File Systems

# Diagram Description automatically generatedSoftware/Hardware Interface Layers

* A picture containing waterfall chart

  Description automatically generatedAll applications need to read/write files: how do we accomplish this?
* We communicate with the OS via std. runtime libraries that send syscalls to the OS.
* It is the OS that interprets the data on disk and gives the user the illusion of a **file**
* Standard runtime library and application then interprets types.
* Standard runtime library offers functionality of navigate directories.
* “UNIX rose from the ashes of a multi-organisational effort in the early 1960’s to develop a dependable timesharing operating system” – Multics
* Designed for a “workgroup” sharing a single system.
* Prepared the foundation for today’s OSs.
* Modern OS’s support multiple file system formats

# Roles of the Operating System in File System

* Hide hardware specifics (SSD or HDD?)
* Provide a uniform view (all disk resources)
* Allocate disk blocks (no need to know geometry)
* Access data (reads for you, OS and filesystem also controls permissions)
* Share data
* Check permissions
* Maintain metadata (tables of where files are)
* Performance (optimising seeking and rotating times)
* Flexibility (works for different types of hardware)

# File system

* Goal: to **abstract** **secondary storage,**
  + Key abstractions are **files**
  + These are organised into **directories.**
* This system enables **sharing of data** between:

Processes

* + People
  + Machines etc.
* Provides additional
  + Access control
  + Consistency (when multiple people accessing it)
  + Reliability (e.g. battery dies when accessing files: want to be left in consistent state)

# File

* A file is an **abstraction**
* Idea: Abstract away concept of disk to offer files
* Graphical user interface, website

  Description automatically generatedShields the user from storage specifics and details
* **Textbook Definition**: a **named collection of related information** with the properties: content, size, owner, protection, last read/write etc.
* Many different file types are understood by **file system**
  + These are: **directories**, **symbolic link**, **devices**
* If it is not of the above type the file system does not care: the application handles it.
* File types understood by **other parts of OS**, **applications** and **libraries** include:
  + **Programs**: executable, object code, source code
  + **Data**: numeric, alphabetic, alphanumeric, binary
* **Types** of files can be encoded into the file themselves
  + e.g. executable files have **headers** to understand that they are binary files for execution
  + Type can be **encoded** into the final name’s or content:
    - Windows encodes types into the name
      * .com, .exe, .bat, .dll, .jpg, .mov, .mp3 etc.
    - Linux deduces types from content of file.

# Basic Operations

* Graphical user interface, text, application

  Description automatically generatedNote that UNIX commands use a **File Descriptor** (fd), whereas Windows use a **handle** in the arguments for read(), write(), sync(), seek() and close().
* Both parameters reference an index that is looked up in a table.

# File Access methods

* Often we want to select **bytes** from files.
* File systems provide different **access methods**:
  + **Sequential** Access: Read/write bytes one at a time, in order
  + **Direct** Access: Random access given a byte number. “Go there, give me this number of bytes”
  + **Record A**ccess (old): File is an array of fixed or variable sized records
  + **Indexed**: one file contains an index to a record in another file. Used by some databases but old.

# Directory Structures

* Diagram

  Description automatically generated4 kinds of structures:
  + **Single-level** directory: Files must have unique names so all can be referenced.
  + **Two-level** directory: **master file** directory that point to **user file** directories.
    - Sharing files requires introduction of **path abstraction**
  + **Tree structured** directories: can lead to eventual replication of files

# Directories

* Directories provide:
  + Ways for users to **organise their files**
  + Convenient **file name space** for user and file system
* Most file systems support **multi-level directories**
  + This involves naming hierarchies(/, /usr, /usr/local etc.
* Most file systems support the notion of **current directory** i.e. which directory we are working in.
* We can use **absolute** and **relative** addressing:
  + **Absolute names**: fully-qualified starting from root of FS
  + **Relative names**: specified with respect to current directory

