Session 14

Storage and Filesystems (UNIX-Like Systems oriented)

OSC – Open Source Community

Table of contents

Whoa!

The question is, what is Storage and filesystems and why we need them?

01

Warm-up

Introduction to Storage in computers and filesystems

Storage why?

All computer applications need to store and retrieve information. Let’s assume that initially, We will use memory (RAM) for this, as Computers only need memory and CPU to do its job.

Where is the problem?

Storage, why?

Limited Storage Capacity

- A process can only use space within its own virtual address space.
- For large applications (e.g., airline reservations, banking), this space is insufficient.

Volatility on Process Termination

- Any data kept in a process's address space disappears when the process ends.
- Unacceptable for applications needing data retention (databases, corporate records).

Storage , why?

Vulnerability to Crashes

- If the system or process crashes, all in-memory data is lost.
- Persistent storage must survive unexpected failures.

Need for Long-Term Persistence

- Many applications require data to remain available for weeks, months, or indefinitely.
- In-memory storage alone cannot meet these requirements.

Concurrent Access by Multiple Processes

- Often necessary for several processes to read or update the same data simultaneously.
- In-process storage does not support safe, coordinated multi-process access.

Why Do We Need Storage Media ?

To serve as effective storage, a device must meet these three requirements:

High Capacity

It must be able to store a very large amount of information.

Persistence

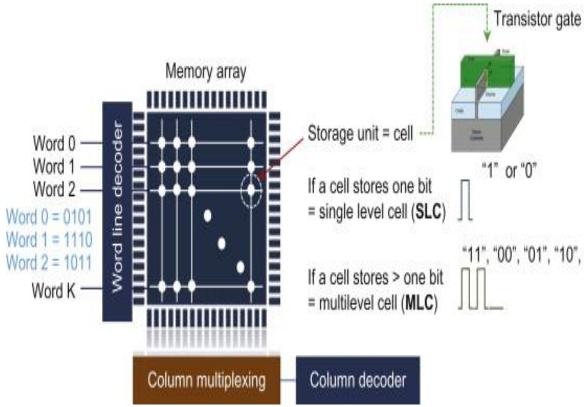
The information must survive after the process ends — even if the computer shuts down.

Concurrent Access

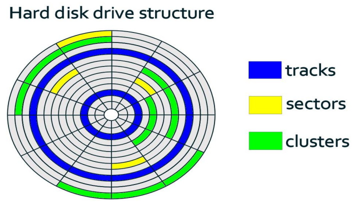
Multiple processes should be able to access the data at the same time.

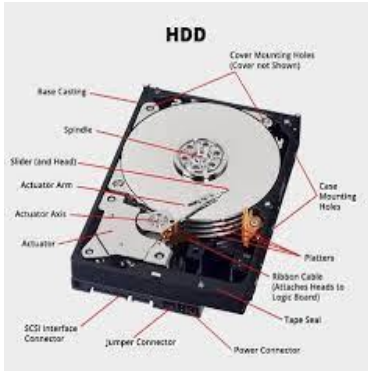
Storage types

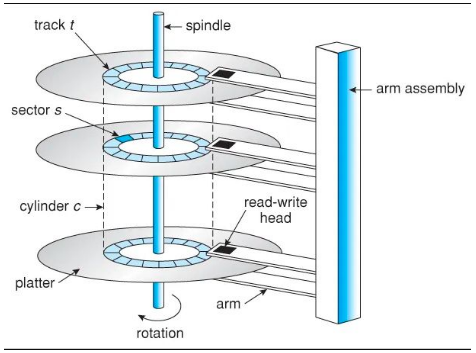
SSD (Solid-state drive)



Hard Disk Drive (HDD)







Sector v.s Data Block

Sector

- The smallest physical storage unit on a disk (usually 512 bytes or 4096 bytes).
- Defined by the disk's hardware and firmware.
- Accessed directly by the disk controller.

Data Block

- A logical unit of storage used by Operating system -filesystem for accuracy- (commonly 4 KB, 8 KB, etc.).
- Groups one or more contiguous sectors together.
- The unit of allocation and I/O in the operating system's file management.

Sectors V.S Data Block

	Sector	Data Block
Level	Hardware-level concept	Software/File-system-level concept
Size	Fixed by disk specifications (commonly 512 bytes or 4096 bytes)	Chosen by the file system (e.g., 1KB, 4KB, 8KB)
Usage	Basic unit for physical disk I/O	Logical unit for file system operations; OS reads/writes in blocks which map to multiple sectors

So, let's define our storage to be a "disk" where it is a linear sequence of fixed-size blocks and supporting mainly two essential operations:

Read Block *k*

It is just getting data from disk blocks/sectors/etc. To the main memory

Write Block *k*

It is just writing the data from main memory to the disk blocks/sectors/etc.

As storage uses these operations for many applications and processes. Here are three important questions:

Search

How do you find information?

User management

How do you keep one user from reading another user’s data?

Controlling

How do you know which data blocks are free?

Hint to answer our three previous arising questions:
File System

But before knowing what is a File System, we must know first ***what is files and how it is related to storage and storage concepts?***

02

Files and Directories

From User’s perspective (boring)

What is a File?

A **file** is a way to **save information on the disk** so we can **use it later**.

It helps us **store data permanently** and **get it back when needed**.

Files: Concept and Purpose

Files concept:A **file** is a way for a program to **store information**. It's created by a process and saved on the disk. Your disk can have thousands or even millions of files, and each file works on its own, without depending on other files.

Approximation (How to Think About a File): You can think of a file like a small storage area, similar to memory (RAM). But instead of being used while the program is running, it stores data on the disk so it can be saved and used later — **even after the program ends.**

Specification (How Files Should Behave): The data inside a file should **not disappear** when a program stops or the computer shuts down. It should stay there until the user decides to delete it. This makes files useful for saving information that we need to keep for a long time.

