Chapter 6: I/O Management

Sukomal Pal, CSE, IIT(BHU)

Chapter 6. I/O Management

- *I/O Hardware: I/O devices, Device controllers, Direct memory access, Principles of I/O Software: Goals of Interrupt handlers, Device drivers, Device independent I/O software, Secondary-Storage Structure: Disk structure, Disk scheduling algorithms*
- *Disk Management: Disk structure, Disk scheduling - FCFS, SSTF, SCAN, C-SCAN, Disk reliability, Disk formatting, Boot-block, Bad blocks*
- *File Management: Concept of File, Access methods, File types, File operation, Directory structure, File System structure, Allocation methods (contiguous, linked, indexed), Free-space management (bit vector, linked list, grouping), directory implementation (linear list, hash table), efficiency and performance.*

Introduction

- A computer interacts with users through various I/O devices:
  - Input: Keyboard, mouse, scanner
  - Output: Display, printer
  - Communication: Network devices
- These are peripheral devices, not core components like CPU, bus, or memory.
- I/O devices differ in form and function but universally manage data for the processor.
- The OS abstracts device complexity and provides a unified interface for:
  - Application programs
  - Kernel modules
- I/O management is a core responsibility of modern operating systems.

I/O Hardware

- I/O devices are peripheral units for input, output, storage, and communication.
- All devices communicate data for the processor.
- OS abstracts I/O complexities for easy use.
- Sections: Devices, Controllers, Processor, DMA

I/O Devices

- Classified by purpose:
  - Communication Devices (keyboard, monitor, NIC)
  - Storage Devices (HDD, SSD, tape)
- Classification by characteristics:
  - Character vs Block
  - Read-only, Write-only, Read-Write
  - Slow vs Fast, Serial vs Parallel, etc.
- Devices connect via **Device Controller** through an I/O bus.

Device Controller

- A controller is an electronic unit with different levels of circuits
- May include microprocessors/microcode
- Uses registers (Command, Status, Input, Output)
- Example: SCSI-HBA
- Controls multiple similar devices

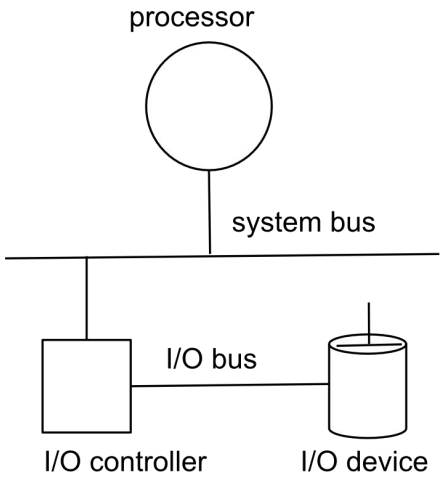


Fig 6.1: I/O Device , bus , controller

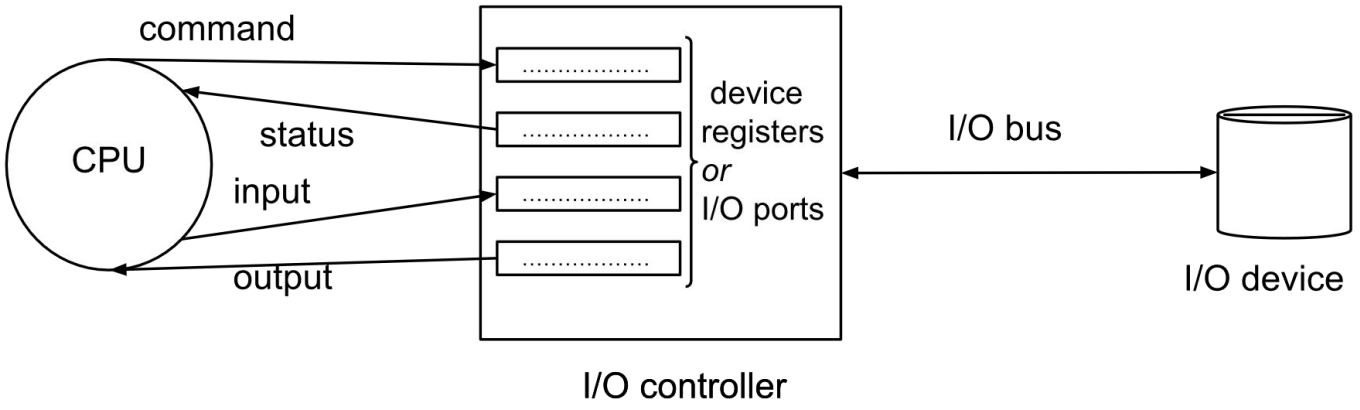


Fig 6.2: I/O Controller and Device Register

Processor and Interrupt Handling

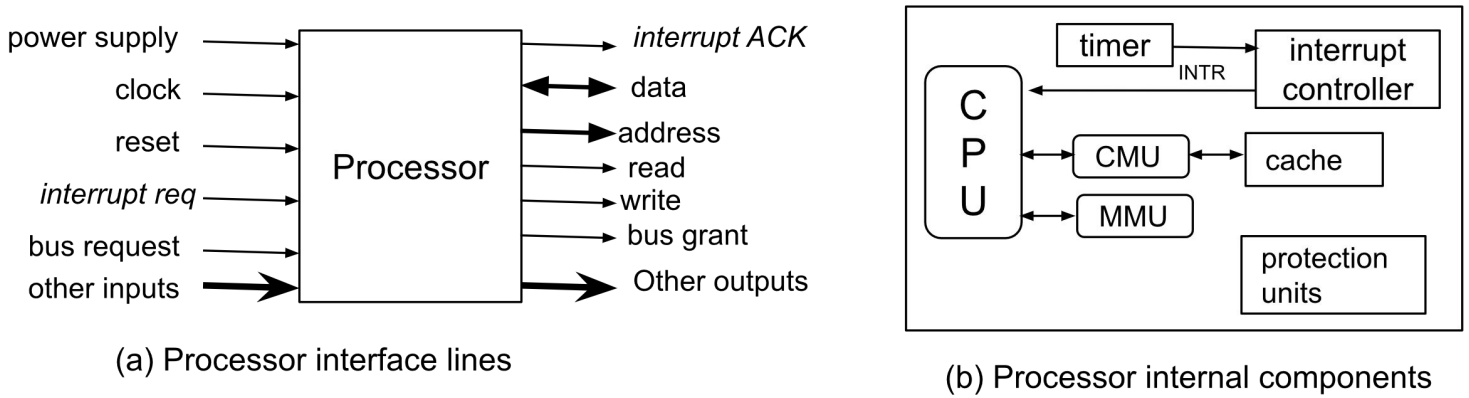


Fig 6.3: Processor Interaction and components

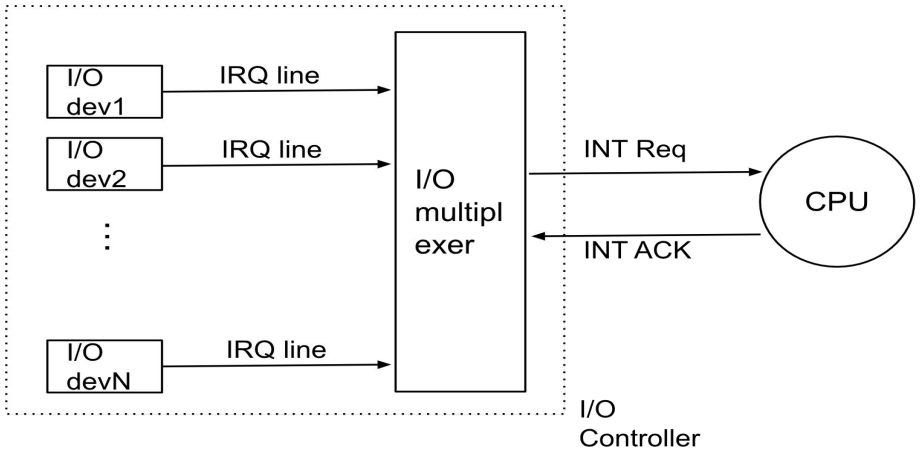


Fig 6.4: I/O controller and CPU

Direct Memory Access

- DMA transfers data directly between I/O and memory
- Reduces CPU involvement (initial setup + interrupt on completion)
- Modes:
  - **Cycle stealing**: Interleaved bus use
  - **Burst mode**: Exclusive bus access
  - **Fly-by mode**: One-clock-cycle direct transfer

