# Lecture 15

## File systems in Linux

- What is the Linux Virtual File System?
- What is a superblock?
- What is an inode?
- How is a path name translated to an inode as part of the open() system call ?

# A. The Virtual File System

- Linux supports a wide range of file systems by working with two levels of abstraction.

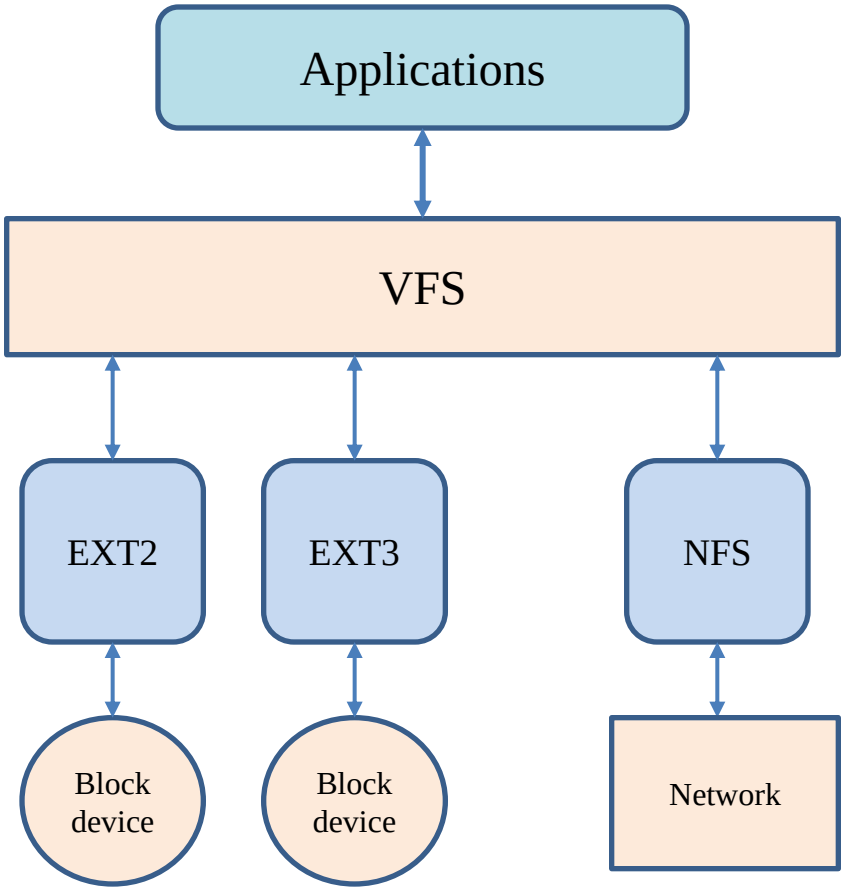
- At the higher level, *the Virtual File System (VFS)* implements a number of common services and generic file operations.
  - It is organised as a collection of base classes, one for each area of file system activity.
  - Manages kernel level file abstractions in one format for all file systems.
  - Receives system call requests from user level (e.g. write, open, stat, link) and resolves them.
  - Interacts with a specific file system based on *mount point traversal*.
  - Receives requests from other parts of the kernel, mostly from *memory management* and *process management*.

- For each VFS service, the specific file system has a structure containing function pointers defining the operations it provides.

- Pointers to these structures are stored in the generic data structures representing mounted file systems, open files, etc.