File structuring

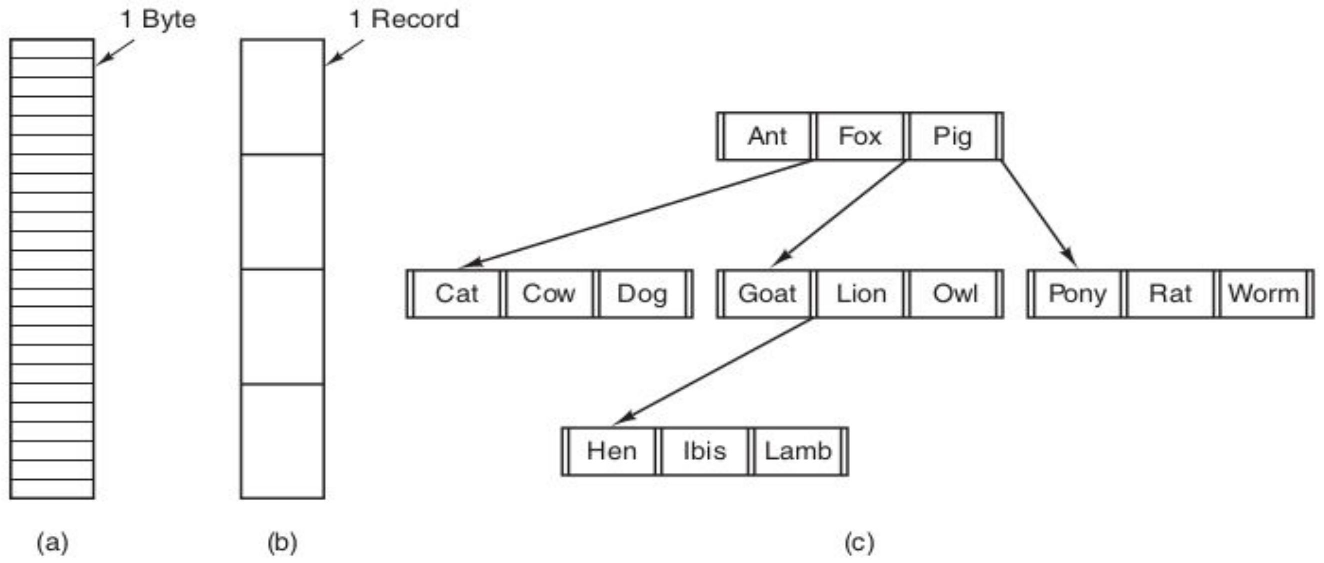


Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

File structuring

Files can be organized in different ways,So **The Previous picture shows three common types of file structures:**

(a) Byte sequence:This is a simple file made of just **a series of bytes**.
The operating system doesn’t care what the bytes mean.It just stores and reads them — the program using the file gives them meaning.Both UNIX and Windows use this type.

(b) Record sequence:In this type, the file is made up of **a list of records**, and each record has a fixed size.This structure is useful when we want to **easily find or update specific pieces of data**, like in a database.
Some old systems and databases use this format.

(c) Tree of records:This is a more **complex structure**, where records are stored in a **tree-like format**.
It allows for **fast searching, sorting, and linking** between pieces of data.
This type is used in some advanced applications or special file systems.

File access

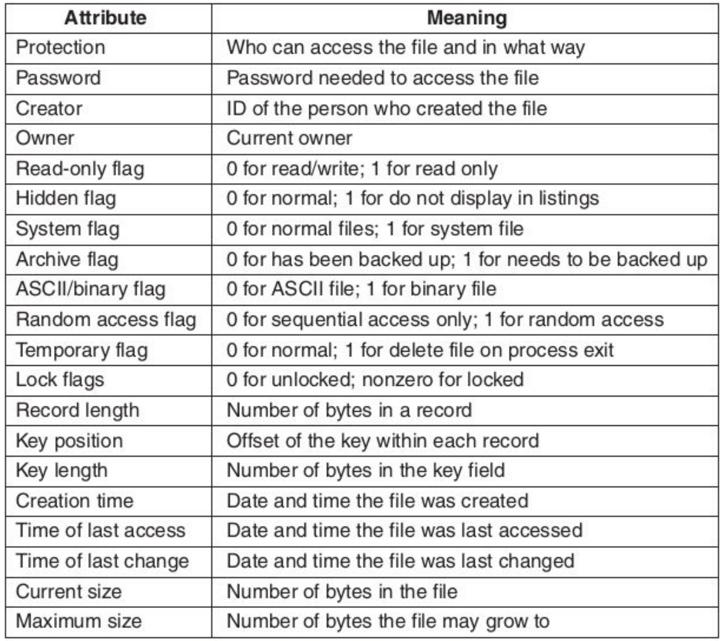
File Access Type	Description
Sequential Access	Early operating systems provided only one kind of file access: sequential access. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files could be rewound, however, so they could be read as often as needed. Sequential files were convenient when the storage medium was magnetic tape rather than disk.
Random Access	When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key rather than by position. Files whose bytes or records can be read in any order are called random-access files. They are required by many applications.

Two methods can be used for specifying where to start reading. In the first one, every read operation gives the position in the file to start reading at. In the second one, a special operation, seek, is provided to set the current position. After a seek, the file can be read sequentially from the now-current position. The latter method is used in UNIX and Windows.

File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, We will call these extra items the file's attributes. Some people call them **metadata**.

The list of attributes varies considerably from system to system. The table of Fig. 4-4 shows some of the possibilities, but other ones also exist. **No existing system has all of these**, but each one is present in some system.



Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 4-4. Some possible file attributes.

Directories

- ❑ To keep track of files, file systems normally have **directories** or folders, **which are themselves files**

- ❑ Directories have two systems:
  - ❑ Single-Level Directory system
  - ❑ Hierarchical Directory Systems

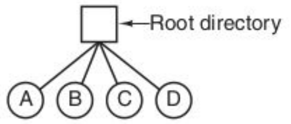


Figure 4-6. A single-level directory system containing four files.

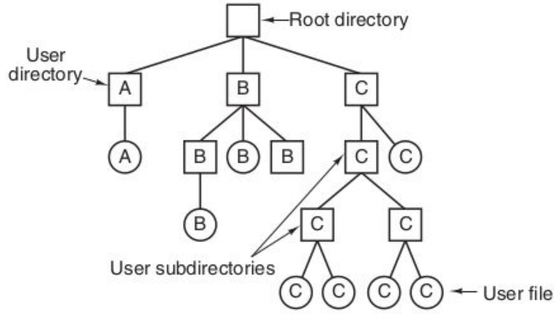


Figure 4-7. A hierarchical directory system.