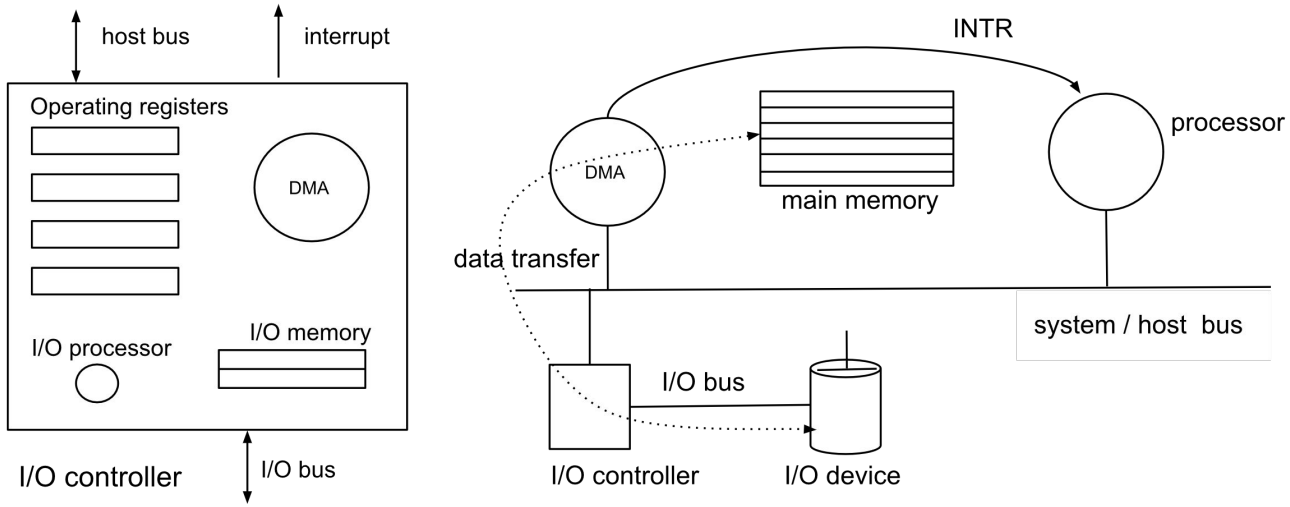


Fig. 6.5: Large-scale data transfer using DMA

I/O Software

- I/O is interrupt-driven in modern systems
- OS manages I/O with layered software
- Major components:
  - Interrupt Handlers
  - Device Drivers
  - I/O Subsystem

Interrupt Handlers

- Handle CPU interruptions from I/O devices
- Two-part structure:
  - **FLIH**: Fast, hardware-specific (saves state, triggers SLIH)
  - **SLIH**: Slower, device-specific logic
- Also manage software traps/exceptions
- Enable CPU-device communication
- Prioritize interrupts
- Balance between interrupt-driven and programmed I/O

Device Drivers

- Lowest OS layer for I/O device control
- Kernel module, interacts with device controller
- Types:
  - **Block** (e.g., HDD)
  - **Character** (e.g., keyboard, network)
- Follows **KDIM** (Kernel-Device Interface Model)
- Provides modular, device-independent access
- Registered at boot/load time

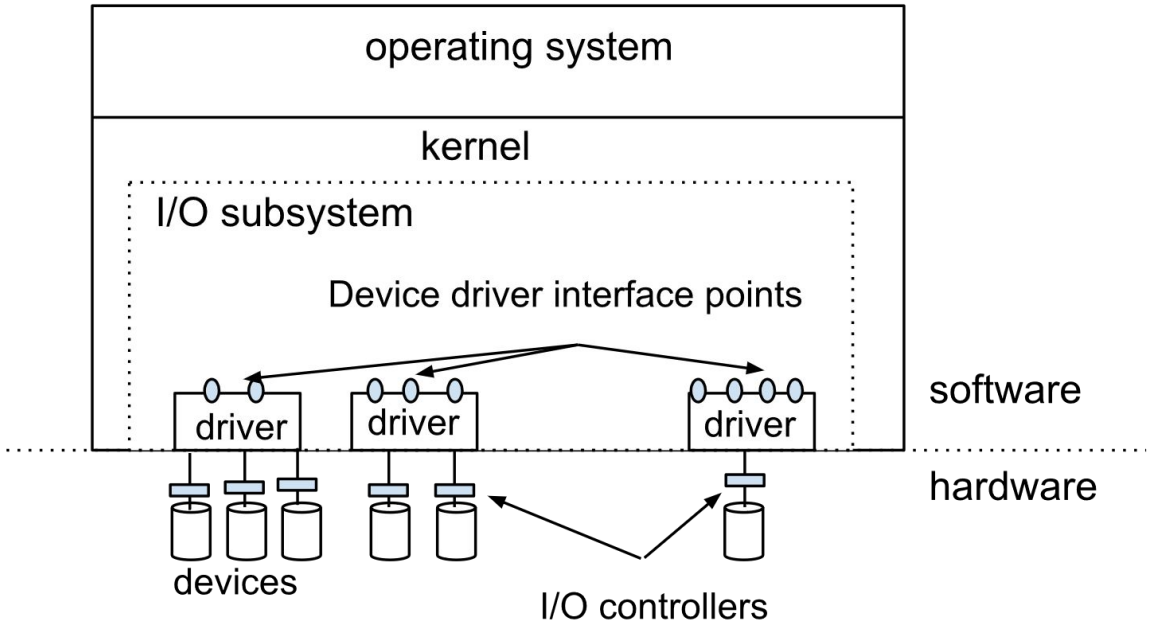


Fig. 6.6: Devices and device drivers

Device Driver Workflow

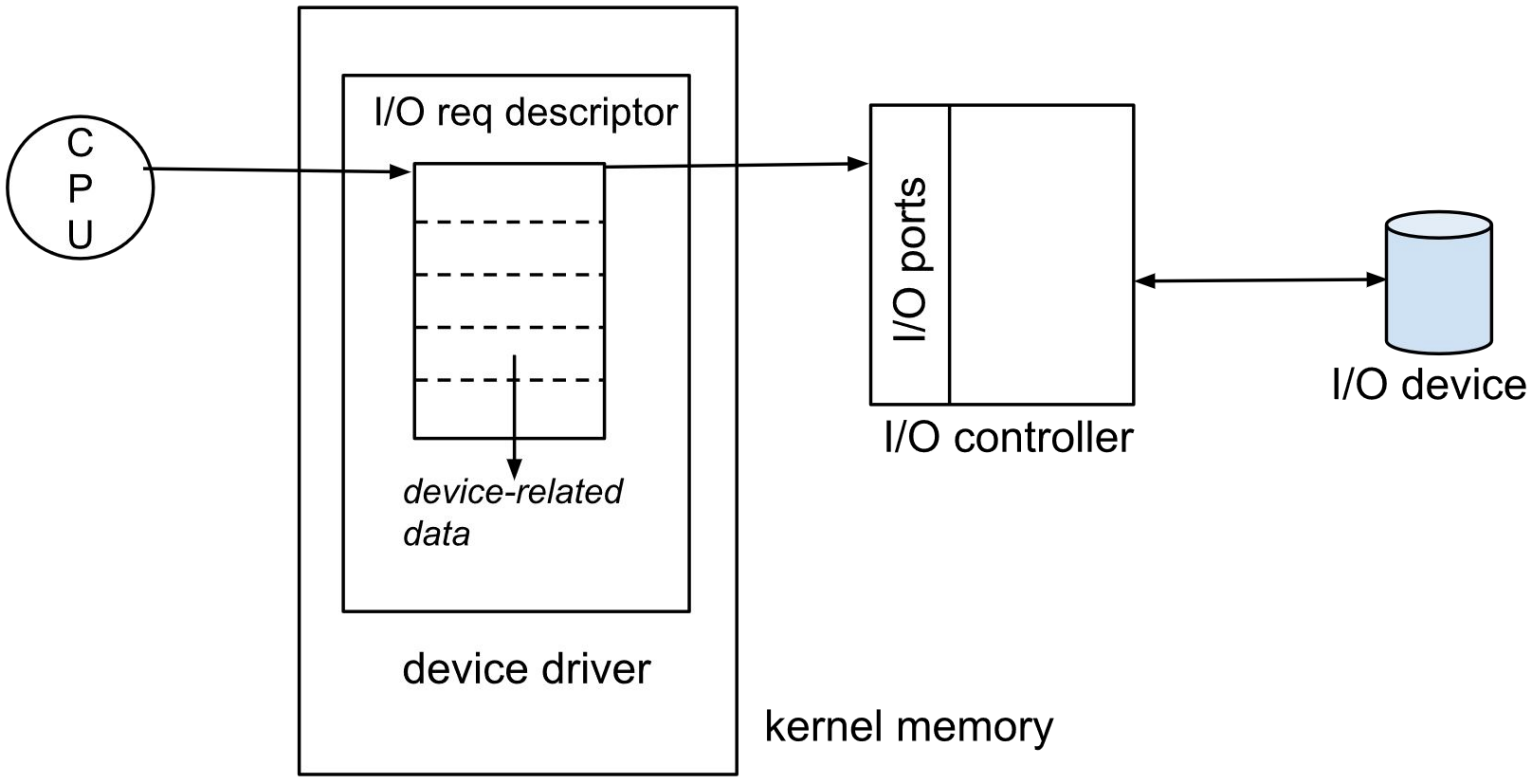


Fig 6.7: Hardware-software interaction for I/O operation

I/O Subsystem

- Device-independent I/O layer in kernel
- Manages and abstracts all device drivers
- Functions:
  - Allocates devices
  - Manages buffers/cache
  - Categorizes devices:
    - Block, Character, Network
- Uses major/minor device numbers for access
- Enables asynchronous (non-blocking) I/O

