- **One major challenge** faced by a file system is how to update persistent data structure despite the presence of a **power loss** or **system crash**.
- **We'll begin by examining** the approach taken by older file systems.
  - **fsck** (file system checker)
  - **journaling** (write-ahead logging)

# A Detailed Example (1)

- **Workload:**
  - **Append of a single data block** (4KB) to an existing file (that has already 1 block only)
  - open() → lseek() → write() → close()

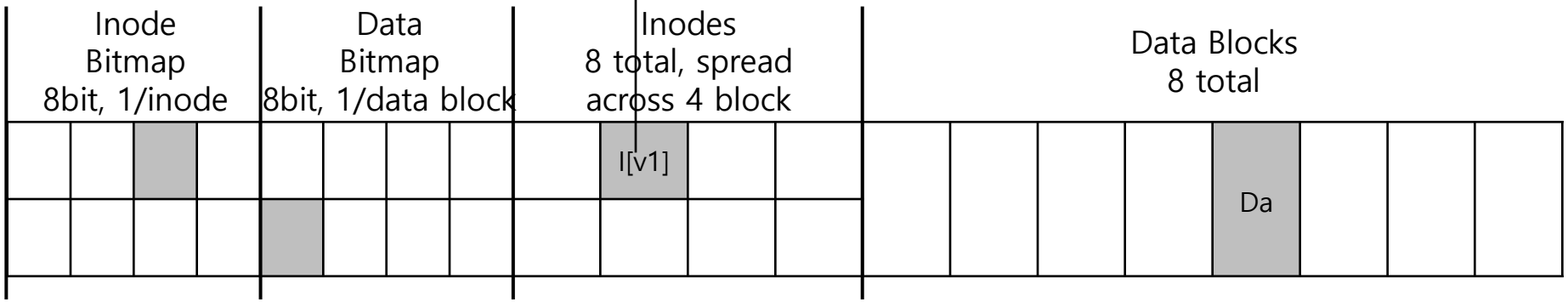



- **Before append a single data block:**
  - single inode is fetched (inode number 2)
  - single allocated data block (data block 4)
  - The inode is denoted I[v1]

# A Detailed Example (2)

- **Inside of I[v1] (inode, before update)**

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

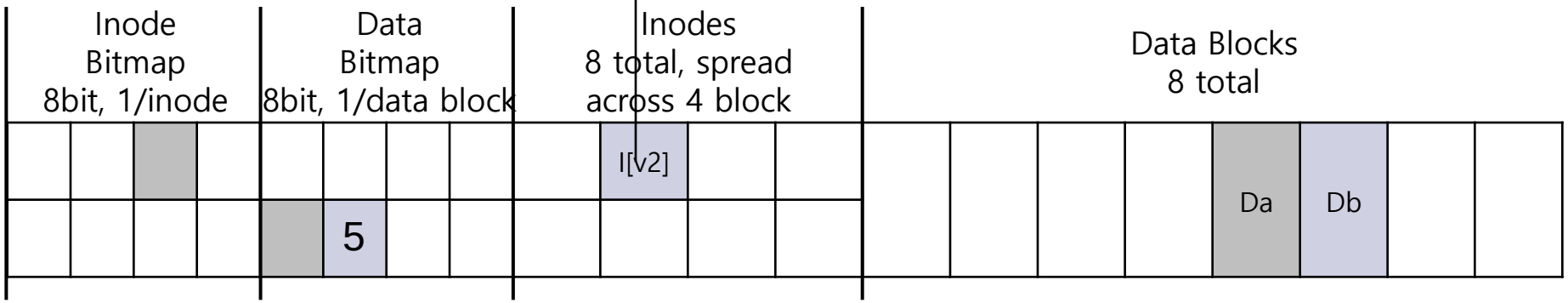



- Size of the file is 1 (one block allocated → Da = block-4)
- First direct pointer points to block-4 (Da)
- All 3 other direct pointers are set to null (unused)

# A Detailed Example (3)

- **After update**

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




- **Data bitmap** is updated to indicate latest DB Data version (i.e., V2) which indicates block 5 is allocated.
- **Inode** is updated (I[v2])
- **New data block** is written to disk (Db) – Data block

