Operating Systems

File System

Interface Layers

Exported Abstractions

File System: Roles of The OS

The OS’s job is to hide the hardware specifics from the User and provide a uniform view of the file-system. This involves taking care of allocating disk blocks, accessing and sharing data, checking permissions, maintaining metadata and ensuring performance and flexibility.

File System

The file system abstracts secondary storage, with the key abstraction being files which are organised into directories.

The file system ensables the sharing of data between processes, people, machines etc… while also providing additional access control, consistency, reliability etc…

File

A file is an abstraction of the disk provided by the OS, they shield the user from the details of the storage. Files are a named collection of related information that has a certain set of properties including content, size, owner, protection, last read/write time etc…

Files can also have types that are understood by the file system (directory, symbolic link, devices) and types that are understood by other parts of the OS, libraries and applications (executable, object code, source code, numeric, alphabetic, alphanumeric, binary).

The type of the file can be encoded in it’s name or content (e.g. .exe, .txt etc…), Linux deducts the type from the content.

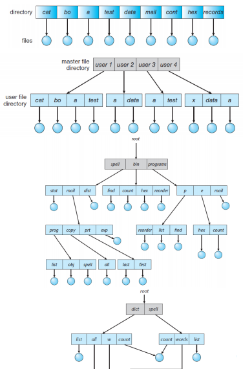
Basic Operations

|  |  |
| --- | --- |
| Unix   * create(name) * open(name, mode) * read(fd, buff, len) * write(fd, buf, len) * sync(fd) * close(fd) * remove(name) * rename(old, new) | Windows   * CreateFile(name, CREATE) * CreateFile(name, OPEN) * ReadFile(handle,…) * WriteFile(handle,…) * FlushFileBuffers(handle,…) * SetFilePointer(handle,…) * CloseHandle(handle,…) * DeleteFile(name) * MoveFile(name) * CopyFile(name) |

File Access Methods

File systems provide different access methods

* Sequential
  + Read/Write bytes one at a time, in order
* Direct
  + Random access given a byte number
* Record
  + File is an array of fixed or variable-sized records
* Indexed
  + One file contains an index to a record in another file

Directories Structures

Single-level directory – File must have unique names

Two-level directory – sharing requires introduction of path abstraction

Tree structured directories – eventual replication of files

Acyclic-graph directories- links as a solution

Directories

Directories provide a way for users to organise their files and a convenient file name space for the user and file system. Most file systems support multi-level directories with naming hierarchies (/, /usr, /usr/local, /usr/local/bin, …) as well as the notion of a current directory (a directory all references will be relative to).

Directory Internals

A directory is typically just a file that happens to contain special metadata and are organised as a symbol table either a list or has table of <name of file, reference to file>. The attributes of directories include size, protection, location on disk, creation time, access time etc.. (much the same as files). The directory lists are usually unordered (listing orders the directory for you).

Path Name Translation Example

If you want to open “/one/two/three” with ‘fd = open(“/one/two/three”, O\_RDWR);’ then inside the file system:

1. Open the root directory ‘/’ (we always know how to find this)
2. Search the directory for ‘one’ and get its location
3. Open directory ‘one’ and search for ‘two’ and get its location
4. Open directory ‘two’ and search for ‘three’ and get its location
5. Open file ‘three’

Note: permissions are checked at each step

The file system spends most of its time walking down directory paths, to aid this the OS will cache prefix lookups so as to not have to do them again.

File Protection

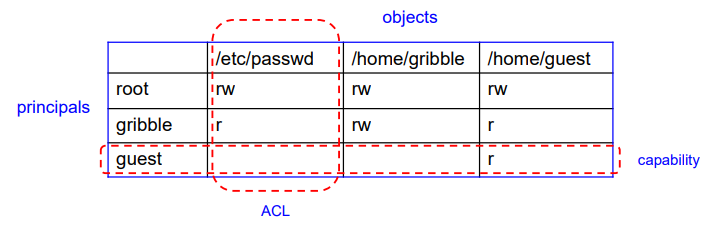
The file system implements a protection system to control who (user) can access what (file) and how that file can be accessed by that user (read, write or exec).

This is often generalised from files to objects, users to principals and read/write to actions.

The protection system dictates whether a given action performed by a given principal on a given object should be allowed.