I/O Subsystem

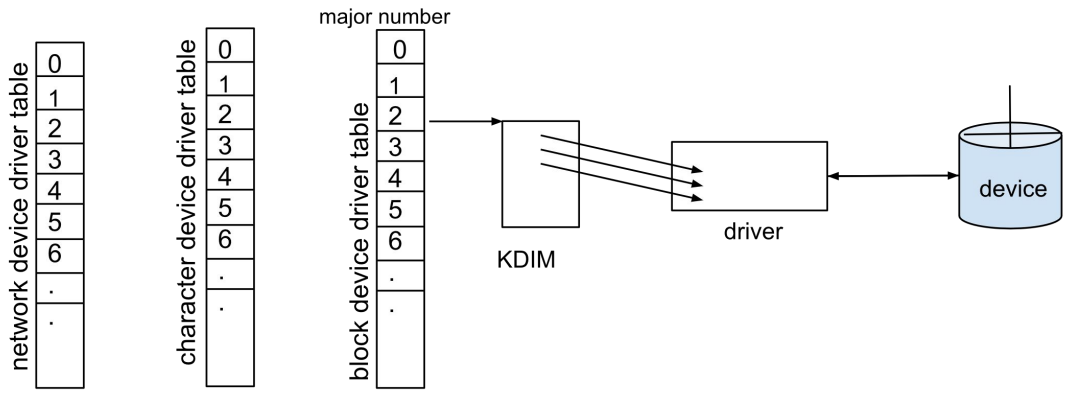


Fig 6.8: Device Driver tables are maintained in I/O Subsystem

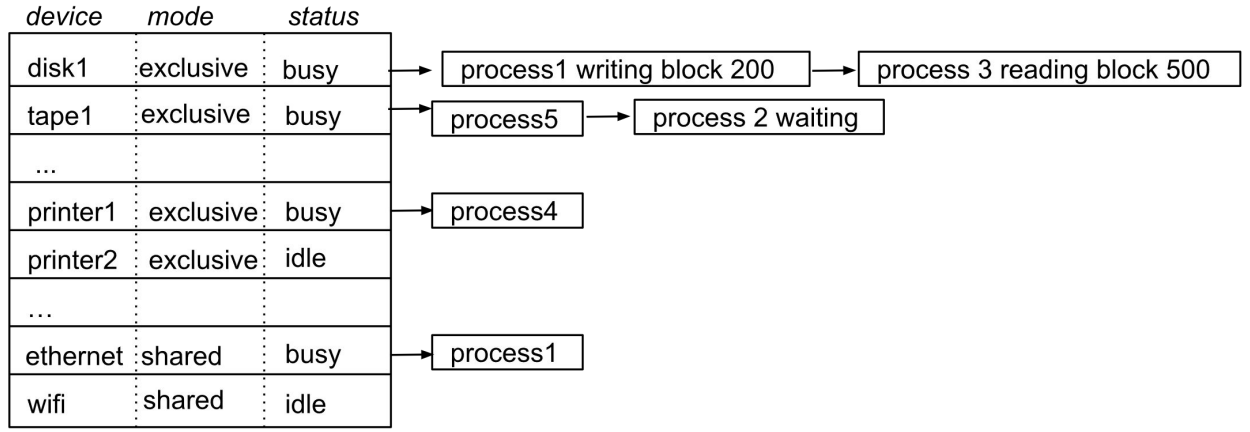


Fig 6.9: Device status tables are maintained in I/O Subsystem

SECONDARY STORAGE STRUCTURE

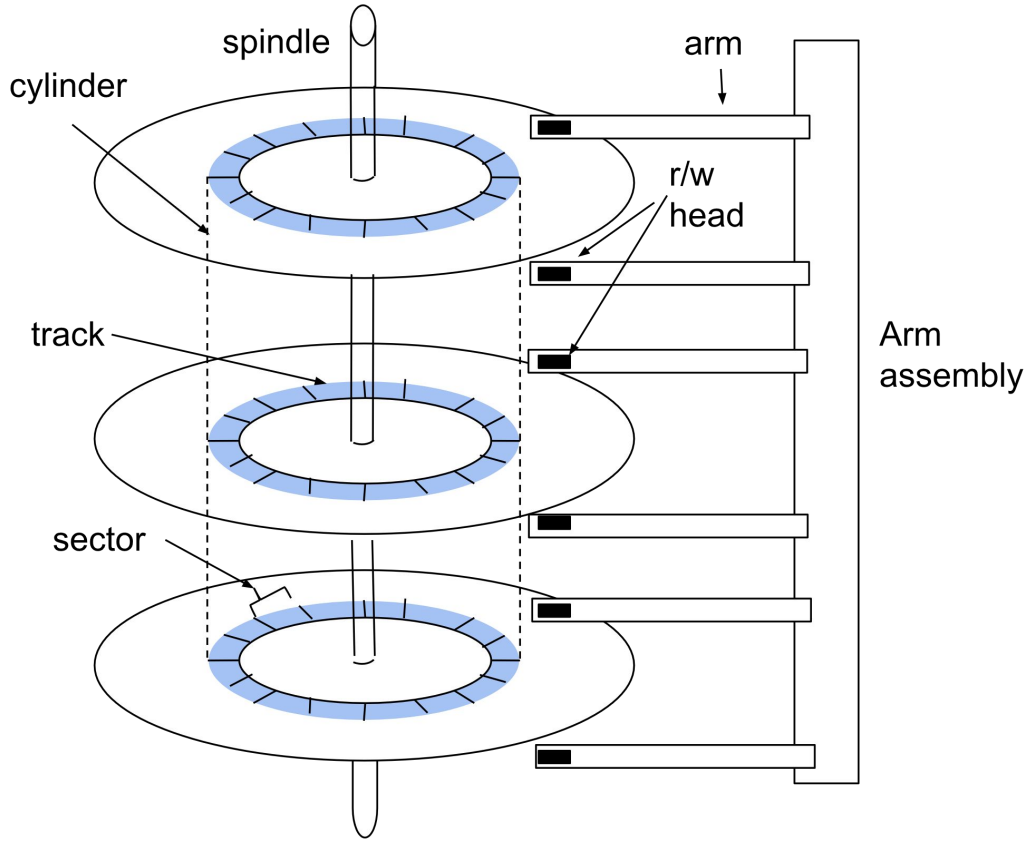


Fig. 6.10: Magnetic Disk Structure

Disk Structure

- A Hard Disk Drive (HDD) is made of magnetic platters stacked on a spindle
- Each platter has:
  - Two read/write heads (one per side)
  - Divided into **tracks**, **sectors**, and **cylinders**
- Sectors typically store 512–1024 bytes
- Unique sector identification:

  <cylinder, track, sector>
- Disk Access Components:
  - **Seek Time**: Move head to track
  - **Rotational Latency**: Wait for sector under head
  - **Data Transfer Time**: Actual data read/write
- **Total Disk Access Time** = Seek Time + Rotational Latency + Data Transfer Time.

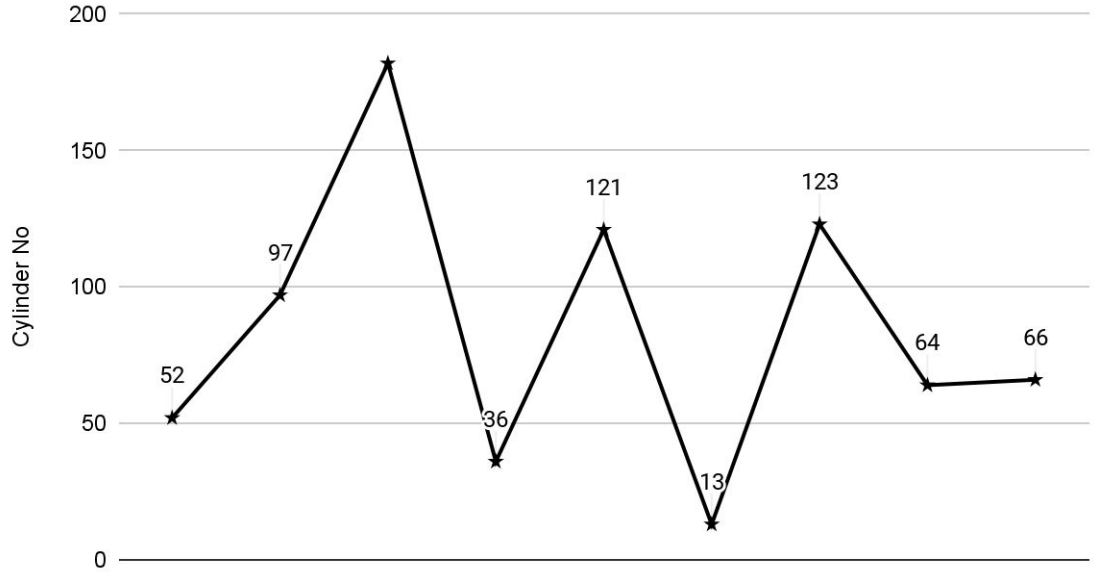
Disk Scheduling

- Seek time is the dominant factor in disk access time.
- Goal: Minimize average/total seek time by scheduling disk access requests.
- Given reference string (initial head at 52): 97, 182, 36, 121, 13, 123, 64, 66
- Disk has 200 cylinders (0–199)

FCFS Scheduling

- The total number of *to and fro* movement of the r/w head is = (97- 52) + (182 - 97) + (182 - 36) + (121 - 36) + (121 -13) + (123 - 13) + (123 - 64) + (66 - 64) = 45 + 85 + 146 + 85 + 108 + 110 + 59 + 2 = 640 cylinders.