# A Detailed Example (end)

- **To achieve the transition**, the system performs **three separate writes to the disk**:
  - **inode I[v2]** – version number (content update)
  - **Data bitmap B[v2]** – version number (content update)
  - **Data block (Db)** – physical data update
- **These 3 writes usually don't happen immediately:**
  - **dirty** "inode, data bitmap, and new data" will sit in main memory (cache)
  - **page cache** or **buffer cache**
- **If a crash happens** after one or two of these 3 writes have taken place, but not all three, the file system could be left in a funny state

# Crash Scenario (1)

- **Imagine only a single write succeeds; there are thus three possible outcomes:**
  1. **Just the data block (Db) is written to disk**
     - The data is on disk, but there is no inode update
     - Thus, it is as if the write never occurred
     - This case is not a problem at all (but we lost the data)
  2. **Just the updated inode (I[v2]) is written to disk:**
     - The inode points to the disk address (5, Db)
     - But, the Db has not yet been written there
     - We will read **garbage** data (old contents of address 5) from the disk
     - **Problem:** file-system inconsistency

# Crash Scenario (2)

3. **Just the updated Data bitmap (B[v2]) is written to disk:**
   - The Data bitmap indicates that block 5 is allocated
   - But there is no inode that points to it, i.e., inode is not updated
   - Thus, the file system is inconsistent again
   - **Problem : space leak,** as block 5 would never be used by the file system → reset the Data bitmap to free block 5.

# Crash Scenario (3)

- **There are also three more crash scenarios. In these cases, two writes succeed and the last one fails**
  1. The inode(I[v2]) and data bitmap (B[v2]) are written to disk, but not data(Db)
     - The file system metadata is completely consistent
     - **Problem**: Block 5 has garbage in it
  2. The inode (I[v2]) and the data block (Db) are written, but not the Data bitmap (B[v2])
     - We have the inode pointing to the correct data on disk
     - **Problem**: inconsistency between the inode and the old version of the Data bitmap (B1). The data block can be allocated by another file (corruption)

# Crash Scenario (end)

3. The data bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2])
   - **Problem:** inconsistency between the inode (v1) and the data bitmap (v2); we have no idea which file that data block belongs to → disk block leakage

# The Crash Consistency Problem

- What we'd like to do ideally is to move the file system from one consistent state to another **atomically**
- **Unfortunately**, we can't do this easily:
  - **The disk only commits one write at a time**
  - **Crashes or power loss** may occur anytime between these updates
- We call this general problem the **crash-consistency problem**

# Solution #1: The File System Checker (Fsck)

https://www2.cs.duke.edu/csl/docs/sysadmin_course/sysadm-86.html

# The File System Checker (1)

- **The File System Checker (fsck*):**
  - **fsck** is an old Unix tool for finding inconsistencies and repairing them.
  - **fsck checks** Super block, Free block, Inode state, Inode links, etc. – more details:
    - **Free blocks** are stored in the cylinder group block maps. fsck checks that all the blocks marked as free are not claimed by any files (i-nodes). When all the blocks have been accounted for, fsck checks to see if the number of free blocks plus the number of blocks claimed by the inodes equal the total number of blocks in the file system. If anything is wrong with the block allocation maps, fsck rebuilds them, leaving out blocks already allocated.

# The File System Checker (2)

- **The summary information** in the superblock contains a count of the total number of free blocks within the file system. The fsck program compares this count to the number of free blocks it finds within the file system. If the counts do not agree, fsck replaces/updates the count in the superblock with the actual free-block count fsck just counted.

- **The above approach can't fix all problems:**
  - **Example:** The file system looks consistent but the inode points to garbage data.
- **The only real goal** is to make sure the file system metadata is internally consistent.

* https://docs.oracle.com/cd/E19455-01/805-7228/6j6q7uf0e/index.html

# The File System Checker (3)

- **Superblock**
  - **fsck** first checks if the superblock looks reasonable!
    - **Sanity checks**: file system size ≥ number of blocks allocated
  - **Goal:** to find suspect superblock (cylinder group superblock)
  - In this case, the system may decide to use an alternate copy of the superblock
- **Free blocks**
  - **fsck** scans the inodes, indirect blocks, double indirect blocks,
  - The only real goal is to make sure the file system metadata is internally consistent.