03

Partitioning and Filesystem Layout

A Talk about partitioning and implementation of filesystems

Filesystem Layout

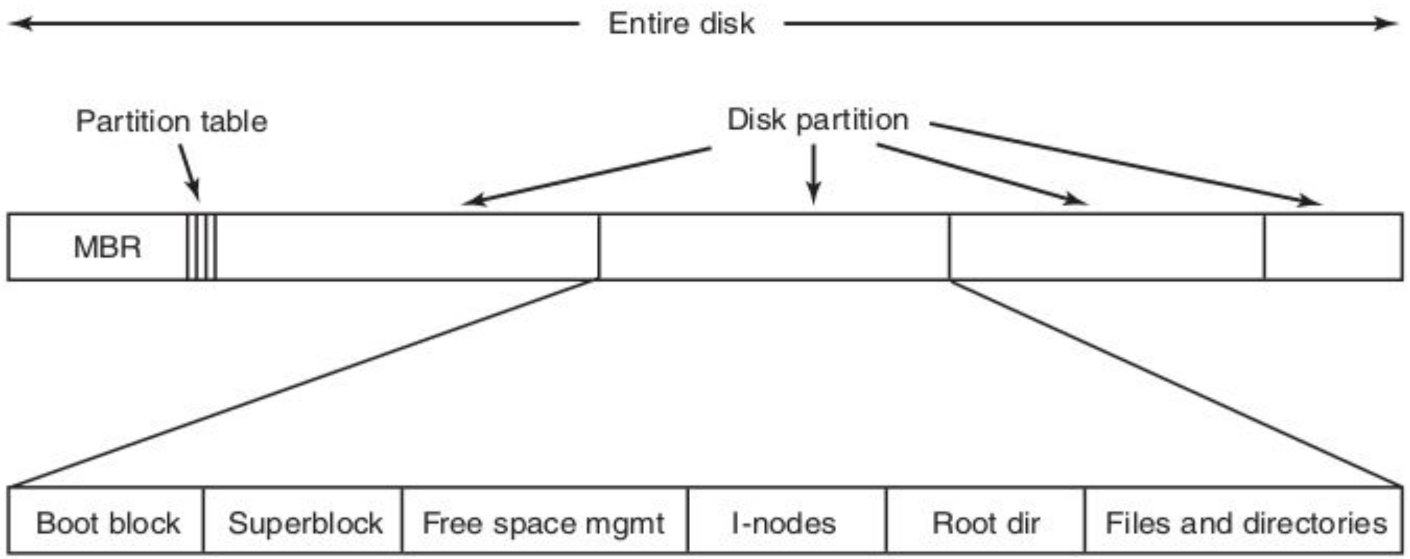


Figure 4-9. A possible file-system layout.

What Is a Partition?

- ❑ A **disk** can be divided into **partitions**.
- ❑ Each partition can have its **own file system**.
- ❑ The very first sector on the disk is called the **MBR (Master Boot Record)**.
- ❑ The MBR contains:
  - ❑ A small program to **start the computer**.
  - ❑ A **partition table** that stores the start and end of each partition.
- ❑ One partition is marked as **active**, meaning it's used to boot the system.

Booting the System

❑ **When the computer starts:**

1. **BIOS** runs and loads the **MBR**.
2. The MBR finds the **active partition**.
3. It loads and runs the **boot block** from that partition.
4. The boot block loads the **operating system**.

❑ Every partition has a **boot block** — even if it doesn't have an OS yet.

What Is a Filesystem?

- The **file system** is part of the OS that **manages files**: How files are **named**, **stored**, **read**, **written**, and **protected**.
- Each partition has its **own file system**.
- The **structure of a file system** may include:
  - **Boot block**
  - **Superblock** – stores file system info
  - **Free space management**
  - **I-nodes** – metadata about files
  - **Root directory**
  - **Actual files and directories**

Filesystem Layout (Based on the Figure)

Based on the figure of the filesystem layout here are brief labels :

- **MBR:** Contains boot loader and partition table
- **Disk Partition:** Contains the actual file system
- Inside Partition:
  - **Boot Block** – Starts OS
  - **Superblock** – File system info
  - **Free Space Management** – Tracks unused space
  - **I-nodes** – Info about each file (permissions, size, etc.)
  - **Root Directory** – Entry point of the file system
  - **Files and Directories** – Actual user data

File Systems Timeline

1960s-1970s

IBM FMS,DMF and
Microsoft FAT-12,16 and
UNIX Filesystem

1980s-1990s

Mac OS HFS and
FAT-32 and
Linux ext filesystem

1990s-2000s

NTFS,XFS,JFS,Ext2

2000s-2010s

ReiserFS,ex3,ZTRFS,ZFS

04

Files and Directories Implementation

Ideas of implementing Files and Directories

Files Implementation Methods

01 Contiguous Allocation

02 Linked-List Allocation

03 Linked-List Allocation Using a Table in Memory

04 I-nodes

Implementing Files : Contiguous Allocation

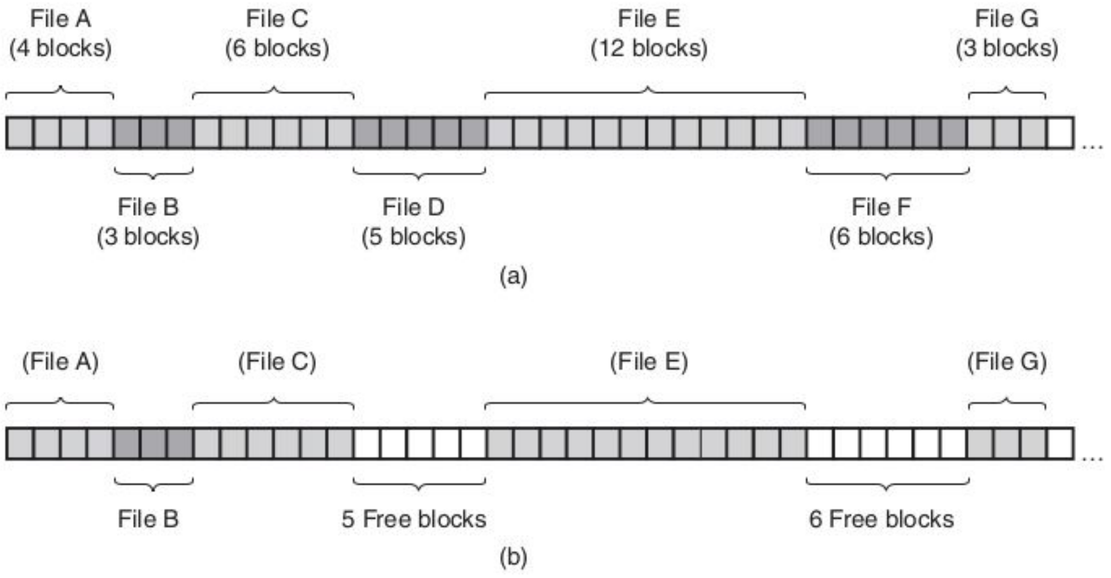


Figure 4-10. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

Files Implementation: Contiguous Allocation

It has two advantages:

- ❑ It is **simple to implement** because keeping track of where a file's blocks are is reduced to remembering two numbers:
    - a) the disk address of the first block
    - b) number of blocks in the file

- ❑ The read performance is excellent because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block).

Files Implementation : Contiguous Allocation

But Unfortunately, It has a very serious drawback:

- ❑ **Disk Fragmentation Over Time:** As files are created and deleted, free spaces ("holes") appear between allocated blocks.
- ❑ **Example Situation:** Files **D** and **F** are removed, leaving contiguous runs of free blocks on the disk.
- ❑ **Why No Immediate Compaction?**
  - ❑ Moving all subsequent blocks to close the hole could involve copying millions of blocks.
  - ❑ Such an operation on large disks can take hours or even days.
- ❑ **End Result:** The disk layout becomes a mix of allocated file blocks and free "holes," leading to fragmentation.

But Why fragmentation is a problem?

Files Implementation : Contiguous Allocation

Initially, this fragmentation is not a problem so when it is a problem?

1. **Initial Smooth Operation**

    - ❏ New files are simply appended at the end of the disk, following existing data.
    - ❏ fragmentation is not a problem until available end-of-disk space is exhausted.

2. **When Disk Becomes Full, problem begins**

- **Compaction**
    1. i. Shifting all subsequent blocks to close holes.
    2. ii. Prohibitively expensive on large disks (takes hours or days to be done).
- **Hole Reuse**
    - ○ Maintain a free-list of holes created by deleted files.
    - ○ On file creation, select a hole large enough to hold the file.
    - ○ **Key Requirement**: The file's final size must be known in advance.

