412 Notes: Filesystem

A. Udaya Shankar
shankar@cs.umd.edu

December 5, 2012

Contents

1 Filesystem interface

- Collection of directories and files on non-volatile storage device, typically a disk or a disk partition.
- Many types of filesystems: FAT, UFS, NTFS, ext3, ..., PFAT, GOSFS, GSFS2.
- Directories are usually organized in a tree. There is only one path to an entry.
- Directories can also be organized in an acyclic graph. There can be multiple paths to an entry.
  - How is deletion of shared file/directory handled.
- Directory: zero or more entries, each entry is a directory or a file.
- File: sequence of bytes (historically, records).
- Entry:
  - name: subject to size and char limit.
  - whether directory or file.
  - size: subject to limit;
    directory may have a separate limit on number of entries.
  - metadata:
    date-time of creation / modification / access
    type of data (if file)
    owner
    access restrictions for different users (read, write, execute, set uid, sticky, ...)
- Operations on storage device with filesystem
  - attach (mount) to computer
  - detach (unmount) from computer
  - format device to hold empty filesystem
- Operations on attached filesystem
  - create/delete entry
  - open entry (for reading, writing, ...)
  - change metadata (name, type, ...)
- Operations on open file: read, write, seek, sync, close
- Consistency provided for shared files (in decreasing order of strength):
  - update on write
  - update on sync
  - update on close

2 Filesystem implementation

Implementing the filesystem interface on a non-volatilve storage device.

Device features

- A storage device is usually **block-structured**, i.e., all IO is in terms of blocks. Its IO occurs at a quite different timescale from that of memory (e.g., slower on average, more bursty). Need to move blocks to memory, access them, and put back on device.

  This causes the filesystem to be organized in blocks also. But the two block sizes are usually not the same. Typically, a **filesystem block** is a multiple of a **device block**. Often abbreviate "filesystem block" to "fs block" or just "block".

- A storage device usually has "geometry", e.g., a disk has surfaces, tracks on a surface, and sectors in a track. The access time for a device block depends on its geometric location and on the current position of RW head.

  The geometry should be accounted for when laying out the entries of a filesystem. But for simplicity (at the cost of performance), we treat the device as consisting of an array of blocks. Alleviate performance degradation by appropriate numbering, e.g., [cylinder, surface, sector].

- The blocks of a device usually become corrupted/unusable over time. Need a layout that can handle such failures without losing the filesystem in it.

  This means that the filesystem layout should have redundancy (use duplication and/or error-correction codes).

Design issues

- Boot sector: The first device block of the device is used as a boot sector, to be used if the device is used to boot up the OS. The boot sector is *not* part of the filesystem. It's structure is defined by hardware (e.g., BIOS).

- Superblock: The superblock is the fs block on the device that is examined first when attempting to attach the filesystem on the device to a computer. Usually located just after the boot sector. Has a standard structure so that the OS can determine the filesystem type, whether it can handle it, and how to attach it.

  - magic number indicating filesystem type
  - fs blocksize (vs device blocksize)
  - size of disk (in fs blocks).
  - list of free fs blocks
  - location (fs block number) of root directory

- Directory entry structure (there is one for each entry in the filesystem).
  - entry name
  - entry metadata: size, owner, protection, creation time, etc.
  - pointers to the fs blocks (in device) containing the entry's data.
    (Recall: If the entry is a file, the data is the file's data. If the entry is a directory, the data is a list of directory entry structures. The list may be unordered, sorted, hashed, etc, depending on the number of entries, max size of names, desired performance, etc.)

- Organization of pointers: There are various ways to organize the pointers to an entry's data, i.e., how to go from the directory entry of an entry to the fs blocks containing the data of the entry.

  - **Contiguous**
    - The data is placed in consecutive fs blocks.
    - Directory entry: starting fs block; data size.
    - Pros: random access.
    - Cons: external fragmentation; poor tolerance to bad blocks.
  - **Linked-list**
    - Each fs block of data has a field pointing to the next fs block of data (or indicating that this is the last block).
    - Directory: starting fs block; data size.
    - Pros: no fragmentation; good tolerance to bad blocks.
    - Cons: poor random access.
  - **Separate linked-list**
    - Keep the links in one fs block. Once that block is loaded into memory, the fs block for any part of the data can be quickly obtained. It still requires traversing a linked-list, but the list is now in memory instead of device.
    - Pros: no fragmentation; good tolerance to bad blocks; good random access (as long as file size is not too bigh).
  - **Indexed**
    - Have an array of pointers. Use multiple levels of pointers for accomodating large files.
    - Pros: no fragmentation; good tolerance to bad blocks; good random access (logarithmic cost in data size).

3 FAT (mostly from Wikepedia)

The FAT (File Allocation Table) filesystem is the DOS filesystem. Currently used on simple storage devices: floppies, flash drives, embedded systems, ... There are different versions: FAT12, FAT16 and FAT32; the number indicates the size (in bits) of fs block pointers. Following describes FAT32 (which uses 32-bit fs block pointers).

A FAT32 disk (or partition) is composed of three sections in the following order:

- Reserved sectors: 32 sectors starting from device block 0.

  Device block 0 is the Boot Sector.