# The File System Checker (4)

- **Inode state**
  - Each inode is checked for corruption or other problem
    - **Example:** type checking (regular file, directory, symbolic link, etc.)
  - If there are problems with the inode fields that are not easily fixed.
    - The inode is considered suspect and cleared by fsck
- **Inode Links Count**
  - fsck also verifies the link count of each allocated inode
    - To verify the link count, fsck scans through the entire directory tree
  - If there is a mismatch between the newly–calculated count and that found within an inode, corrective action must be taken
    - Usually by fixing the count within the inode

# The File System Checker (5)

- **Inode Links Count** (Cont.)
  - If an allocated inode is discovered but no directory refers to it, it is moved to the lost+found directory
- **Duplicates**
  - fsck also checks for duplicated pointers
  - **Example:** Two different inodes refer to the same block:
    - If an inode is obviously bad, it may be cleared
    - Alternately, the pointed-to block could be copied

# The File System Checker (6)

- **Basic summary of what fsck does (Cont.):**
  - **Bad blocks**
    - **A check for bad block pointers** is also performed while scanning through the list of all pointers
    - **A pointer is considered "bad"** if it obviously points to something outside its valid range
    - **Example**: It has an address that refers to a block greater than or outside the partition size
      - In this case, fsck can't do anything too intelligent; it just removes the pointer

# The File System Checker (7)

- **Basic summary of what fsck does (Cont.):**
  - **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
    - **Example:**
      - making sure that "." and ".." are the first entries
      - each inode referred to in a directory entry is allocated in the i-bitmap?
      - ensuring that no directory is linked-to more than once in the entire hierarchy

# The File System Checker (end)

- **Buliding a working fsck** requires intricate knowledge of the file system.
- **fsck have a bigger and fundamental problem**: **too slow**
  - **scanning the entire disk** may take many minutes or hours
  - **Performance of fsck became prohibitive / expensive:**
    - as disk grew in capacity and RAIDs grew in popularity
- **At a higher level**, the basic premise of fsck seems just bad and irrational.
  - It is incredibly expensive to scan the entire disk more than once
  - It works but is wasteful/slow
  - Thus, as disk (and RAIDs) grew, researchers started to look for other solutions

# Solution #2: Journaling

# Journaling (1)

- **Journaling (Write-Ahead Logging - WAL):**
  - **When updating the disk**, before over writing the structures in place, first write down a little note describing what you are about to do
  - **Writing this note is the "write ahead" part**, and we write it to a structure that we organize as a "log"
  - **By writing the note to disk**, you are guaranteeing that if a crash takes places during the update of the structures you are updating, you can go back and look at the note you made and try it again
  - **Thus**, you will know exactly what to fix after a crash, instead of having to scan the entire disk multiple times

# Journaling (Cont.)

- **We'll describe how Linux ext3 incorporates journaling into the file system:**
  - **Most of the on-disk structures** are identical to Linux ext2
  - **The new key structure is the journal** itself
  - **It occupies some small amount of space** within the partition or on another device


**Fig.1 Ext2 File system structure**


**Fig.2 Ext3 File system structure**

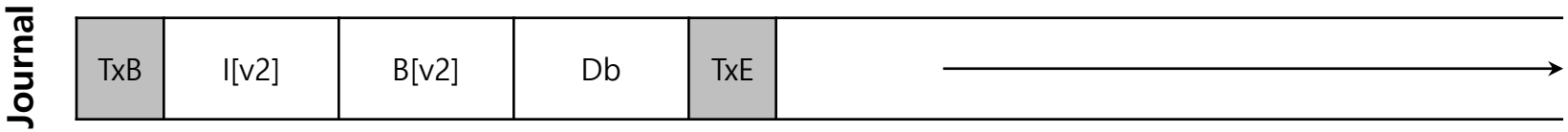
# Data Journaling (1)

- **Data journaling** is available as a mode with the ext3 file system

- **Example:** our canonical update again:
  - **We wish to update** inode (I[v2]), data bitmap (B[v2]), and data block (Db) to disk
  - **inode bitmap** is not modified while updating a file, so it is not relevant in this context
  - **Before writing them (i.e., the 3 IO) to their final disk locations**, we are now first going to write them to the log (a.k.a. journal)

# Data Journaling (2)

- **Writing the Journal log:**




- **TxB:** Transaction Begin block
  - It contains some kind of **transaction identifier(TID)**
- **The middle three blocks** just contain the exact content of the three blocks themselves
  - This is known as **physical logging**
- **TxE:** Transaction End block
  - Marker of the end of this transaction
  - It also contain the TID