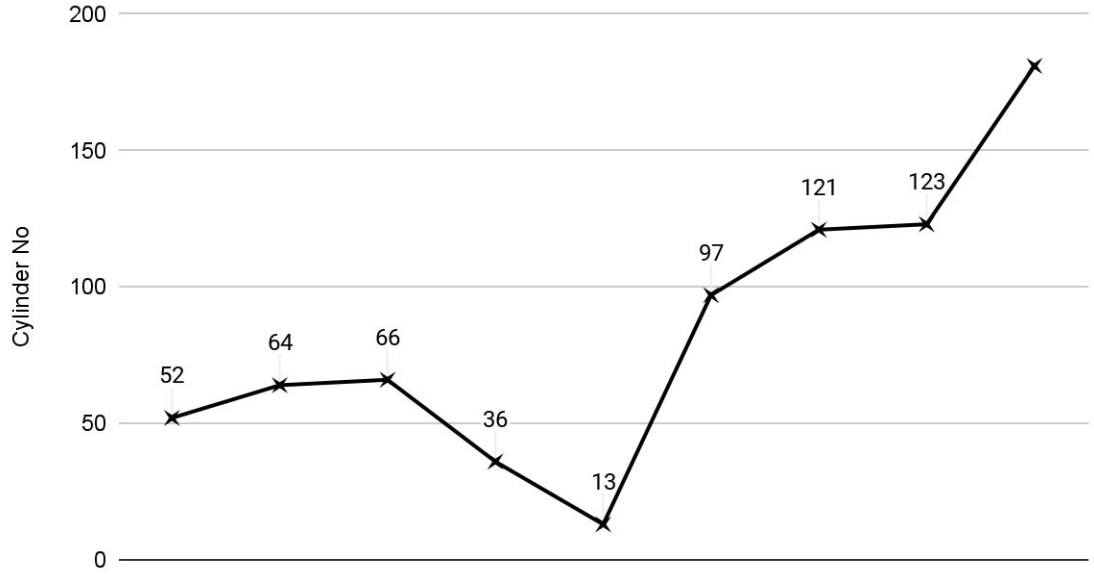
Fig 6.11: FCFS Scheduling algorithm



SSTF Scheduling

- The total number of to and from movement is = (64 - 52) + (66 - 64) + (66 - 36) + (36 - 13) + (97 - 13) + (121 - 97) + (123 - 121) + (187 - 121) = 243 cylinders.

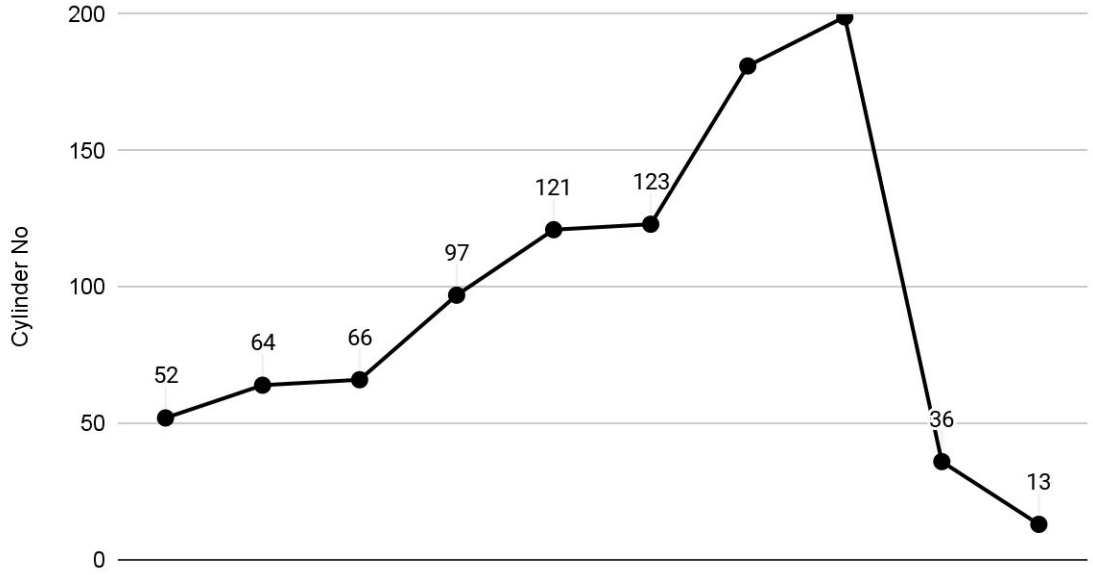
Fig. 6.12. SSTF Algorithm



SCAN Scheduling

- Total head movement for the reference string is = (64-52) + (66 - 64) + (97- 66) + (121 - 97) + (123 - 121) + (187 - 123) + (199 - 187) + (199 - 36) + (36 - 13) = (199 - 52) + (199 - 13) = 147 + 186 = 333 cylinders.

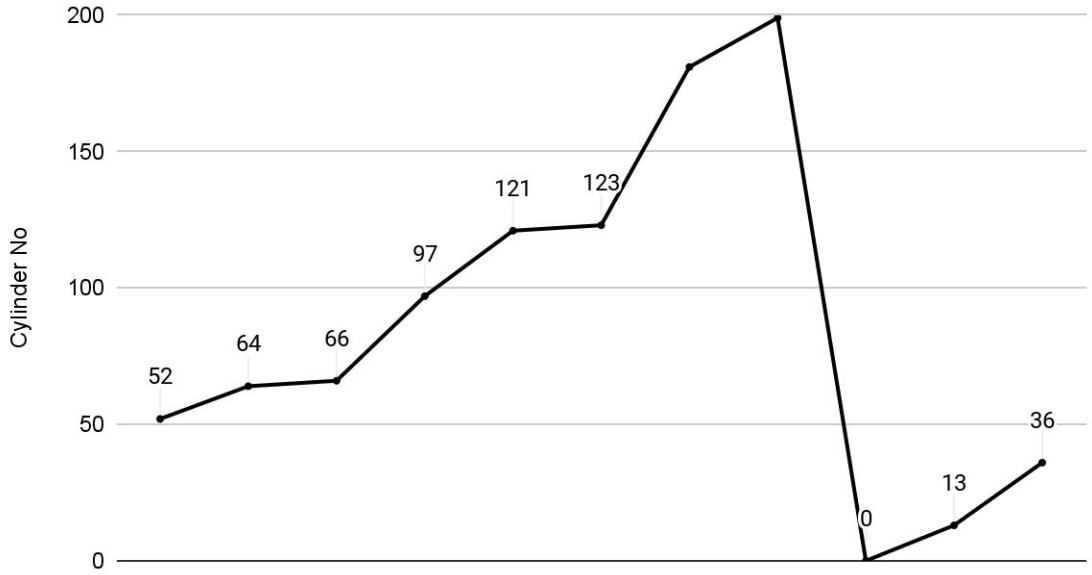
Fig. 6.13 SCAN algorithm



C-SCAN Scheduling

- Total amount of head movement is = (64 - 52) + (64 - 64) + (97 - 66) + (121 - 97) + (123 - 121) + (181 - 123) + (199 - 181) + (199 - 0) + (13 - 0) + (36 - 13) = (199 - 52) + 199 + 36 = 414 cylinders.

Fig. 6.14: C-SCAN Algorithm



Disk Reliability & RAID

- Disk Reliability = Ability to retain data and support R/W without failure
- Measured by MTBF (Mean Time Between Failures)
- Use of redundant disks improves data reliability significantly
- RAID (Redundant Array of Independent Disks):
  - Uses multiple disks as one logical storage
  - Enhances performance, reliability, or both
- **RAID Levels:**
  - **RAID 0**: Striping, no redundancy → Fast, not reliable
  - **RAID 1**: Mirroring → Reliable, doubles disk use
  - **RAID 2**: Striping + ECC (Hamming code)
  - **RAID 3**: Bit-level striping + single parity disk
  - **RAID 4**: Block-level striping + single parity disk
  - **RAID 5**: Block-level striping + **distributed parity**
  - **RAID 6**: Like RAID 5 + **dual parity** for higher fault tolerance