Practical Exception: CD-ROM File Systems

- All file sizes are predetermined and immutable.
- Enables stable, contiguous allocation without fragmentation or compaction.

Files Implementation : Linked-List Allocation

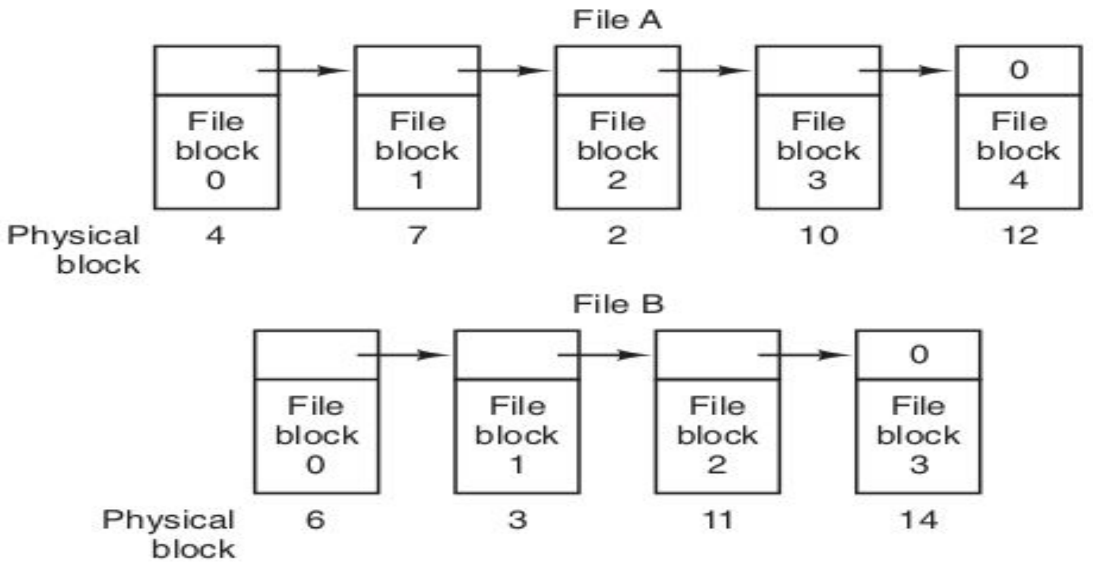


Figure 4-11. Storing a file as a linked list of disk blocks.

Files Implementation : Linked-List Allocation

Unlike contiguous allocation method:

- ❑ Every **disk block can be used** in this method.
- ❑ No space is lost to disk fragmentation (except for internal fragmentation in the last block).
- ❑ It is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there.

Files Implementation: Disadvantages Of Linked-List Allocation

- ❑ **Random access is extremely slow :**To get to block n, the operating system has to start at the beginning and read the n − 1 blocks prior to it, one at a time

- ❑ **Block Size Inefficiency**
  - ❑ Actual usable data per block shrinks because a few bytes at the start store a pointer to the next block.
  - ❑ Block size ceases to be a neat power of two.
  - ❑ Many applications expect—and optimize for—power-of-two block sizes.
  - ❑ Non-standard block sizes force programs to split I/O across block boundaries.

Files Implementation: Disadvantages Of Linked-List Allocation

- **Overhead from Pointer Chaining**
  - To read one “logical” block, the system must:
    - Read the remainder of the first disk block (after the pointer).
    - Read the next disk block to fetch the remaining data.
    - Concatenate the two buffers.
  - Each extra read and copy operation adds latency and CPU overhead.

Files Implementation: Linked-List Allocation Using a Table in Memory, File Allocation Table (FAT)

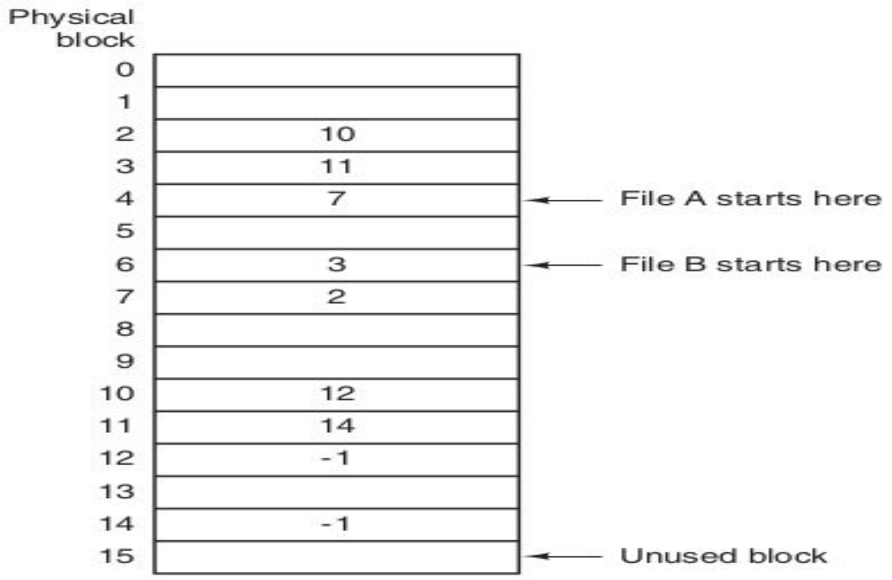


Figure 4-12. Linked-list allocation using a file-allocation table in main memory.

Files Implementation: Advantages of File Allocation Table (FAT)

- The **entire block is available** for storing data (no metadata like pointers inside blocks).
- **Random access** is much easier:
  - Although the block chain must still be followed,
  - The entire **chain resides in memory**, avoiding disk accesses during traversal.
- The **directory entry** only needs to store a **single integer**:
  - The **starting block number**.
  - Still allows access to the **entire file**, regardless of its size.

Files Implementation: Disadvantages of File Allocation Table(FAT)

- ❑ The **entire FAT must remain in memory** for efficient access.
- ❑ Example scenario:
  - ❑ Disk size: **1 TB**
  - ❑ Block size: **1 KB**
  - ❑ Number of blocks: **1 billion (10^9)**
  - ❑ Entry size:
    - ❑ Minimum: **3 bytes** → Total memory = **3GB**
    - ❑ Preferred (faster lookup): **4 bytes** → Total memory = **4GB**

Files Implementation: Disadvantages of File Allocation Table(FAT)

- ❏ This is **not practical** for modern large disks.
- ❏ Shows that **FAT does not scale well** for large storage systems.
- ❏ **Historical Note:**
  - ❏ FAT was the **original MS-DOS file system**. Still **supported in all versions of Windows**.