# Data Journaling (3)

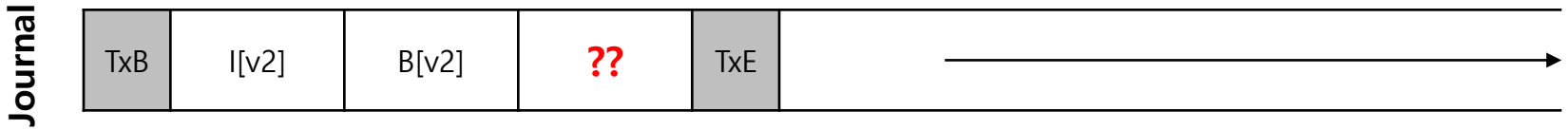
- **Checkpoint:**
  - **Once this transaction is safely on disk (i.e., written the Journal log)**, we are ready to overwrite the current structures (metadata) in the file system
  - **This process (writing the 3 I/O to disk)** is called **checkpointing**
  - **Thus, to checkpoint the file system**, we issue the writes I[v2], B[v2], and Db to their disk locations
- **Our initial sequence of operations:**
  1. **Journal (log record) write**
     - Write the tx to the log and wait for these writes to complete
     - **TxB**, all pending data, metadata updates, **TxE**
  2. **Checkpoint (update the metadata on disk)**
     - Write the pending metadata and data updates to their final locations

# Data Journaling (4)

- **When a crash occurs during the writes to the journal:**
  1. **Transaction - each one at a time (i.e., sequentially):**
     - 5 transactions (TxB, I[v2], B[v2], Db, TxE)
     - This is slow because of waiting for each to complete
  2. **Transaction - all block writes at once:**
     - Five writes → a single sequential write: Faster way
     - **However, this is unsafe**
       - Given such a big write, the disk internally may perform scheduling and complete small pieces of the big write in any order; the disk may **(a)** write TxB, I[v2], B[v2], and TxE and only later **(b)** write Db

# Data Journaling (5)

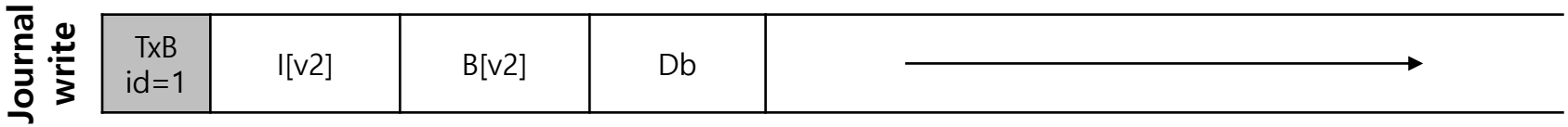
- **Unfortunately**, if the disk loses power between (a) and (b)!



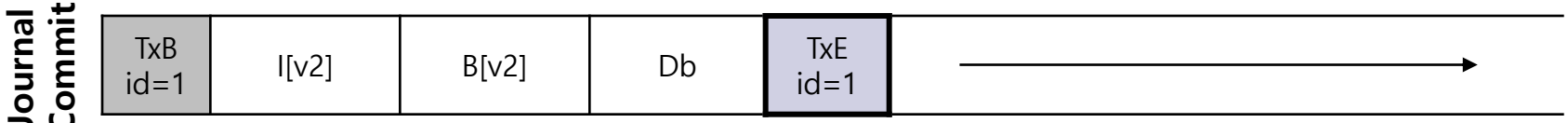

  - Transaction looks like a valid transaction:
    - Further, the file system can't look at that forth block and know it is wrong?
    - It is much worse if it happens to a critical piece of file system, such as superblock.