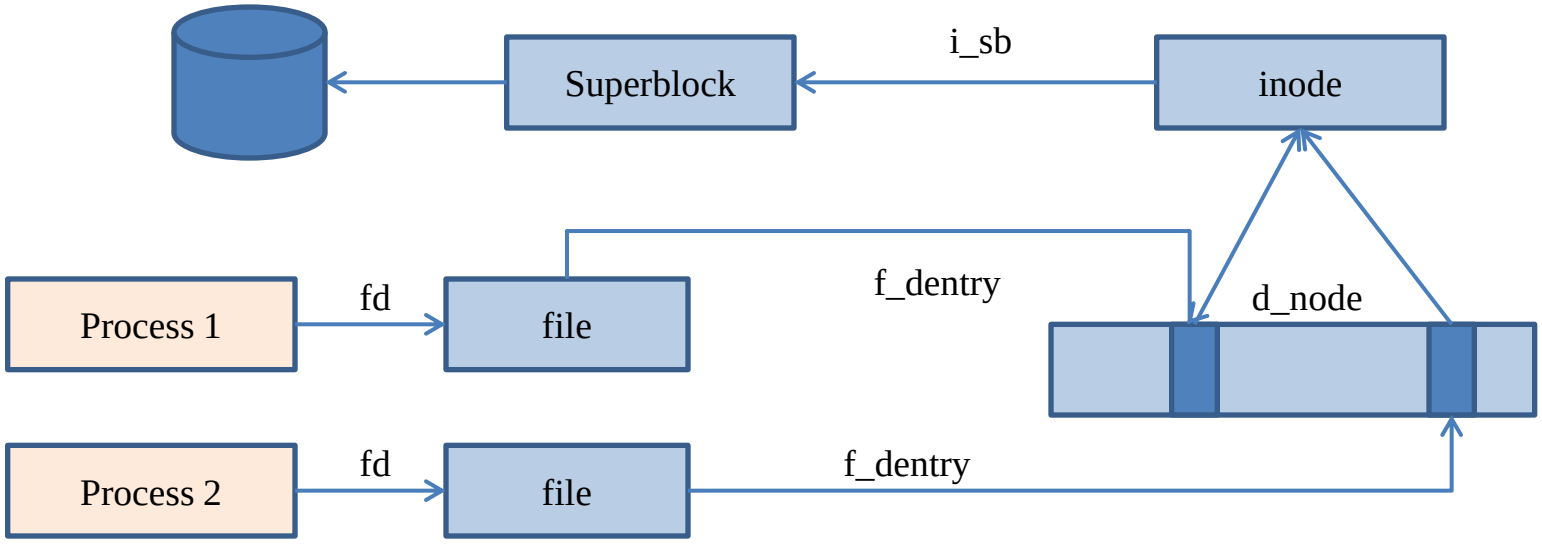
- The starting point for associating file system specific operations with generic operations comes when a file system registers itself (file system is mounted), *by passing a structure with a pointer to the function that loads the superblock*.

# VFS – an abstraction layer between the applications and different file system implementations



# VFS main components

- *superblock:* information about the mounted file system;
- *inode:* information about a specific file;
- *file:* information about an open file;
- *dentry:* information about directory entry.



# Superblocks

- When a FS is mounted, a FS specific function is called to load an internal representation of the FS metadata. Named after the original UNIX on-disk metadata, this is called the *superblock*.

- A member of the internal superblock structure points to a structure of type *struct super_operations*. This structure contains a number of function pointers that are needed to carry out operations on a mounted FS.

- Although the name suggests that these functions are primarily related to the superblock, most are actually functions needed to fetch and manipulate other metadata structures called *inodes*.

# Superblock data structure

- Total number of inodes
- File system size in blocks
- Free block counter
- Free inode counter
- Block size
- Blocks per group
- inodes per group
- 128-bit file system identifier
- Mount counter

# inode structure

- An inode represents an individual file.

- Some of the more interesting members of the structure include:
  - *alloc_inode() – allocate the memory for and initialize an in-memory inode structure;*
  - *read_inode() – read an inode from the fs (disk);*
  - *write_inode() – write a modified inode back to the file system;*
  - *write_super() – similarly handle a modified superblock;*
  - *Sync_fs() – ensure that the fs as stored on the device is up to date with respect to any cached data.*

- One of its members points to a structure which contains function pointers for the operations with inodes or directories described by the inode.

- Some of the relevant operations include:
  - create() – create a new file in a directory;
  - lookup() – fetch a directory entry from a directory;
  - mkdir() – create a new subdirectory;
  - getattr() – return metadata from an i-node.

- Directories are implemented as lists of directory entries. The VFS maintains in the main memory a cache of directory entries that provide a mapping from file names to inodes. They can be quickly searched to avoid unnecessary disk accesses.

- In addition to the inode operations, the internal inode structure also points to a structure of type struct file_operations. When a file is opened, *an internal structure representing the open file is created*. This structure points to the file operations structure (open, read, write, ioctl,..).

# B. The EXT3 file system

- The third extended (EXT3) FS is probably the most used file system in Linux.
- In any disk or partition holding the EXT3, the first block is reserved for boot. The next block is a superblock, which is replicated in several places in the FSs (the block size can be 1024, 2048, 4096 and even 8192 B).
- EXT3 divides the FS into block groups, each of them having a copy of the superblock. The following block is one-block group descriptor table, and then two blocks of free bitmaps – one for free blocks within the group and one for free inodes within the group. Following are inode blocks. The rest of the blocks are for data.
- The strategy is to keep the blocks allocated to a file in the same group together with its inode.
- Each entry of the directory structure contains the file name and the i-number. EXT3 provides an option to speedup directory searches by adding a hash table to the directory.
- EXT3 has a journal, stored as a regular file.

# EXT3 name lookup

- The open() system call enters the kernel with the function sys_open(). The major function is to locate the inode that corresponds to the path name passed by the application.
- The open sys call checks the permission of the operation with the inode and then builds the internal open file data structure.

- After about seven nested function calls, the VFS function __link_path_walk() is invoked, to follow the path name along the directory tree.
- The path name can be either absolute or relative. For an absolute path name, one leading slash is sufficient. The first name points to the directory entry (which in turn points to the inode) – this is the root directory, or the current directory.