Files Implementation: I-Nodes

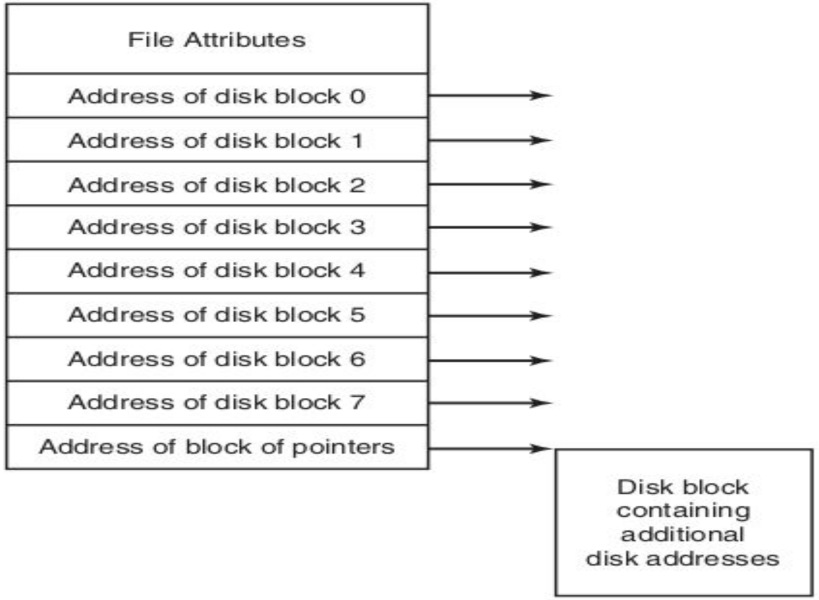


Figure 4-13. An example i-node.

Files Implementation: I-Nodes

I-Node Scheme – Memory Efficiency

- **Open-file requirement:**
  - An **inode** need be in memory **only while its file is open**.
  - If each inode occupies ***n* bytes**, and a maximum of ***k* files** may be open simultaneously, the total memory is just ***k·n* bytes**.
- **Compact in-memory array:**
  - The **array of open i-nodes** is **kn bytes** in size.
  - This reserved space is typically **much smaller** than a full-disk block table.
- **Scalability advantage:**
  - A FAT-style table grows in proportion to **total disk blocks (*n*)**.
  - In contrast, the i-node array grows only with **maximum open files (*k*)**.
  - **Disk size** (100 GB, 1 TB, 10 TB) **does not affect** the inode memory footprint.
- **Key takeaway:**
  - The i-node scheme offers **significant memory savings** for large disks by limiting in-memory structures to **only currently open files**.

Files Implementation: I-Nodes

Extending i-Node Addressing for Large Files

- **Direct-pointer limit:**
  - Each i-node contains a fixed number of **direct disk-block addresses**.
  - Once a file’s size exceeds this capacity, no more direct pointers are available.
- **Single-indirect block:**
  - **Reserve the final pointer** in the i-node for the address of an **indirect block**.
  - This indirect block is **filled with additional disk-block addresses**, effectively extending the file.
- **Multi-level indirection:**
  - **Double indirect:** A block of pointers to **other indirect blocks**, each of which holds data-block addresses.
  - **Triple indirect:** A pointer to a block that points to blocks of indirect blocks—ideal for **very large files**.