# Data Journaling (6)

- **To avoid this problem**, the file system issues the transactional write in two steps
- **First**, writes all blokes **except the TxE block** to journal




- **Second**, The file system issues the write of the TxE




- **An important aspect of this process** is the atomicity guarantee provided by the disk:
  - The disk guarantees that any 512-byte write either happen or not
  - Thus, TxE should be a single 512-byte block

# Data Journaling (7)

- Thus, our current protocol to update the file system, with each of its three phases labeled:
  1. **Journal write**: write the contents of the transaction (4 IO) to the log
  2. **Journal commit (TxE block) added:** write the transaction commit block after step-1 is completed
  3. **Checkpoint:** write the contents of the update (Data Block) to their locations in the file system

# Data Journaling (end)

- **Recovery**
  - **If the crash happens before the transactions** is written to the log (i.e., including the tx commit)
    - The pending update is **skipped/lost**
  - **If the crash happens after the transactions** (3 IOs) are written to the log, but **before the checkpoint is done**
    - **Recover** the update as follow:
      - Scan the log and look for transactions that have committed (TxE) to the disk (log records)
      - Transactions (metadata) are replayed

# Log-structured File Systems

# LFS: Log-structured File System

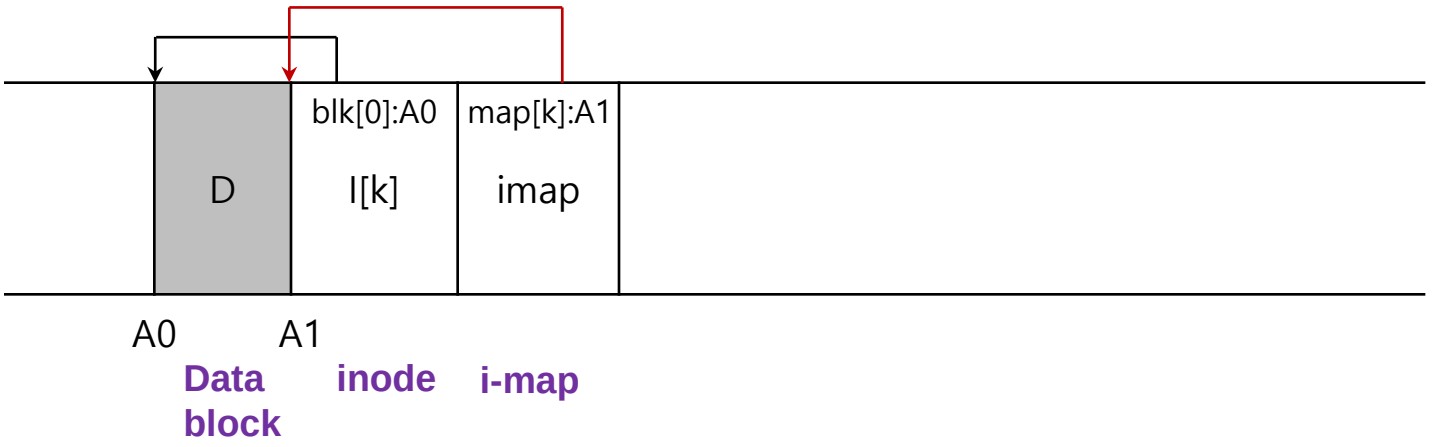
- **Memory sizes** are growing.
- **Large gap** between random IO and sequential IO performance.
- **Existing File System** performs poorly on common workloads.
- **File System were not** RAID-aware.

# Writing to Disk Sequentially and Effectively

- **Writing single blocks sequentially** does not guarantee efficient writes:
  - After writing into A0, next write to A1 will be delayed by a disk rotation (as you missed beginning of A1)
- **Write buffering (delayed writes) for effectiveness:**
  - Keeps track of updates in **memory buffer** cache (also called **dirty buffers**)
  - Writes them to disk all at once, when it has sufficient number of updates to optimize (Merge) I/O.