Disk Reliability & RAID

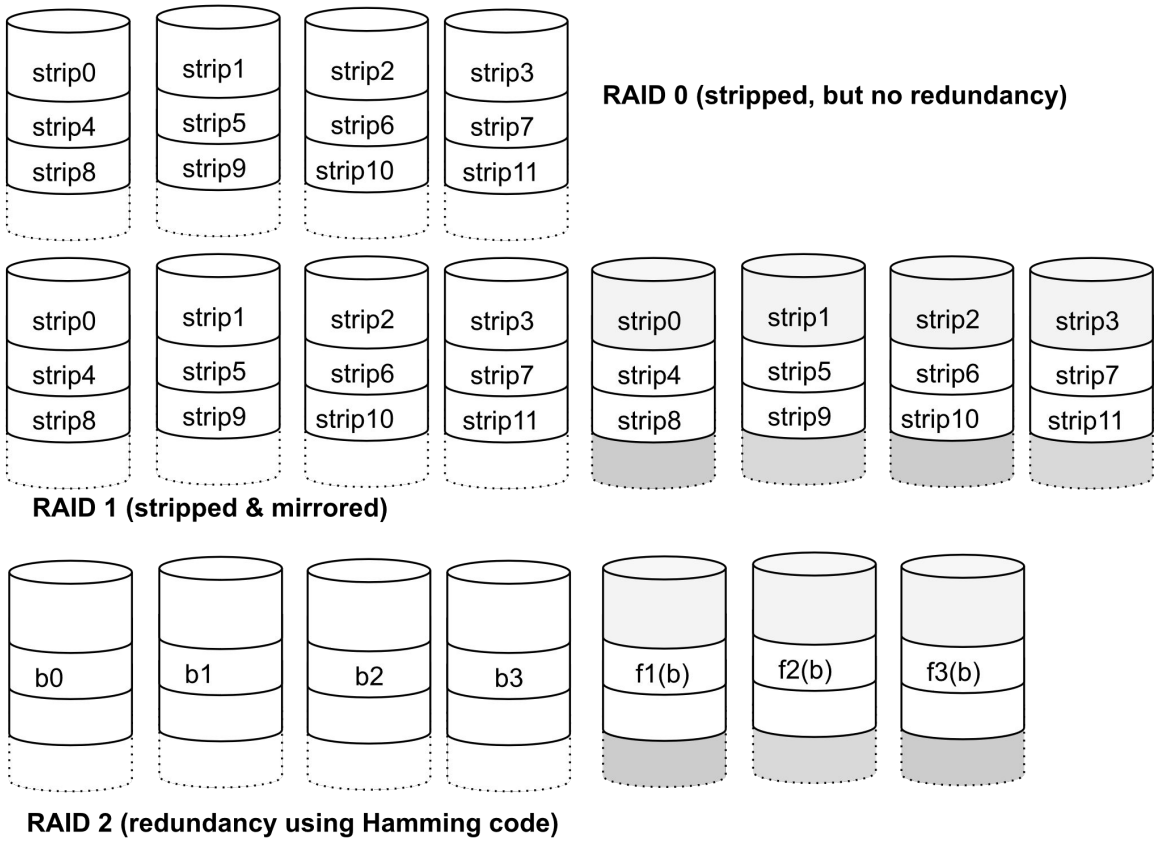


Fig. 6.15: Different RAID Levels (0-2)

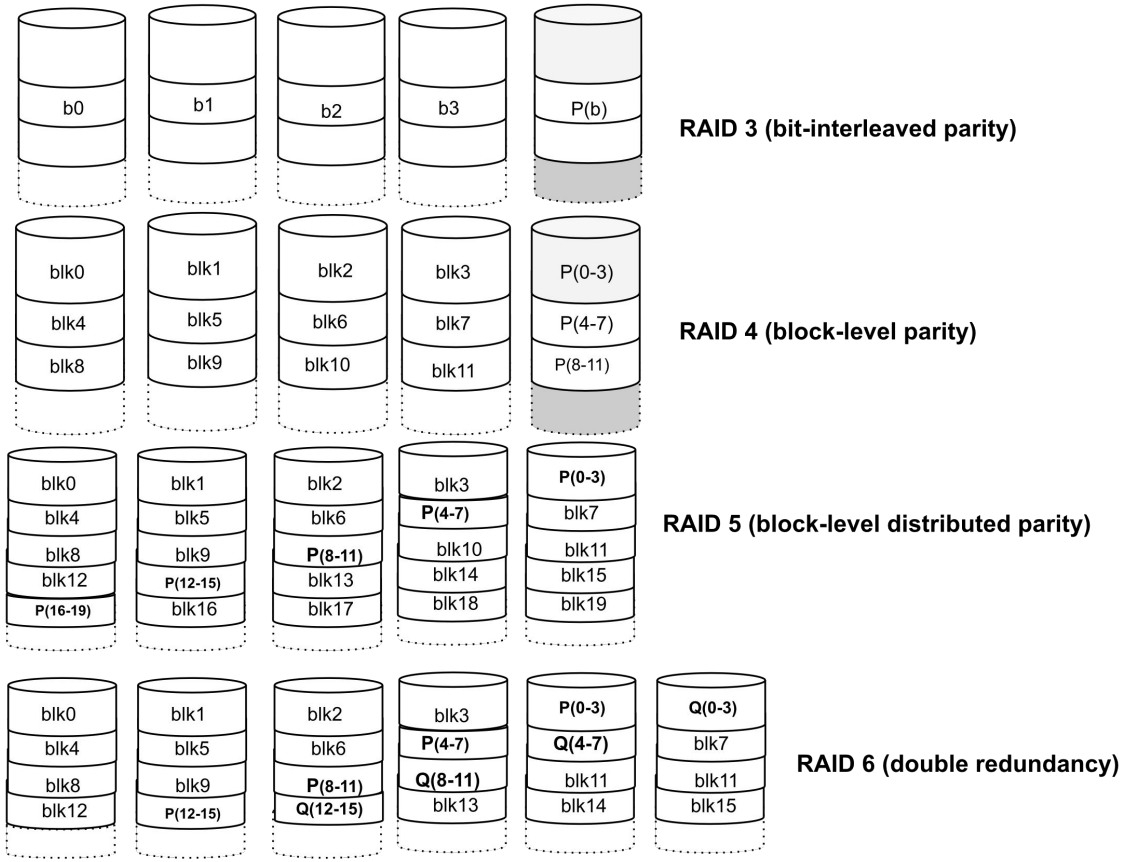


Fig. 6.16: RAID Levels (3-6)

Disk Formatting

- Disks must be formatted before use

1. **Physical Formatting (Low-Level)**
   - Divides disk into tracks & sectors
   - Each sector: Header (ID), Payload (data), Trailer (checksum)
   - Uses:
     - Sector Interleaving: skips sectors for access gap
     - Sector Skewing: aligns track-switching with rotation

Fig. 6.17: Different components of a sector

Disk Formatting: Physical Formatting

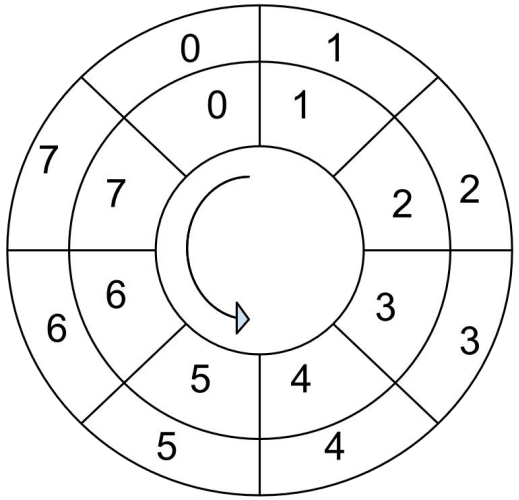


Fig. 6.18: Linear assignment of sector numbers

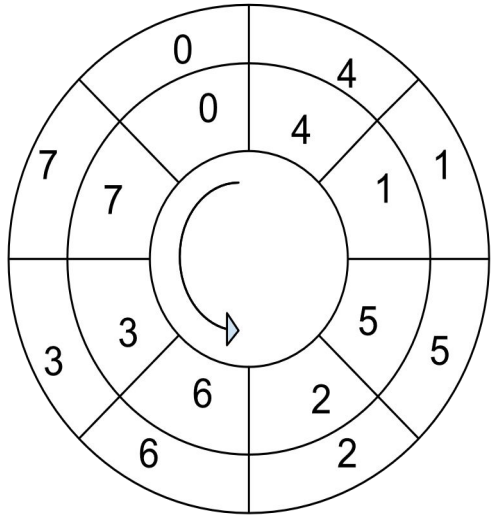


Fig. 6.19: Numbering with one sector interleaving

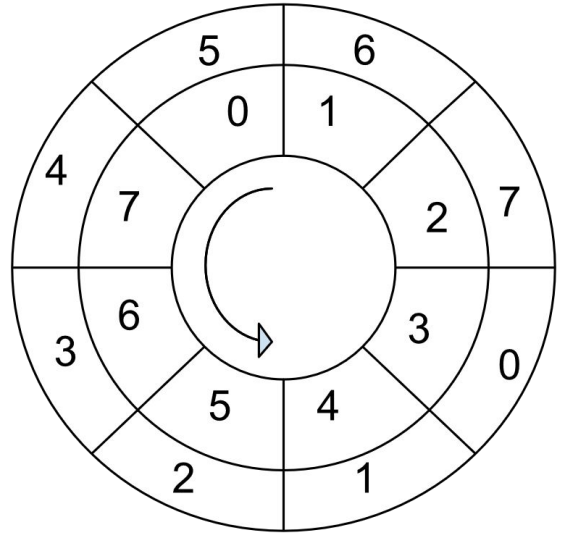


Fig. 6.20: Skew sectoring with skew of 3

Disk Formatting

2. Partitioning

- Divides disk into logical sections (partitions)
- Each partition = mini-disk with boot block
- Improves reliability and OS flexibility

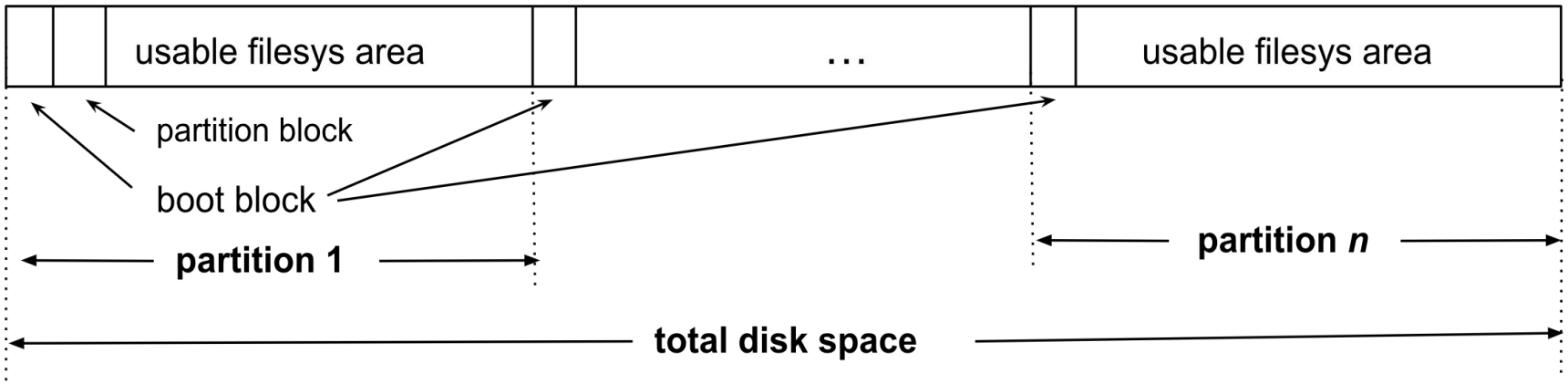


Fig. 6.21: Disk partitions

Disk Formatting

3. Logical Formatting

- File system initialization (e.g., FAT32, EXT3)
- Sets sector size, directory structures, etc.
- Enables OS & apps to access the disk