# Directory Internals

* Directory is **typically just a file** which contains special metadata for related files
* Organised as a **symbol table**:
  + List of <name of file, reference to file> entries
  + Hash table of <name of file, reference to file>
* **Attributes** of the file include: size, protection, location on disk, creation time, access time etc.
* The directory list is usually **unordered.**
  + Commands presenting the list will sort it for you e.g. ls

# Path Name Translation

## Example

* The user wants to open “/one/two/three”
* User command: fd = open(“/one/two/three”, O\_RDWR);
* Inside file system:

1. Open directory “/”.
2. Search for and obtain location of “one”
3. Open directory “one”
4. Search for and obtain location of “two”
5. Open directory “two”
6. Search for and obtain location of “three”
7. Open file “three”

* Two errors will result in abortion:
  + File/directories do not exist
  + User does **not have permission**

# File Protection

* File system implements the following protection system:
  + Controls **who** (user) can access **what** (files)
  + Controls **how** the file can be accessed by user (read, write, read/write, exec)
* These terms are often generalised:
  + Generalize files to **objects** (the “what”)
  + Generalize users to **principals** (the “who”, user or program)
  + Generalize read/write to **actions** (the “how”, or operations)
* The protection system dictates whether a **given action**  performed by a **given principal** on a **given object** should be allowed.
  + e.g. you may read or write your files, but other principals cannot
  + e.g. you can read /group/teaching/cs3, but not write to it
  + e.g. you can write to /passwords, but cannot read it

## Protection Models

* Two different models:
  + Access Control Lists (ACLs)
    - For each **object**, keep **list of principals** and principals’ **allowed actions**
  + Capabilities:
    - Table

      Description automatically generated with medium confidenceFor each **principal**, keep **list of objects** and principals’ **allowed actions**.
* ACL’s are better for less principals, as permissions for each file must be set for every user.
* Capabilities are better for less objects, as permissions for each user must be set for every object.
* Alternatively, Linux splits users into groups of users.
  + This means we can categorise permissions, meaning that we can reduce the number entries for permissions for users.

# Virtual File System Layer

* Every OS supports a set of file systems as different devices might use different file systems.
* The virtual file system layer abstracts away these differences, so though we so a multi-tree of directories, we are unaware of the differences between these file systems.

# File System Data Structures

* Every file system implements multiple data structures
* It achieves its task by storing these structures on both memory and secondary storage during runtime.
* In **secondary storage**, the file system stores:
  + **Boot Control Block**: contains information needed by the OS to boot on runtime
  + **Volume Control Block**: contains volume details, such as
    - No. of blocks in the volume,
    - Size of the blocks,
    - Free-block count
    - Free-block pointers
    - Free-FCB (File Control Block count and FCB pointers)
  + **Directory Structure** (per file system): used to organise the files.
  + **Per-file FCB**: used to hold details regarding the file.
    - Uses unique ID associating it with a directory entry.
* In **memory**:
  + **Mount Table**: contains information about each mounted volume
  + **Directory-structure cache**: holds directory information of recently accessed directories.
    - Improves speed to access data as reading from disk is expensive
  + **System-wide open-file table**: contains copy of FCB of each **open file**, and other information
  + **Per process open-file table**: containspointers to appropriate entries in the system-wide open-file table for all files the process has opened
  + **Buffers**: hold file-system blocks while they are being read/written to a FS