Protection Models

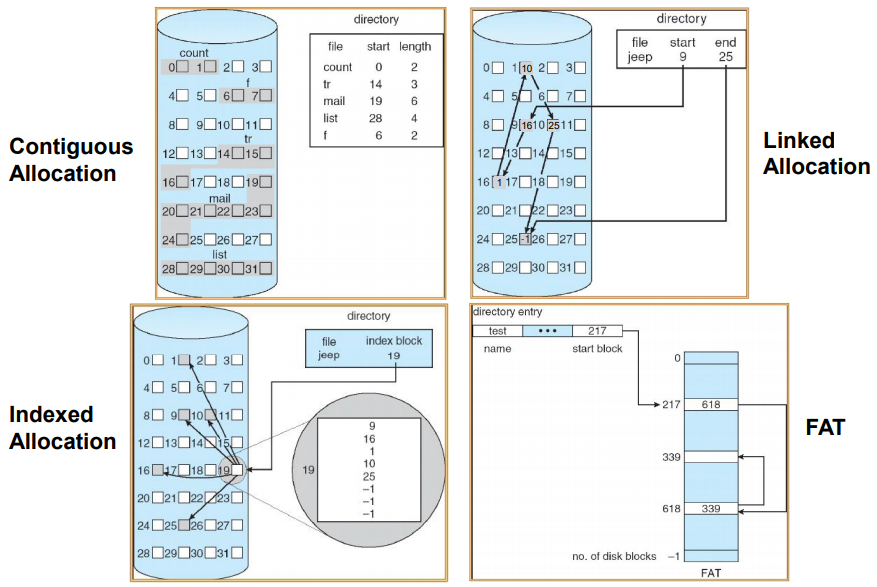
There are two different models for protection:

* Access control lists (ACLs)
  + For each object, keep a list of principals and their allowed actions.
* Capabilities
  + For each principal, keep a list of objects and the allowed actions on them

We can condense the length of the ACL by using three classes of users as the principals, the owner of the file, a group of users, everyone else.

File System Data Structures (high-level)

|  |  |
| --- | --- |
| In Storage we store:   * Boot control block   + Contains information needed by the system to boot an operating system * A volume control block   + Contains volume details such as the number of blocks in the volume, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers. * A directory structure   + Used to organise the files * A per-file FCB   + Contains details about the file   + Has a unique identifier number to allow association with a directory entry | In Memory we store:   * Mount table   + Contains information about each mounted volume * Directory-structure cache   + Holds the directory information of recently accessed directories * System-wide open-file table   + Contains a copy of the FCB of each open file and other information * Per-process open-file table   + Contains pointers to the appropriate entries in the system-wide open-file table for all files the process has opened * Buffers   + Hold file-system blocks when they are being read/written to a FS |

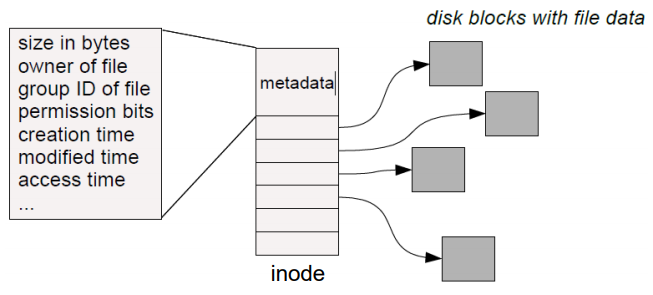
Disk Allocation Strategies

There are many ways to allocate blocks from storage each with befits and downsides.

Contiguous and linked allocation are self explanatory with the first having issues with fragmentation but support sequential and direct access while the later has no fragmentation issues and has trouble supporting direct access.

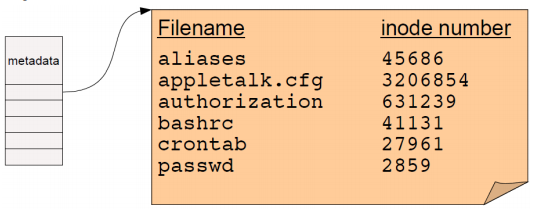
Indexed allocation is an adaptation of linked allocation where we have a block holding all the pointers to the actual data blocks.

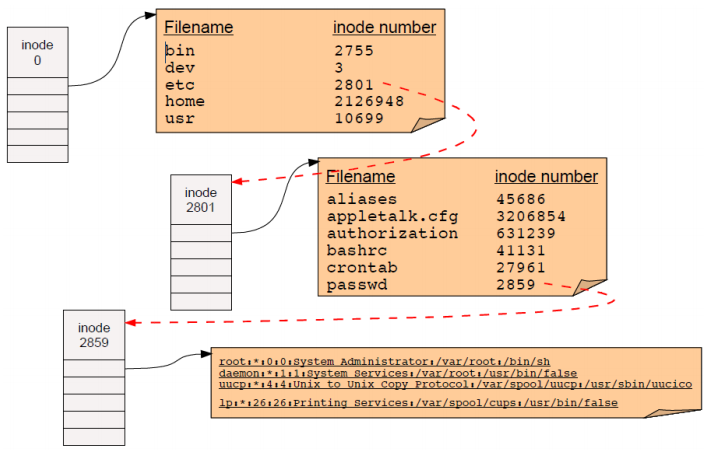
FAT is an adaptation of this again.

Inode File System: Indexed Allocation

Every file and directy is represented by an inode (an index node) which is a number. The inode contains two kinds of information, metadata describing the file’s owner, access rights etc… and the location of the file’s blocks on disk.

Inode File System: Directories

A directory is a flat file of fixed-size entries, each consisting of an inode number and a file name. These are the contents of the directory ‘file data’ itself, NOT the directory’s inode, in UNIX the filenames aren’t stored in the inode.