Boot Block

- **Boot Block:** First block on a secondary storage device
- Stores the boot loader, a small program to load the OS
- Boot Process Overview:
  - POST: Power-On Self-Test checks hardware
  - BIOS: Loads from fixed ROM location; initializes devices
  - Boot Loader (from boot block): Loaded into RAM
    - Contains pointer to OS location on disk
  - OS Loading: Boot loader loads kernel & subsystems
  - Windows Example:
    - Boot block = Master Boot Record (MBR)
    - MBR contains:
      - Boot code
      - Partition table
    - Boot code finds boot partition → loads kernel

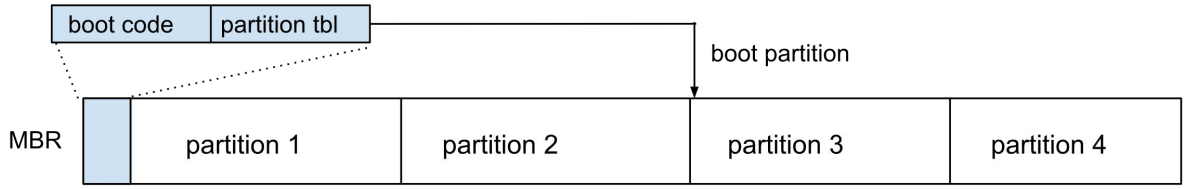


Fig. 6.22: Boot Block in a Windows system and booting from disk

Bad Blocks

- Unusable disk sectors caused by defects or wear
  - May exist from manufacturing or develop during use
- Error Types:
  - **Soft Errors**: Recoverable (e.g., via ECC)
  - **Hard Errors**: Unrecoverable; result in data loss
- **IDE Disks** (manual):
  - Bad blocks marked during formatting or with tools (e.g., badblocks in Linux)
- **Modern Disks** (automatic):
  - Controller maintains list of bad blocks + spare sectors
  - Uses **Sector Sparing** (replace bad with spare sector)
  - Uses **Sector Slipping** (shift mapping to skip bad block)

FILE MANAGEMENT SYSTEM

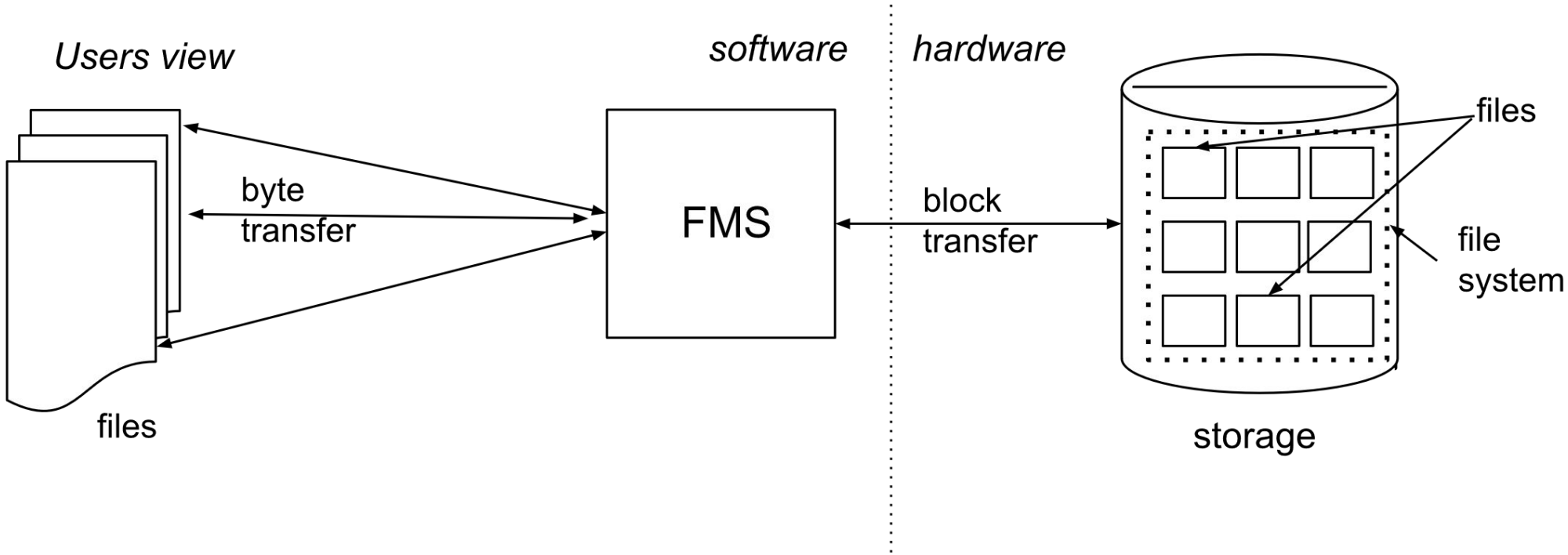


Fig. 6.23: File Management System

File Concepts and Structure

- **File**: Logical unit of data storage independent of physical media.
- Used for input/output by most applications.
- Persistent beyond the lifetime of a process.
- **Logical Structure**:
  - **Field**: Smallest logical unit (e.g., name, ID, date).
  - **Record**: Collection of fields (e.g., employee profile).
  - **File**: Collection of records or fields.
- Users view files as byte streams, unaware of physical block locations.
- **File Properties**:
  - Data type, length, permissible operations
- **Filesystem**:
  - Logical grouping of files based on properties
  - OS supports multiple filesystems (FAT32, EXT4, NTFS, etc.)
  - A file belongs to one filesystem at a time

File Concepts and Structure

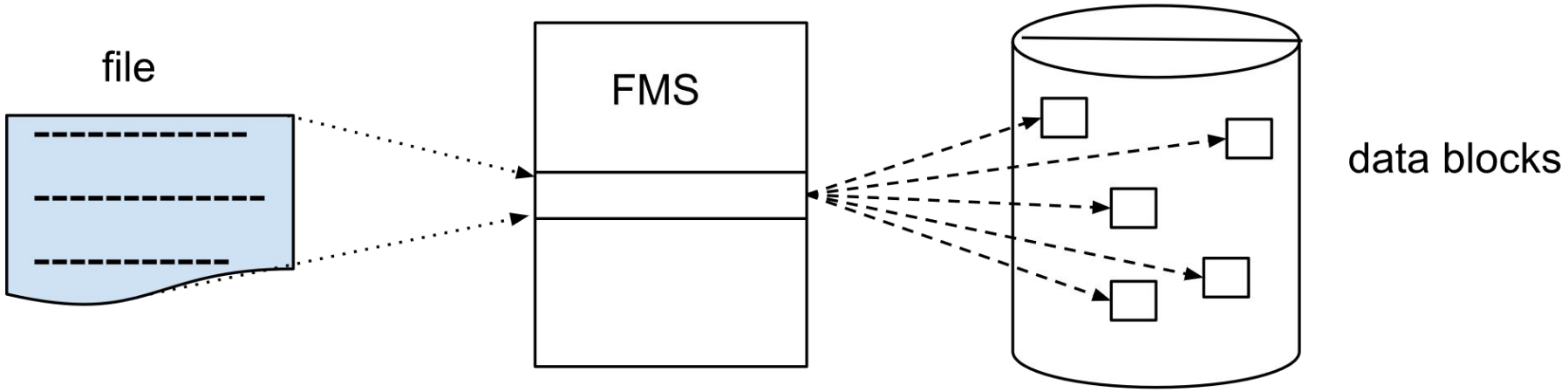


Fig. 6.24: Mapping from a logical file to physical disk blocks

File Access Methods

Sequential Access:

- Follows the tape model: access from beginning to end
- Operations: read_next(), write_next(), rewind()
- Uses a file pointer to track position
- File system handles translation to physical addresses
- Common in editors, compilers, etc.