  - BIOS parameter block (fs type, pointers to location of the other sections).
  - OS boot loader code.
  - Count of reserved sectors (usually 32 for FAT32 filesystems)

  Device block 1 is the File System Information Sector.

  - signatures
  - last known number of free fs blocks (or 0xFFF..FF if unknown)
  - number of most recently allocated fs block (or 0xFFF..FF if unknown),

  Device blocks 6, 7, 8: Backup Boot Sector.

- FAT region: Size = #FATs×#sectors/FAT.
  Usually holds two copies of the FAT (for redundancy, but rarely used).

  A FAT is a table of entries. Each FAT entry indicates:

  - fs block number of next fs block in chain
  - end-of-chain entry
  - bad-block entry
  - zero (unused fs block)

- Data region: file and directory data. Root Directory Table typically starts at the beginning (fs block 2).

Directory entry 32-byte quantity holding the following:

- name, extension
- attributes (read-only, hidden, system, volume label, subdirectory, archive, device)
- data and time of last modification
- fs block # of first fs block of the entry's data
- size of entry's data

4 UFS (mostly from Wikepedia)

The following is extracted from Wikepedia.

UFS volume consists of following parts:

- few blocks at start reserved for boot blocks.
- superblock: magic number, filesystem geometry/statistics/tuning params
- collection of cylinder groups, each cylinder group has:
  - backup copy of superblock
  - header with statistics, free lists, etc.
  - number of inodes. (Fixed at format time.)
  - number of data blocks

Inodes are numbered sequentially starting at 0. Inodes 0 and 1 are reserved. Inode 2 is the root directory's inode.

Space for a directory is allocated in so-called **chunks**. Chunk size is transferable to disk in one IO op. A directory entry fits in one chunk, but a chunk can hold many directory entries.

Directory entry:

- inode number
- size
- length of file name (up to 255 chars)
- entry name

Inode:

- pointers to file blocks: direct, single-indirect, double-indirect.
- type/access modes
- owner
- number of references
- size, number of blocks

5 Unix-style protection (mostly from Wikepedia)

Every user account has a (unisigned integer) **user id**, abbreviated **uid**. The root user (aka superuser, administrator) has uid equal to 0. Processes and filesystem entries have associated uids, indicating their owners and determining the access that processes have to filesystem entries.

5.1 Process's uids

Every process has two associated uids:

- **effective** user id, abbreviated **euid**, which is the uid of the user account on whose behalf it is currently executing.

  A process's euid determines its access to filesystem entries.

- **real** uid, abbreviated **ruid**, which is the uid of the process's owner.

  A process's ruid determines which processes it can signal: process x can signal process y only if x is a superuser or x's ruid equals y's ruid.

A process's ruid/euid can be changed in the following ways:

- When a process is created, its ruid and euid are set to the creating process's euid.
- When a process with euid 0 executes a `SetUid(z)` system call, where `z` is a non-zero uid. the process's ruid and euid are set to `z`. `SetUid` has no effect when called by a process with non-zero euid; so such a process cannot become a superuser process.

  Example usage: The login process executes with euid 0 (so that it can access the files needed to authenticate a login attempt). After a successful login, the login process starts a shell process and sets its euid/ruid to the authenticated user's uid.

- When a process executes a file `f` whose "setuid bit" is set, the process's euid changes to that of `f`'s owner's uid while it is executing `f`.
- Upon bootup, the first process ("init") runs with uid of 0. It spawns all other processes directly or indirectly.

5.2 Directory entry's uids and permissions

Every directory entry has three classes of users: **owner** (aka "user"); **group** (owner need not be in this group); and **others** (users other than owner or group). Each class's access is defined by three bits: r (read), w (write), x (execute).

- For a file:
  - r: read the file
  - w: modify the file
  - x: execute the file

- For a directory:
  - r: read the names of entries in the directory (but not attributes)
  - w: modify entries in the directory (create, delete, rename)
  - x: access an entry's contents and metainfo (need to know the entry's name).

When a directory entry is created, these attributes are set according to the creating process's attributes (euid, umask, etc).

Each directory entry also has a **setuid** bit.

If an *executable file* has the setuid bit on, the following holds: when a process executes this file (assuming the process's euid has execute permission on the file), the process's euid changes to that of the file's owner while the process is executing the file.

Typically, the executable file's owner is root, allowing a normal user to get root privileges while executing the file (e.g., changing login password). (Note that this is a high-level analog of system calls.)

Each directory entry also has a **setuid** bit.

Setting the sticky bit for an executable file is a hint to the OS to retain the text segment in swap space after the process executed.

Setting the sticky bit for a directory means that only the directory's owner (or superuser) can rename or delete files. (Without the sticky bit set, any user with write and execute permissions on the directory cn rename/delte files.) Usually set on /tmp directory.

5.3 Groups

UFS also has the notion of groups. Briefly, a group is identified by a group id, abbreviated gid. A gid defines a set of uids.

A user account can be in different groups, i.e., have multiple gids.

A process has an **effective** gid (egid) and **real** gid (rgid), which play a similar role as euid and ruid.

A directory entry has a setgid bit, which plays a similar role to setuid for executables and an entirely different role for directories.