Directories Implementation

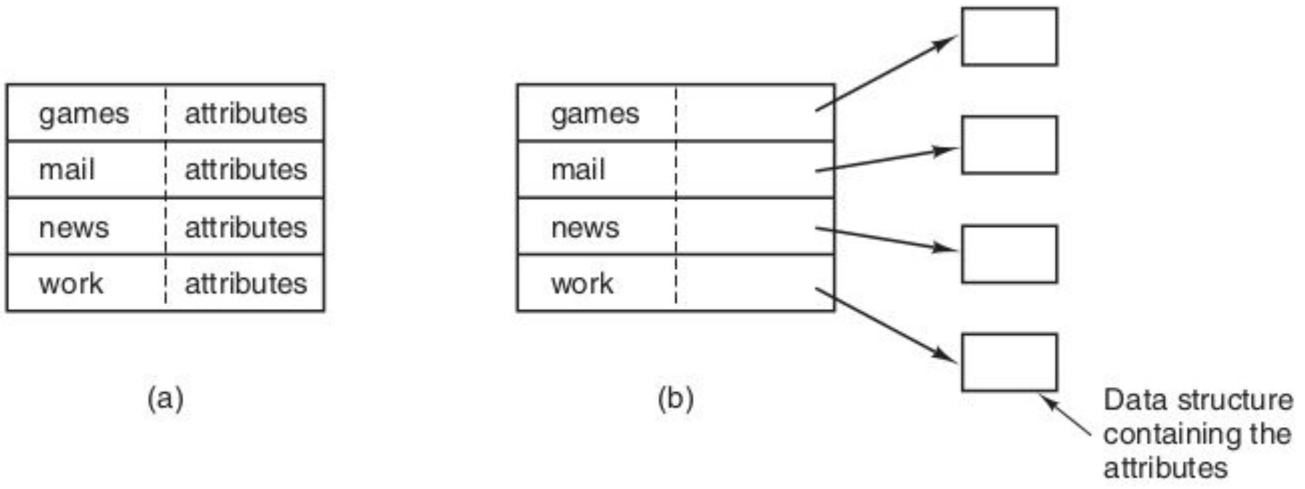


Figure 4-14. (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

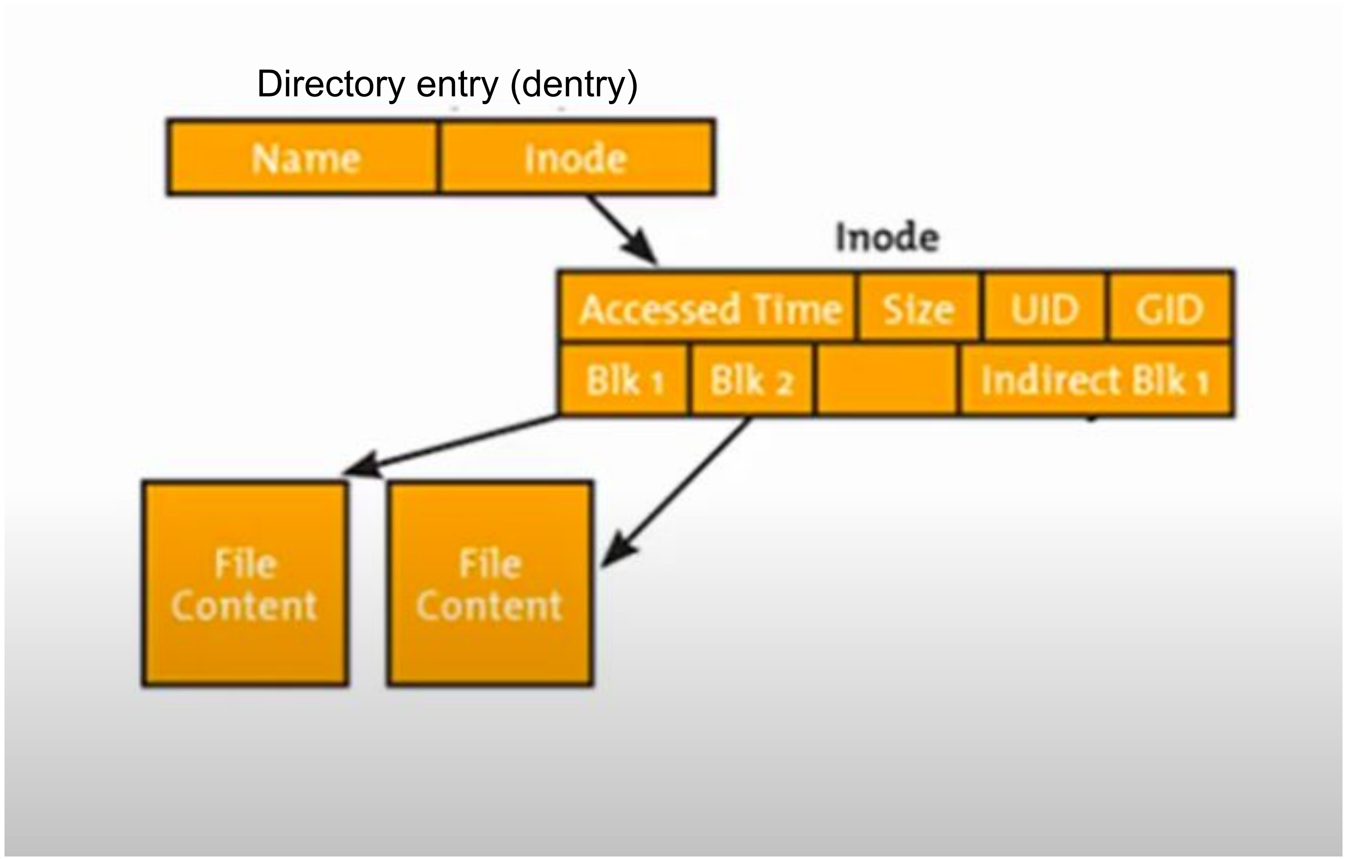
Implementing Directories

- I-node for a directory
  - Just like a regular file’s i-node, a directory’s i-node stores metadata (permissions, owner, timestamps) and pointers to its data blocks.
  - One bit in the i-node’s mode field flags “this is a directory” rather than a plain file.

- **Directory data blocks = array of entries**

Those data blocks aren’t arbitrary bytes—they’re laid out as a sequence of directory-entry records. Each record contains at least:

1. the i-node number of the file/subdirectory
2. the length of this record
3. the length of the filename
4. the filename itself


Directory entry (dentry)
Name
Inode
Inode
Accessed Time
Size
UID
GID
Blk 1
Blk 2
Indirect Blk 1
File Content
File Content

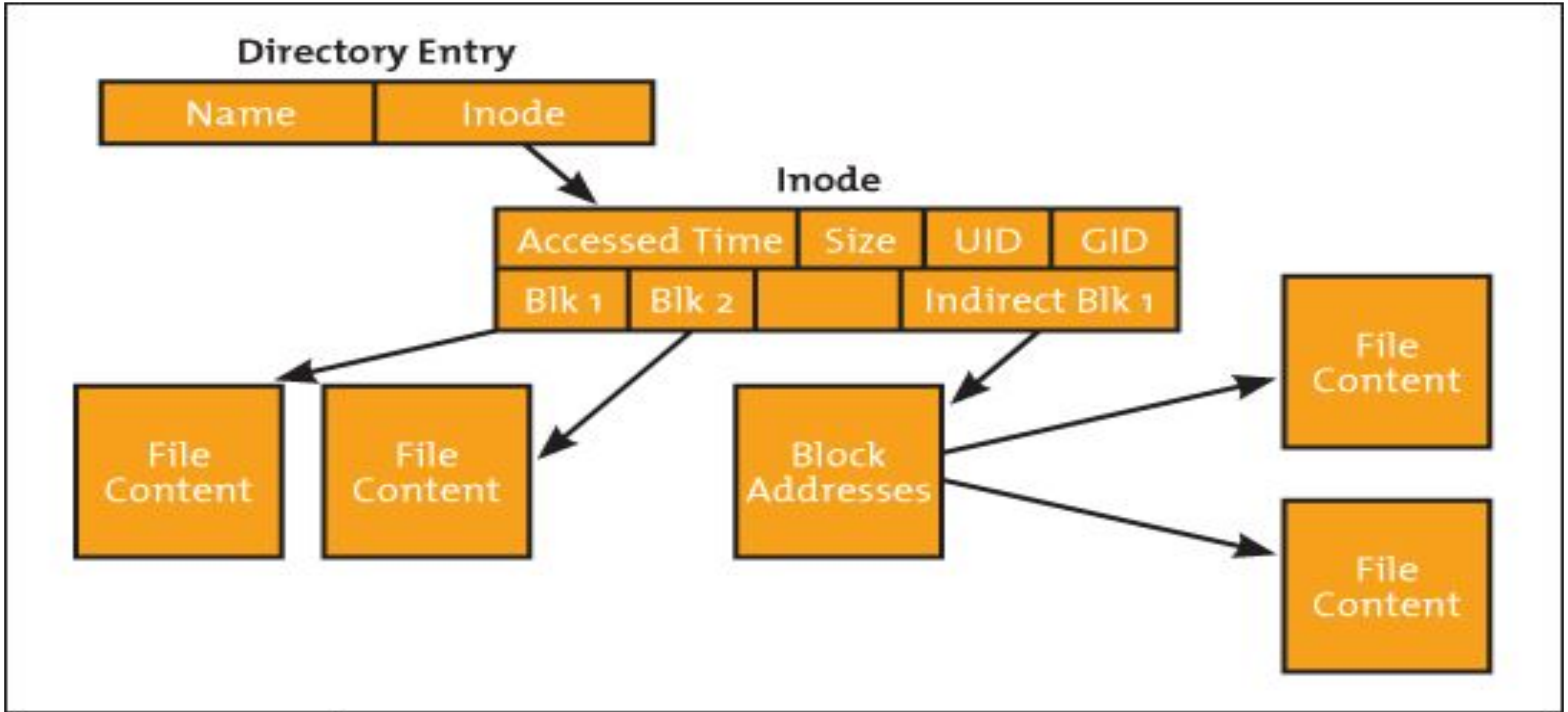
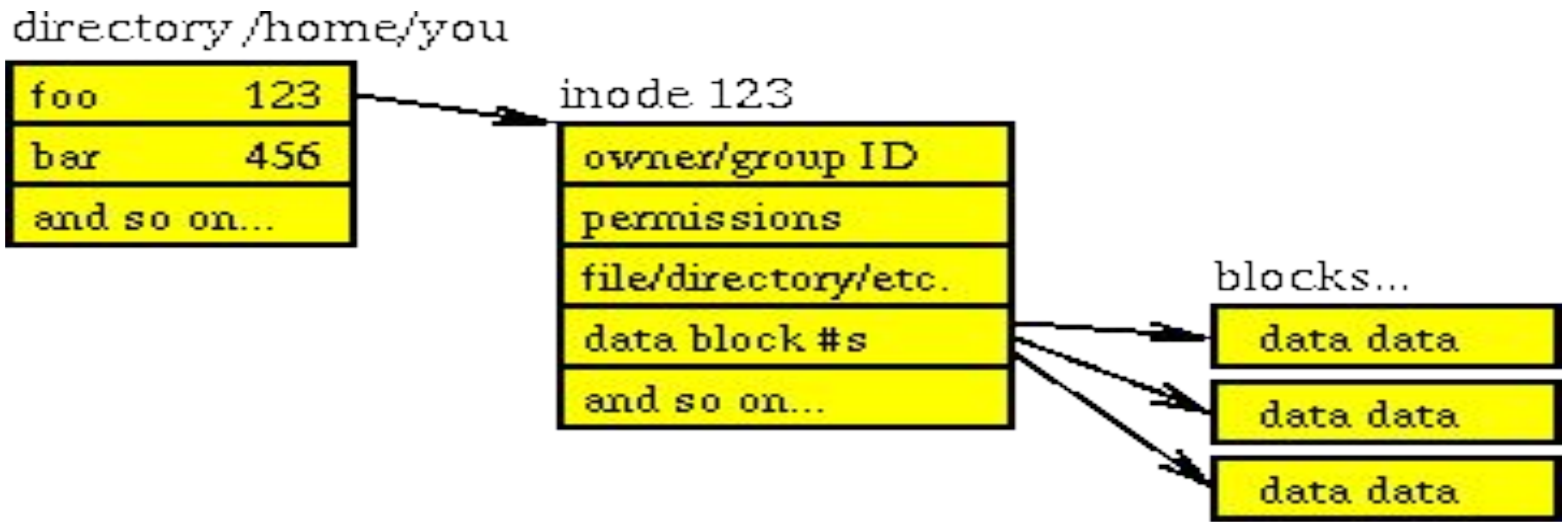