File Access Methods

Direct Access:

- Follows the disk model: access any block randomly
- Operations: read(n), write(n)
- n = logical/relative block number, translated to physical block
- OS abstracts the mapping from users and apps
- Sequential access is a special case of direct access

File Access Methods

Indexed Access:

- Uses an index file with sorted metadata
- Each index entry points to data in a relative file
- Efficient for searching records
- Two-level indexing for large files
- Similar to hierarchical paging in memory management

File Types

File type	Meaning	Examples of extension
source code	from different programming languages like C, C++, Java, Perl, assembly languages	.c, .java, .pl, .asm
object code	compiled code before linking	.obj, .o
executable	ready to run, loadable program	.exe, .com, .bin
batch/script	command line interpreted code	.bat, .sh
library	libraries or shared objects used in source code	.lib, .a. .so, .dll
markup	textual data with formatting info	.html, .xml, .tex
archive / compress	for compression of files, storage and archives	.zip, .rar, .bzip, .bz2, .tar
word processor	word processor formats	.doc, .docx, .abi, .rtf
image	file for viewing / printing images	.jpeg, .jpg, .gif
multimedia	audio-visual content	.avi, .mov, .mpeg, .mp3, .mp4

Table 6.1: File types

File Operations

Create:

- Initializes file/dir with metadata & attributes
- Allocates space and updates container directory
- Directory creation involves extra setup (e.g., search structures)

Delete:

- Removes entry from container directory
- Frees space if not used by any other process

File Operations

Open:

- Requires filename and access mode (read/write)
- Returns external file descriptor
- OS maintains internal file descriptor and file pointer

Close:

- Takes file descriptor as input
- Releases internal file descriptor, file pointer, and associated resources

File Operations

Reposition:

- Moves file pointer to specific offset
- Applicable to random access files only

Read:

- Copies data from file to buffer
- Updates file pointer to end of read segment

Write:

- Writes string at current file pointer
- May overwrite or append data
- Repositions file pointer after write

File Operations

Truncate:

- Reduces file size to specified byte count
- Frees unused space

Other Operations(OS-dependent):

- Memory mapping
- File locking
- Metadata updates

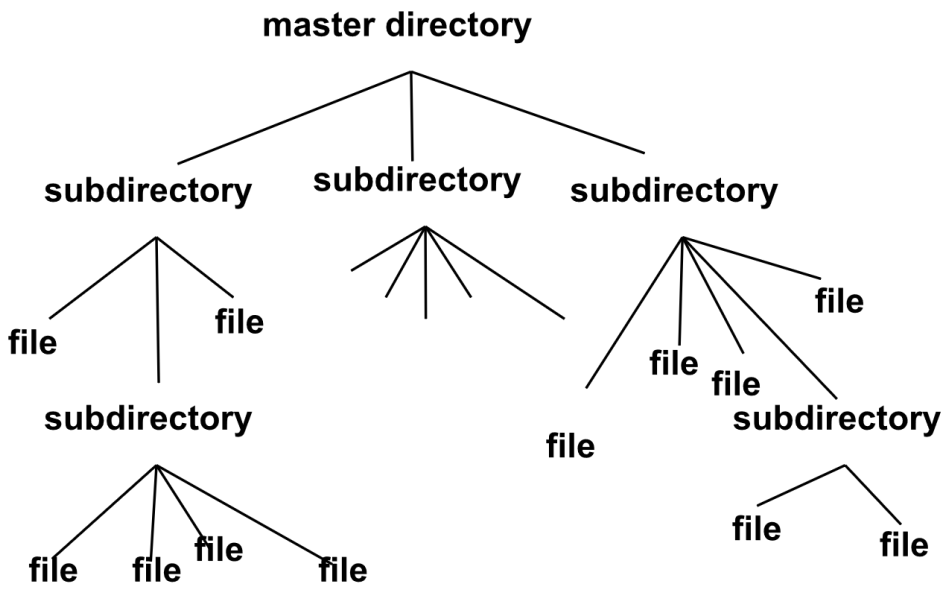
Directory Structure

file name	type & size	location info	protection info	open count	lock	flags	misc info

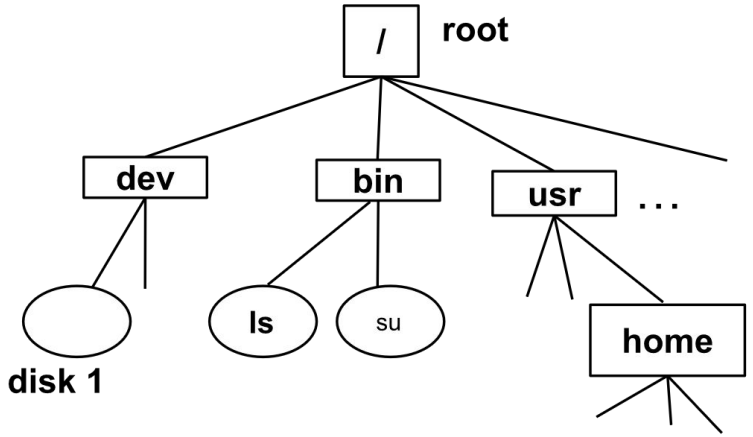
file name: name of the file / subdirectory. Name beyond the allowed length will be truncated
type & size: file type is denoted by the extension like .c, .exe etc, size expressed in bytes
location info: information about the file's location in the disk, expressed in the form of table or linked list of disk blocks
protection info: information about the users having access permission to the file and manner of access
open count: number of processes that opened the file
lock: whether a process is having exclusive access
flags: information about whether the file is a directory, a link, a mounted filesystem or regular file
misc. Info: miscellaneous information like owner of the file, creation time, last modification time etc.

Fig. 6.29: Different fields in an entry of a directory

Directory Structure



(a) Hierarchical Directory Structure



(b) UNIX file organization

Fig 6.30: A popular, flexible, powerful directory structure

File System Structure

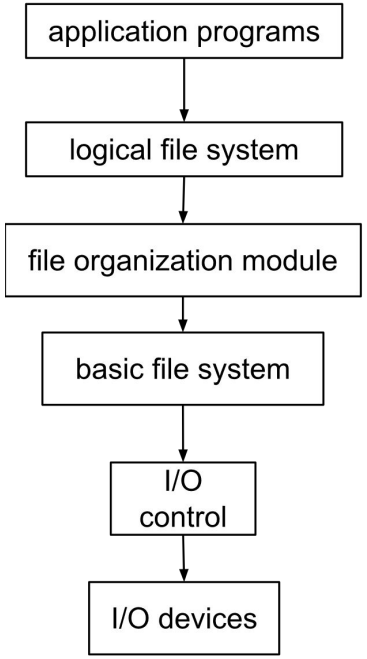


Fig. 6.31: File system layers

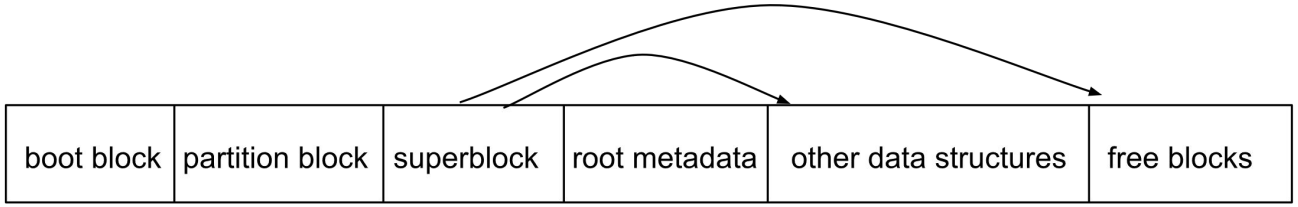


Fig. 6.32: Different blocks of a file system within a disk partition

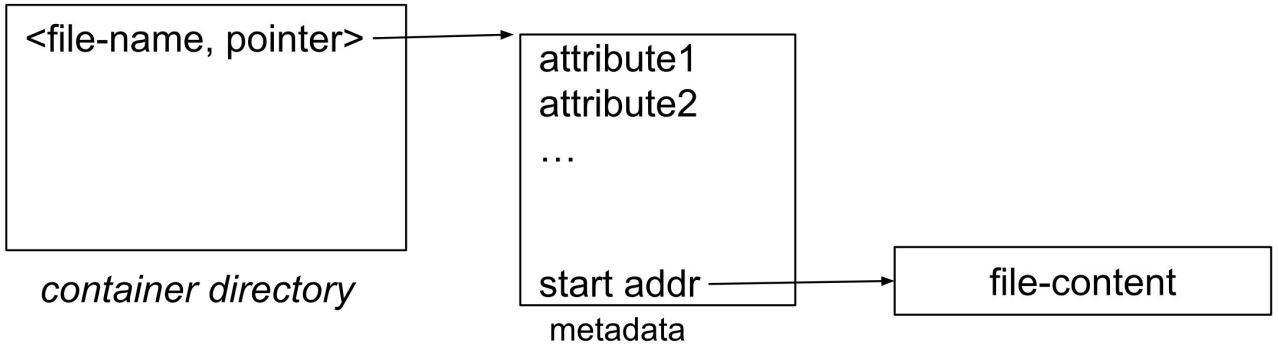


Fig. 6.33: Address resolution across layers in a file system

Contiguous Allocation

- File blocks allocated in continuous physical locations
- Easy to implement; supports random access

Problems:

- File growth may need relocation
- Causes **external fragmentation**
- Suitable for static files with known size