# Write operation

- When the file is open, the application can call read() and/or write().
- Writing to a file begins by determining the point where writing starts (part of the open file structure). After that, the control is passed to VFS, where the request is checked to make sure that it doesn't violate security (e.g., file open read-only) or other limitations. Then, the control is passed to the EXT3 specific write function.

  sys_file() →vfs_write() →ext3_file_write()

# Linux Input/Output

- In Linux, all device drivers appear as normal files.

- There are three categories of devices:
  - *Block devices*: allow random access to any fixed-size block of data; examples: hard disks, cd-rom, flash memory.
  - *Character devices*: serial access to characters; examples: keyboard, mouse.
  - *Network devices*: a connection is opened to the kernel's networking subsystem.

# Block devices

- Block devices provide the main interface to all disks.
- A block is the unit of data transferred in a I/O operation.
- The *request manager* manages the reading/writing of the buffer used in a block device transfer. A separate list of requests is stored for each block device driver. This list is maintained in sorted order of increasing starting sector number. A request will be removed from the list after the completion of the operation.
- As new requests arrive, the request manager attempts to merge requests in the list.

# The disk organisation

- The surface of the disk platter is logically divided into circular tracks, that are subdivided into sectors. The set of tracks that are at one arm position makes up a cylinder.

- The disk is addressed as one-dimensional array of logical blocks. These blocks are sequentially mapped onto the sectors of the disk. Therefore, a block number can be translated into a disk address - cylinder number, track number, sector number.

# Disk scheduling

- *The access time*: seek time + rotational latency.
- *The disk bandwidth*: number of bytes transferred divided by the total time between the first request for service and the completion of the last transfer.
- *Disk scheduling* optimises above metrics by good decisions on the next disk request to be served.

- *FCFS*: it is fair but it generally does not provide the fastest service. Example: sequence of requests to blocks on cylinders: 42, 211, 234, 19, 87,.... We can determine the total head movement in terms of cylinders.

# SSTF scheduling

- SSTF: shortest-seek-time-first. Idea: service all the requests close to the current head position before moving far away. SSTF selects the request with the least seek time from the current head position. Example: 42, 19, 87, 211, 234.

- It may cause starvation to some requests, therefore it is not optimal.

# SCAN scheduling

- Also known as the elevator algorithm: the head moves from one end to the other, servicing requests as it reaches each cylinder, then reverses direction, etc.
- Assuming a uniform distribution of requests, when the head approaches one end, relatively few requests are in front of it. The heaviest density of requests is at the other end of the disk.
- C-SCAN (Circular SCAN) is a variant of the above aiming at providing a more uniform wait time. Everything is the same except when the head arrives at an end it will return immediately to the other end, without servicing any requests.
- In practice, neither algorithm is implemented as above. The arm goes only as far as the final request in each direction. Then, it reverses direction without going all the way to the end of the disk – this is called LOOK (SCAN) and C-LOOK (C-SCAN).

# Selection of an I/O scheduling algorithm

- The pattern of I/O requests has a substantial impact on the overall performance. For example, if there is only one request, all algorithms behave the same.
- The file allocation strategy is important: there will be less arm movement if the file is stored contiguously. If the file is indexed, the blocks can be widely spread on the disk.
- The location of directories and index blocks is also important. We can have the directory on the first cylinder, or on the middle cylinder, the latter leading generally to less arm movement.
- Caching directories and index blocks reduce the need for arm movement.
- Other system operations such as demand paging have higher priorities than applications I/O.

# I/O Scheduling in Linux kernel

- V2.6: deadline I/O scheduler works similarly to the elevator algorithm except it associates a deadline with each request.
- By default, read request deadline is .5 sec, and for write is 5 sec.
- The deadline scheduler maintains a *sorted queue* of pending I/O ordered by sector number. It manages two other queues: a *read queue* and a *write queue*, ordered according to the deadline.
- Generally, I/O operations are selected from the sorted queue. However, if a deadline expires, the new request is picked from that queue.
- This policy assures that no request waits longer than its expiration time.

# Character devices

- Any character device driver registered to the kernel must also register a set of functions that implement the file I/O operations accepted by the driver.

- The kernel does almost no pre-processing of a file operation to a character device, it passes the request o the driver.

# Networked-attached storage

- This is a special purpose storage system that is accessed remotely over a network.
- Clients access the storage via RPCs.
- The storage system is usually implemented as a RAID array with an RPC interface.
- RAID (*redundant arrays of independent disks*) are designed to provide performance and reliability.
- The simplest strategy to provide redundancy is to duplicate every disk: a logical disk corresponds to two physical disks – every write is executed on both disks (mirroring).
- One addition to a RAID system is a non-volatile RAM cache which doesn't loose information if power is down.