# Disk Allocation Strategies

* Similarly to memory, we need to allocate parts of the disk to different applications.
* **Contiguous Allocation**:
  + search for number of blocks the application wants to safe its files.
  + Can lead to external fragmentation as blocks must be contiguous, and de-fragmentation is expensive. Furthermore to extend files they must be moved to a larger space
* **Linked Allocation**:
  + Creates linked list by using a pointer at the end of each block that points to another block in memory
  + Can only be used for sequential file access and requires space for the pointers: furthermore can be unreliable as losing or damaging a pointer would have a detrimental effect on the file.
* **File Allocation Table (FAT)**
  + Stores a **directory table** which stores an entry per file. Stores the following attributesL
    - Filename
    - Starting block
    - Metadata associated with it
  + Then stores a **File Allocation Table**:
    - Entry per block and indexed by block number.
    - 2 attributes: whether it is in use and the next block. If in use, the next block will either point to another block or a special “end-of-list” value.
  + To add a block, we simply find a 0 block, point the last block in the file to the 0 block and change the 0 block’s value to 1.
  + Problems: can result in a significant number of disk head seeks unless kept in memory. If then kept in memory, it can use a lot of RAM
* **Indexed Allocation**
  + Each file has its own **index block**: an array of pointers to storage addresses.
  + Solves following problems
    - Not finding space via contiguous allocation.
    - Having linked lists to seek continuously to allocate new memory.
    - Graphical user interface, diagram

      Description automatically generatedHaving a huge big table in memory: only table per file.

## Indexed Allocation: Further Information

* Diagram

  Description automatically generatedEvery file and directory is represented by an **inode**: index node
* Inode is a number that describes:
  + **Metadata** describing file’s own, access rights etc.
  + **Location** of file’s blocks on disk

### Inode File System: Directories

* Text

  Description automatically generatedA directory is just a **flat file** of **fixed-sized entries**
* Each entry consists of an inode number and a file name.
* Note: these are the contents of the directory’s file data, **not** the directory’s inode
* There are **special** inodes:
  + root inode
  + Inodes containing all **bad blocks** e.g. blocks unreadable after excessive usage

### Inode File System: Pathname Resolution

* To look up a pathname, you start at the root directory and work down the chain of inodes.
* Example: “/etc/passwd”

Diagram

Description automatically generated

* Directories map filenames to inode numbers
* **Multiple pointers** can point to the **same inode** can exist in different directories,
* Can work even for the same directory with different filenames.
* This prevents having to save the file multiple times.
* Text, letter

  Description automatically generatedIn UNIX this is called “hard link” and can be done using ln:
* Now “/home/foo” and “/tmp/foo” point to the same directory.

### Inode Filesystems: Data and Metadata Layout

* **Superblock**: specific boundaries of next areas and contains heads of freelists of inodes and file blocks
* **Inode Area**: Contains descriptors (inodes for each file on disk, of all the samfe size
* **File Contents Area**: fixed-size blocks

**A picture containing table

Description automatically generated**

### Inode Filesystems: Locating Inodes on Disk

* Directories give the inode number of a file, but how do we find the inode itself on disk?
* Shape

  Description automatically generated with low confidenceIdea: Top part of filesystem contains **all inodes**
* **Inode number** is just **index** of the inode.
* This makes it easy to compute block address of a given inode:
  + block\_addr(inode\_num) = block\_offset\_of\_first\_inode + (inode\_num \* inode\_size)
* This implies that a filesystem has a fixed number of potential inodes. This number is generally set when the FS is created
* The superblock stores important metadata on filesystem layout, list of free blocks, etc.

### Inode File System: Block List in an Inode

* Inodes may point as references to other blocks in the disk.
* This simply model is not good for representing large models.
* Solution: inodes that can point to other inodes that have the pointers to the data.
* This allows you to use other blocks of memory as additional inode pointers.

### Example: Max File size

* Each inode contains 13 block pointers
* First 10 are direct pointers, then single, then double, then triple indirect pointers
* Assume 4B Block pointers and 512B block size
* Calculate the max file size:
  + 10 direct pointers: Direct Access files:
  + 1 single indirect pointer:
  + 1 double indirect pointer:
  + 1 triple indirect pointer:
  + Max file size:

# Conclusion

* The file system is essentially a massive data structure

## Diagram Description automatically generatedFile System Layout

* One important goal of a file system is to **lay data structure** on disk
  + This keeps in mind the physical characteristics of the disk, trying to minimise seeks
  + Sometimes it will try to organise data based on the workload:
    - locality across files within a directory
    - Sequential access to many files
  + Old layouts were inefficient: constantly seeking. Newer file systems are more efficient and newer storage devices **changed constrains**.