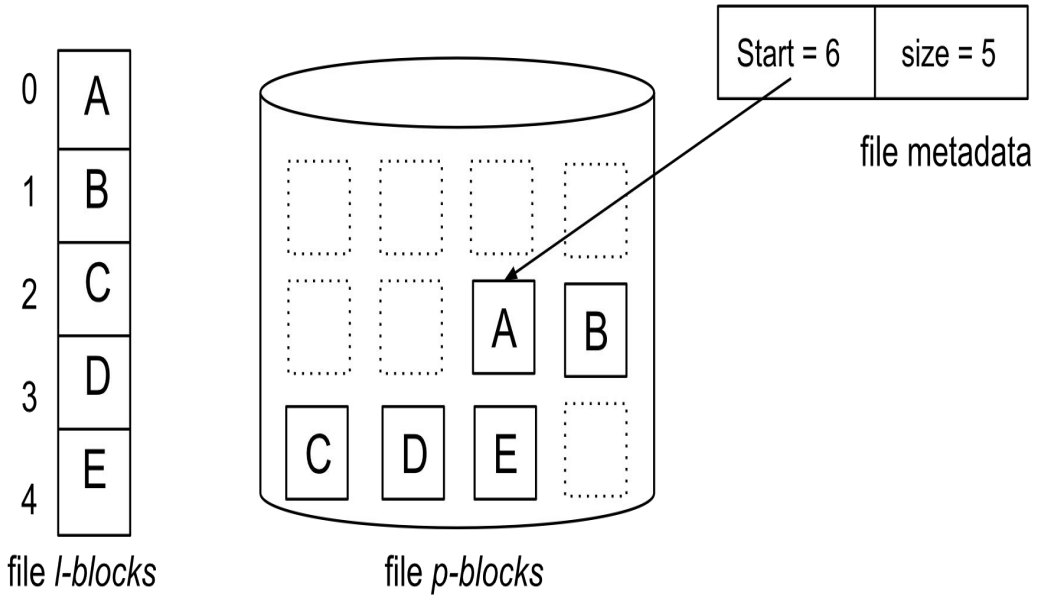


Fig. 6.34: Contiguous Allocation of a file in a storage device

Linked Allocation

- File blocks are linked as a non-contiguous chain
- No external fragmentation; supports dynamic growth

Problems:

- **No random access** — only sequential
- Each block contains a pointer, increasing space overhead
- Link corruption can break the

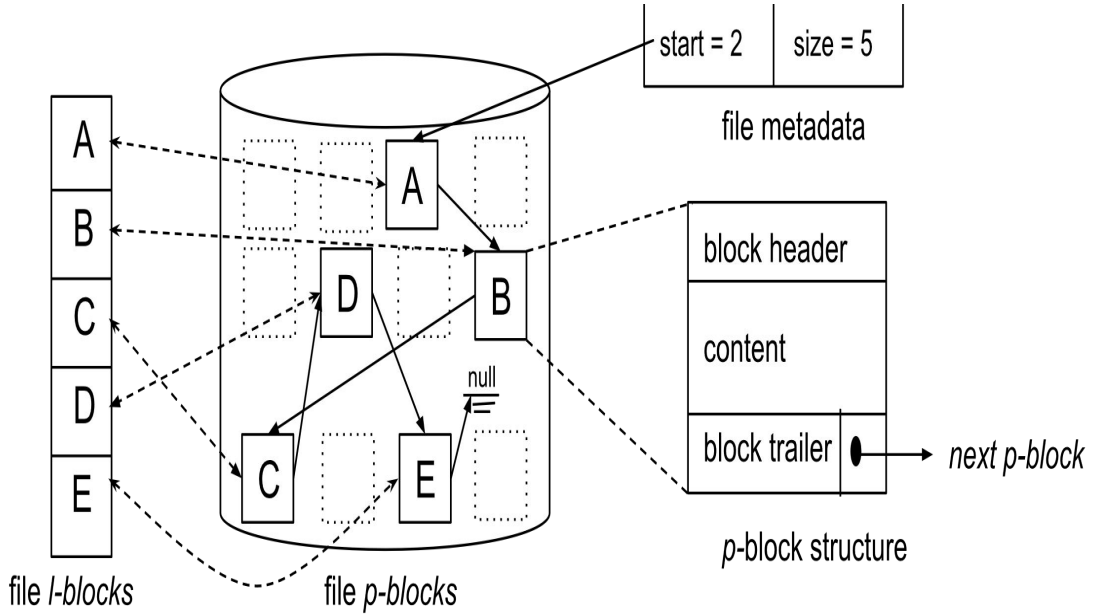


Fig. 6.35: Linked Allocation of a file in a storage device

Indexed Allocation

- Solves issues of contiguous and linked allocation
- Logical blocks (l-blocks) can be mapped to any physical block (p-block).
- Uses two types of blocks:
  - **d-blocks** (data blocks): store file content only
  - **i-blocks** (index blocks): store pointers to d-blocks or other i-blocks
- Last pointer in an i-block may point to another i-block (multi-level index)
- Supports random access with efficient lookup
- No external fragmentation
- Small files may need one i-block; large files use multiple i-blocks

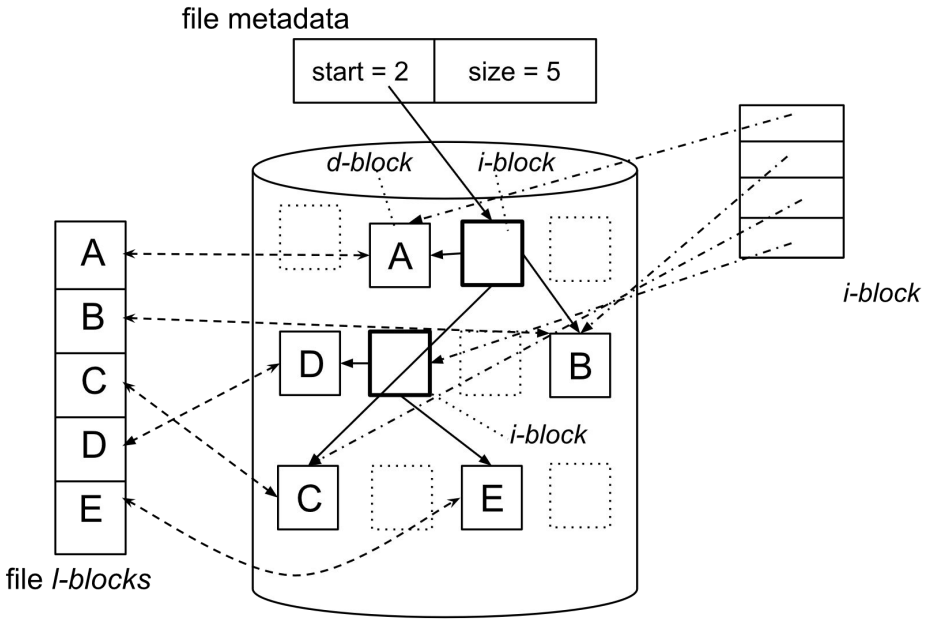


Fig. 6.36: Indexed Allocation of a file in a storage device

Free-space Management

Bit Vector:

- Bitmap (1 = free, 0 = used)
- Easy to implement, fast lookup
- Requires memory for large disks

Linked List:

- Free blocks linked together
- Simple for small requests, slow for large devices

Grouping:

- Groups of free blocks with address pointers
- Faster lookup than standard linked list

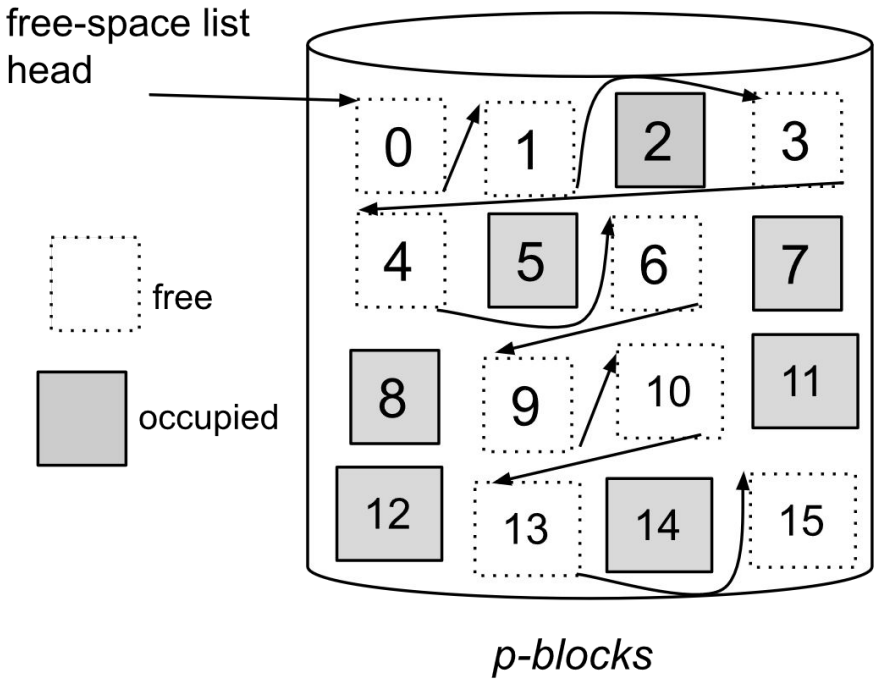


Fig. 6.37: Free space linked list

Directory Implementation

Linear List:

- Files stored in a simple list
- Search is linear in time complexity
- File creation/deletion involves scanning list
- **Optimizations**: sorting, caching, shifting entries
- Complex management; better alternatives exist (e.g., trees)

Hash Table

- File name → hash value → directory entry
- Constant time search, insertion, and deletion
- Problems:
  - Resizing hash table needs rehashing
  - Collisions handled by overflow linked lists.

Reference

[1] “OPERATING SYSTEMS”, Author: Dr. Sukomal Pal Associate Professor Department of Computer Science & Engineering IIT (BHU), Varanasi, UP