FIGURE 1 RELATIONSHIP BETWEEN THE DIRECTORY ENTRY, AN INODE, AND BLOCKS OF AN ALLOCATED FILE

Regular User: from GUI choose file!
But,how can this happens internally in the Computer


directory /home/you
foo 123
bar 456
and so on...
inode 123
owner/group ID
permissions
file/directory/etc.
data block #s
and so on...
blocks...
data data
data data
data data

File Naming

File Naming Conventions Across File Systems

- ❑ **Name length limits:**
  - ❑ **Classic rule:** 1–8 characters (e.g., `andrea`, `bruce`, `cathy`)
  - ❑ **Modern filesystems:** Up to **255 characters** (e.g., long descriptive names)
- ❑ **Allowed characters:**
  - ❑ **Letters** (A–Z, a–z)
  - ❑ **Digits** (0–9)
  - ❑ **Special characters** (e.g., `!`, `_`, `-`, space in some systems)

File Naming

File Naming Conventions Across File Systems

- **Case sensitivity:**
  - **Case-sensitive:**
    - UNIX, Linux:
      - `maria`, `Maria`, and `MARIA` are **three distinct files**
  - **Case-insensitive:**
    - MS-DOS, Windows FAT:
      - `maria`, `Maria`, and `MARIA` all refer to the **same file**
- **Historical note:**
  - Although **ancient**, MS-DOS conventions (8.3 names, case-insensitive) are still widely used in **embedded systems**.

File Naming, cont.

Structure:

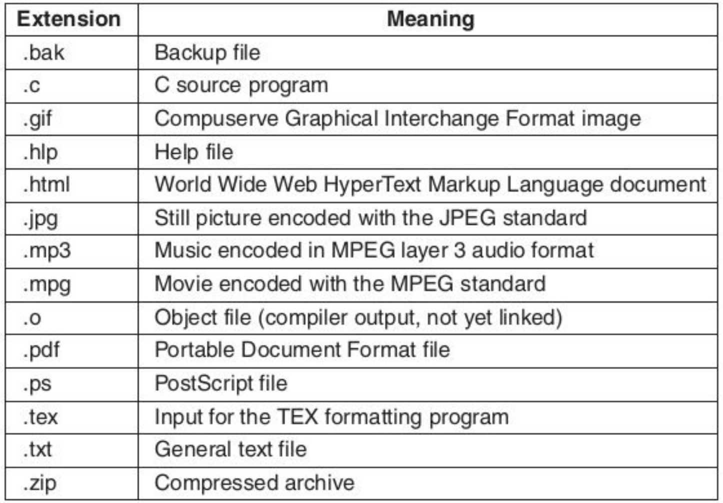
- **Base name** + `.` + **extension** (e.g., `prog.c`)

MS-DOS (8.3 format):

- **Base name:** 1–8 characters
- **Extension:** optional, 1–3 characters

UNIX/Linux:

- **No fixed limits** on extension length
- **Multiple extensions** allowed (e.g., `homepage.html.zip`)
  - `.html` → HTML Web page
  - `.zip` → ZIP-compressed archive



Extension	Meaning
.bak	Backup file
.c	C source program
.gif	Compuserve Graphical Interchange Format image
.hlp	Help file
.html	World Wide Web HyperText Markup Language document
.jpg	Still picture encoded with the JPEG standard
.mp3	Music encoded in MPEG layer 3 audio format
.mpg	Movie encoded with the MPEG standard
.o	Object file (compiler output, not yet linked)
.pdf	Portable Document Format file
.ps	PostScript file
.tex	Input for the TEX formatting program
.txt	General text file
.zip	Compressed archive

Figure 4-1. Some typical file extensions.

File Naming, cont.

File Extensions: Convention vs. Enforcement

- **OS-level behavior (e.g., UNIX/Linux):**
  - File extensions are **purely conventional**.
  - The operating system **does not enforce** or interpret extensions.
- **User convenience:**
  - Extensions (like `.txt`, `.jpg`, `.c`) serve as **reminders** to humans about file contents or intended use.
- **Application-level enforcement:**
  - Specific programs **may require** particular extensions:
    - A C compiler might **refuse** to compile files unless they end in `.c`.
    - Archive utilities often look for `.zip`, `.tar`, etc., to determine format.
- **Key takeaway:**
  - **Extensions don't affect** the OS itself, but can be **critical** for individual tools and workflows.

File Types

File Type	Description
Regular File	Regular files are the ones that contain user information. All the files of Fig. 4-2 are regular files
Directory	Directories are system files for maintaining the structure of the file system.
Character Special File	Character special files are related to input/output and used to model serial I/O devices, such as terminals, printers, and networks.
Block Special File	Block special files are used to model disks.