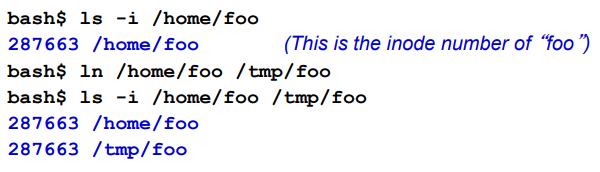
There are also special inodes, the root inode and the Inode containing all bad blocks.

Inode File System: Pathname Resolution

To look up a pathname “/etc/passwd”, we start at the root directory and walk down the chain of inodes:

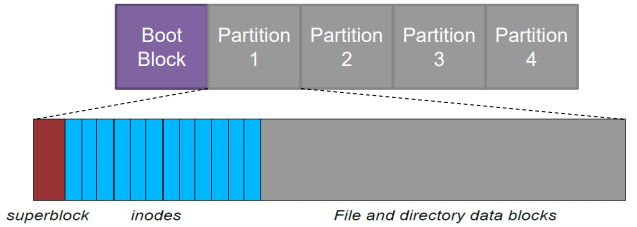
Inode File System: More About Directories

Directories map filenames to inode numbers. There may be multiple pointers to the same inode in different directories (or even the same directory with different filenames), this is as we want to avoid saving the same file multiple times.

In UNIX this is called a ‘hard link’ and can be done using the command (ln) 

“/home/foo” and “/tmp/foo” now refer to the same file on disk.

Note that this is not a copy!

Inode File System: Data and Metadata Layout

The superblock specifies the boundaries of next areas containing the head of the freelist of inodes and file blocks

The inode area contains the descriptors (inodes) for each file on the disk, all of which are the same size.

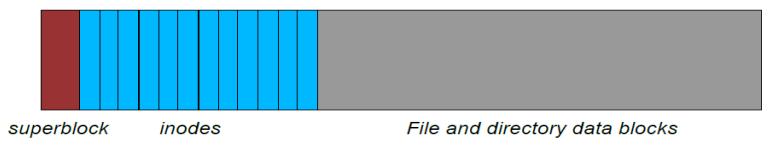
Inode File System: Locating Inodes on Disk

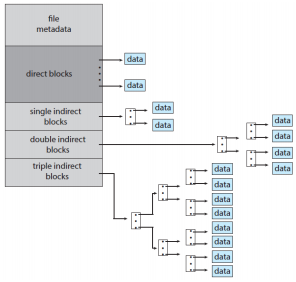
Directories give the inode number of a file, but how do we find the inode itself on disk?

The basic idea is to have the top part of the filesystem contain all of the inodes and have the inode number just be the index of the inode in this area.

It’s easy to compute the 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 (generally set when the filesystem is created).

The superblock stores important metadata on the filesystem layout, the list of free blocks etc…



Inode File System: Block List in and Inode

The block list in an inode points to the block in the file contents area, this is able to represent very small and very large files by having each inode contain 13 block pointers, the first 10 of which point directly to data blocks, then the 11th points to a single indirect pointer, the 12th a double and the 13th a triple.

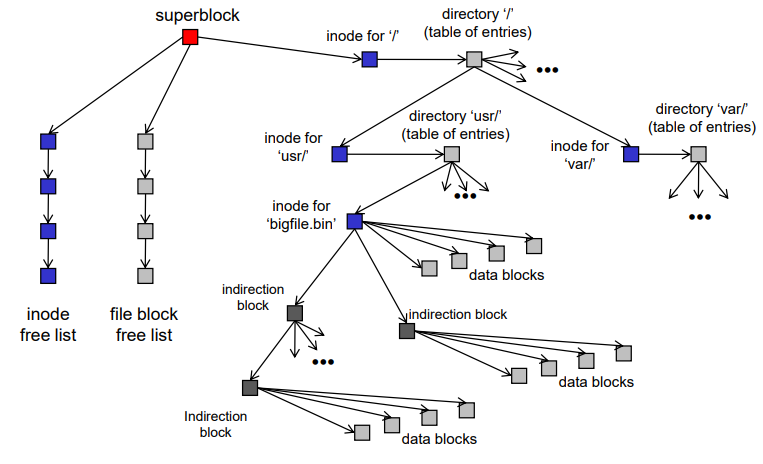
Assuming 4B block pointers and 512B block size, the block list occupies 52B in the inode.

We can get to 10\*512B (5120B) file directly using the 10 direct pointers.

We can get to (512/4)\*512B=64kB with a single indirect reference using the 11th pointer in the inode which will point to a 512B block that contains (512/4) pointers to blocks holding file data.

With the double indirect this goes even further with (512/4)^2 \* 512B = 8MB, and the triple indirect reference gives us (512/4)^3 \* 512B = 1GB.

This means the maximum file size in this scenario is 5120B + 64kB + 8MB + 1GB = ~1GB

All Together

Put together the file system is just a huge data structure