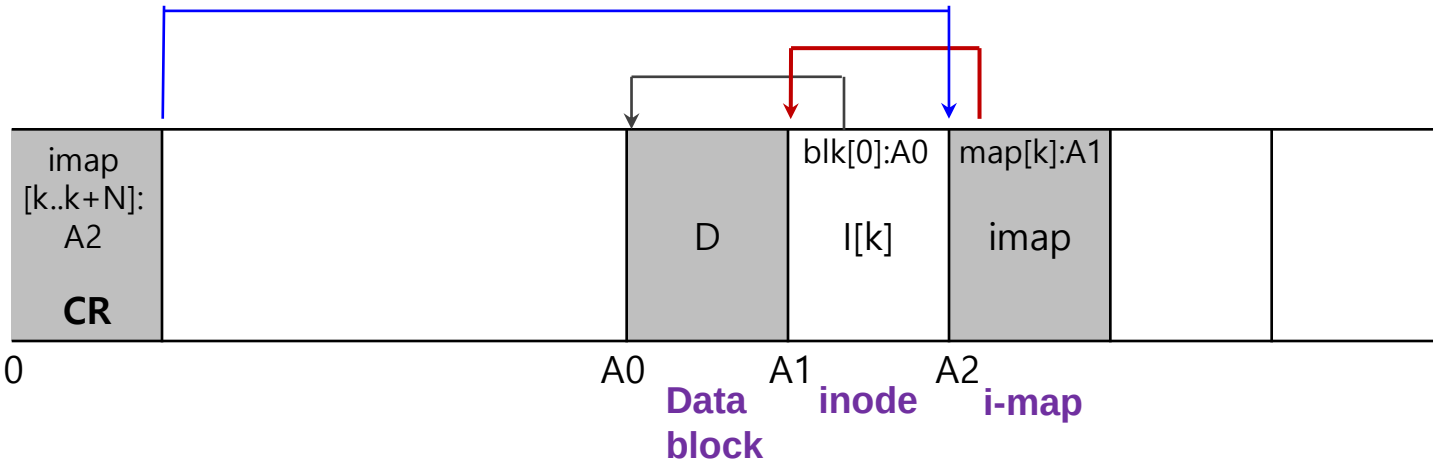
# Finding Inode in LFS

- **Inodes** are scattered throughout the disk!
- **Solution** is through indirection "**Inode-Map**" (imap)
- **LFS (Log-structured File System)** place the chunks of the inode-map right next to where it is writing all of the other new information (i.e., inode, data block)



# The Checkpoint Region (CR)

- **How to find the inode map** - spread across the disk (per disk partition)?
  - **The LFS File system** have fixed location on disk to begin a file lookup
- **Checkpoint Region (CR)** contains pointers to the latest of the inode map:
  - Only updated periodically (ex. Every 30 seconds)
    → performance is not ill-affected