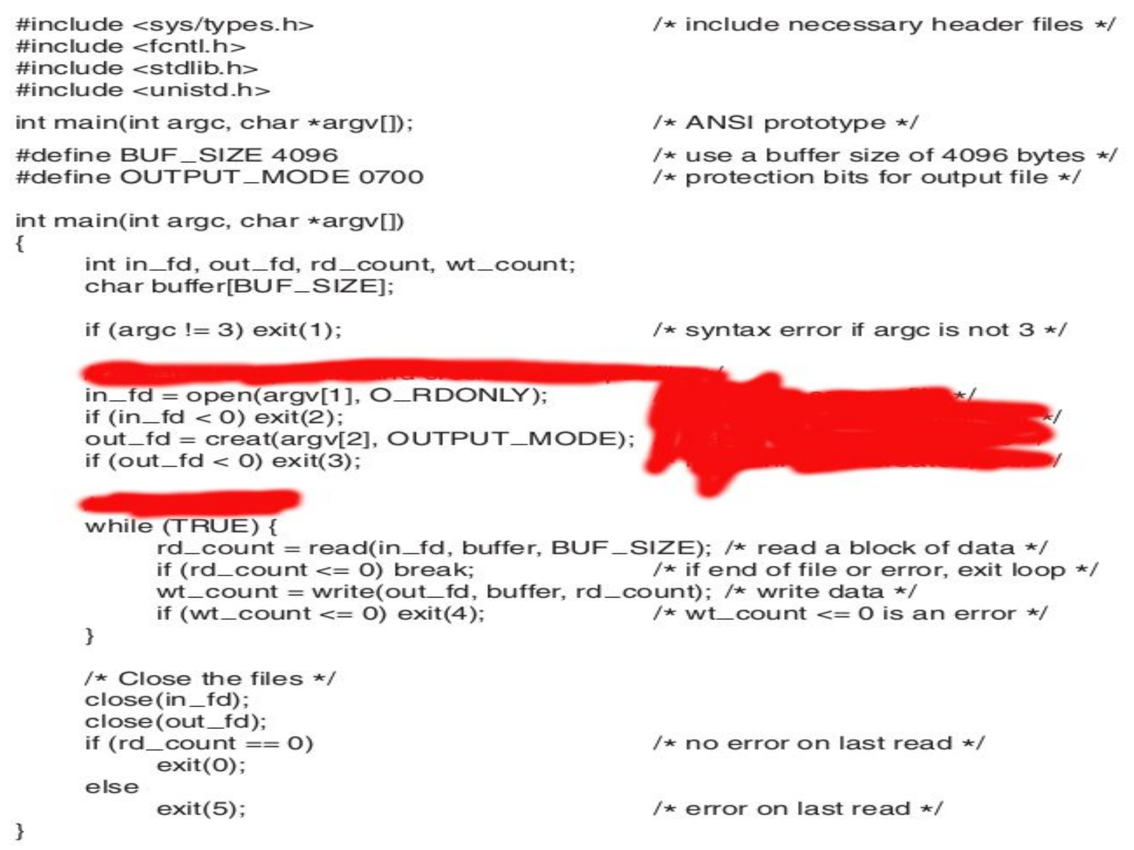
File Operations

Operation	Description
Create	The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
Delete	When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
Open	Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
Close	When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file’s last block, even though that block may not be entirely full yet.
Read	Data are read from file. Usually, the bytes come from the current position. The caller must specify how many data are needed and must also provide a buffer to put them in.
Write	Data are written to the file again, usually at the current position. If the current position is the end of the file, the file’s size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.

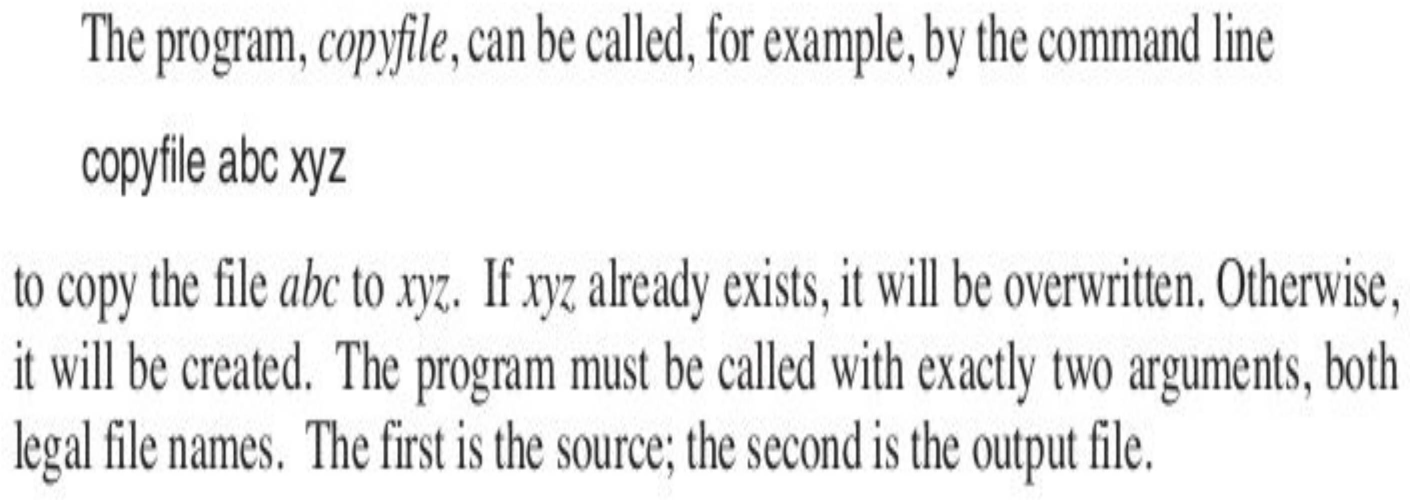
File Operations

Operation	Description
Append	This call is a restricted form of write. It can add data only to the end of the file. Systems that provide a minimal set of system calls rarely have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.
Seek	For random-access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.
Rename	It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

Code Guess what does that code do?



```
#include <sys/types.h>                              /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);                   /* ANSI prototype */

#define BUF_SIZE 4096                               /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                            /* protection bits for output file */

int main(int argc, char *argv[])
{
     int in_fd, out_fd, rd_count, wt_count;
     char buffer[BUF_SIZE];

     if (argc != 3) exit(1);                        /* syntax error if argc is not 3 */

     [redacted]
     in_fd = open(argv[1], O_RDONLY);               [redacted]
     if (in_fd < 0) exit(2);
     out_fd = creat(argv[2], OUTPUT_MODE);
     if (out_fd < 0) exit(3);

     [redacted]
     while (TRUE) {
          rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
          if (rd_count <= 0) break;                 /* if end of file or error, exit loop */
          wt_count = write(out_fd, buffer, rd_count); /* write data */
          if (wt_count <= 0) exit(4);               /* wt_count <= 0 is an error */
     }

     /* Close the files */
     close(in_fd);
     close(out_fd);
     if (rd_count == 0)                             /* no error on last read */
          exit(0);
     else
          exit(5);                                  /* error on last read */
}
```


The program, *copyfile*, can be called, for example, by the command line

```
copyfile abc xyz
```

to copy the file *abc* to *xyz*. If *xyz* already exists, it will be overwritten. Otherwise, it will be created. The program must be called with exactly two arguments, both legal file names. The first is the source; the second is the output file.

Resources

Documents and books:

- Andrew S. Tannenbaum, Modern operating systems - Chapter 4: File Systems
- LSB Workgroup - The Linux Foundation, Filesystem Hierarchy Standard

Videos

- Course 102: Lecture 5: File Handling Internals

Thanks!

Created by: Azad Mohamed