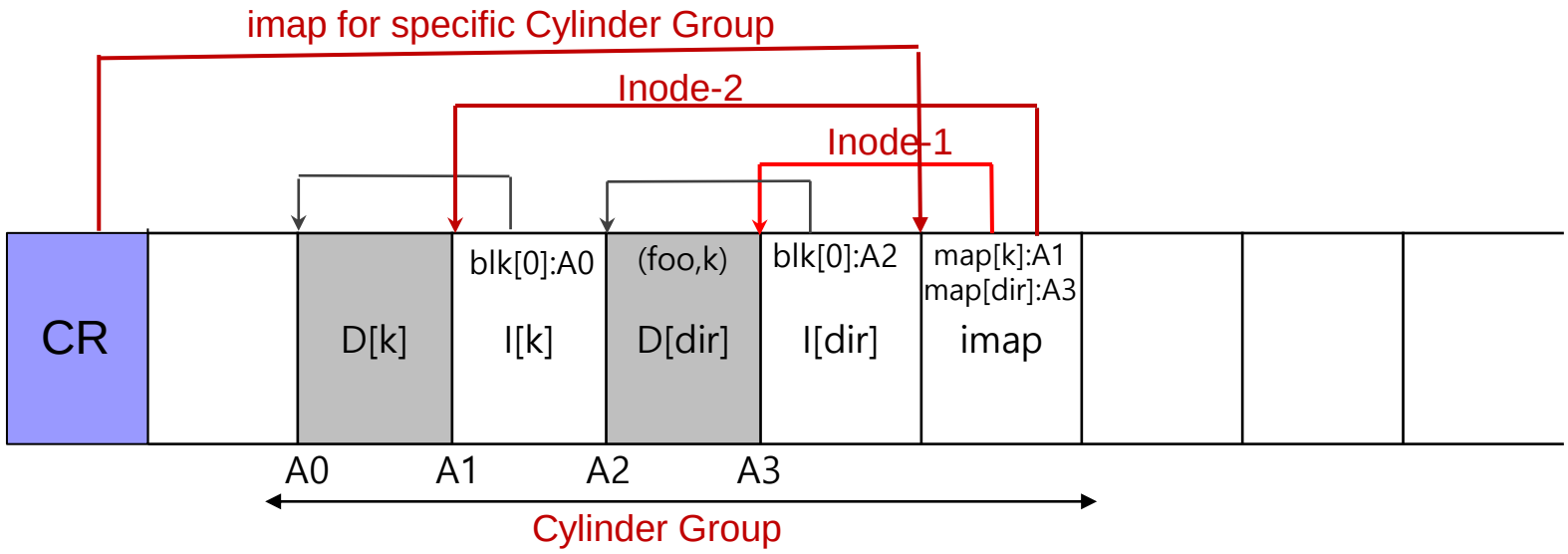


# Reading a File from Disk: A Recap

- **Read checkpoint region** (CR)
- **Read entire inode map** for the relevant cylinder group (disk partition) and cache it in memory
- **Read the most recent inode**
- **Read a block from file** by using direct or indirect or doubly-indirect pointers

# What About Directories?

- **Directory structure of LFS** is basically identical to classic UNIX file systems:
  - **Directory is a file** which its data blocks consist of directory information



# Data Integrity and Protection

# Disk Failure Modes

- **Latent Sector Error (LSE):** when sector(s) on the drive becomes inaccessible.
- **Common failures** are **frequency of LSE** and **block corruption**.

| | Cheap drive | Costly drive | Comment |
|---|---|---|---|
| LSEs | 9.40% | 1.40% | Costly drive is 7X better than cheap drive |
| Corruption | 0.50% | 0.05% | Costly drive is 10X better than cheap drive |

**Frequency of LSEs and Block Corruption**

# Disk Failure Modes (Cont.)

- **Frequency of latent-sector errors (LSEs):**
  - **Costly drives** with more than one LSE are as likely to develop additional LSEs.
  - **For most drives**, annual error rate increases in year two.
  - **LSEs increase with disk size**.
  - **Most disks** with LSEs have less than 50.
  - **Disks with LSEs** are more likely to develop additional LSEs.
  - **There exists** a significant amount of spatial and temporal locality.

# Disk Failure Modes (Cont.)

- **Block corruption:**
  - **Chance of corruption varies** greatly across different drive models; even within the same drive class
  - **Age affects** are different across models
  - **Workload and disk size** have little impact on corruption
  - **Most disks** with corruption only have a few corruptions
  - **Corruption is not independent** within a disk or across disks in RAID
  - ***There exists spatial locality***, and some temporal locality
  - **There is a weak correlation** with LSEs

# Handling Latent Sector Errors

- **Latent sector errors (LSE)** are easily detected and handled.

- **Using redundancy mechanisms**:
  - In a mirrored RAID-1 or RAID-4 and RAID-5 system based on parity, the system should reconstruct the block from the other data blocks and the parity block in the parity group.

# Detecting Corruption: The Checksum

- **How** can a client tells that a block has gone bad?

- **Using Checksum mechanisms:**
  - **This is simple:** the result of a function that takes a chunk of data as input and computes a function over said data, producing a small summary (checksum) of the contents of the data and compares with existing checksum.

# Using Checksums

- **When reading a block D**, the client reads its checksum from disk Cs(D), **stored checksum**
- **Computes the checksum** over the retrieved block D, **computed checksum** Cc(D).
- **Compares the stored and computed checksums:**
  - If they are equal (Cs(D) == Cc(D)), the data is safe.
  - If they do not match (Cs(D) != Cc(D)), the data has changed since the time it was stored (since the stored checksum reflects the value of the data at that time).

# Scrubbing:

## Periodically check the blocks checksums

- **When do these checksums** actually get checked?
  - Most data is rarely accessed, and thus remain unchecked.

- **To remedy this problem**, many systems utilize **disk scrubbing**:
  - **By periodically reading** through every block of the system
  - **Checking** whether checksum are still valid
  - **Reduce the chances** that all copies of certain piece of data become corrupted; then you cannot recover!

# Overhead of Checksumming

- **Two distinct kinds of overheads**: **space** and **time**
- **Space overheads:**
  - **Disk space itself**: A typical ratio might be an 8-byte checksum per 4KB data block, for a (0.0019 = 0.19%) on-disk space overhead.
  - **Memory of the system**: This overhead is short-lived and is not much of a concern.
- **Time overheads:**
  - **CPU must compute the checksum over each block:**
    - To reduce CPU overheads; combine data copying and check-summing into one streamlined activity, i.e., compute checksum while you are copying the data.

# END