1 Log structured file system

Consider the **creation** of a new file on a Unix system:

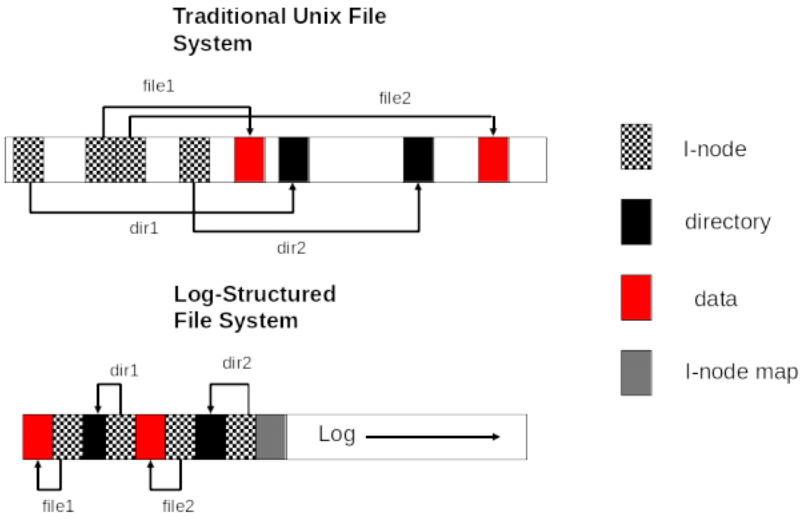
- **Allocate, initialise and write** the i-node for the file i-nodes are usually located at the start of the disk
- **Update** and write the directory entry for the file (directories are tables/files that map names onto i-nodes in Unix)
- Write the data to the disk

The corresponding blocks **are not** necessarily in adjacent locations! Also in linked lists/FAT file systems blocks can be distributed **all over** the disk.

Due to seek and **rotational delays**, hard disks are **slow** compared to other components in a computer (e.g. CPU, main memory).**Can we develop a file system that copes better with the inherent delays of traditional disks?**.
A *log structured file system* aims to improve speed of a file system on a traditional hard disk by **minimising** head movements and rotational delays using **the entire disk** as a great big log. A log is a data structure that is written **only** at the end

1.1 Context

Log structured file systems **buffer read and write operations** (i-nodes, data, etc.) in memory, enabling us to write **larger volumes** in one go. Once the buffer is full it is **flushed** to the disk and written as one contiguous segment at the end of a log. I-nodes and data are all written to the **same segment** .Finding i-nodes (traditionally located at the start of the partition) becomes more difficult An **i-node map** is maintained in memory to quickly find the address of i-nodes on the disk



1.2 Deleting files

A cleaner thread is running in the background and spends its time **scanning** the log circularly and **compacting** it. A hard drive is treated as a **circular buffer**. It **removes** deleted files and files being used right now are marked as **free segments** as they will be later written at the end.

1.3 Advantages and dissadvantages

It greatly **increases** disk performance on writes, file creates, deletes. Writes are more **robust** as they are done as a **single operation**. (Multiple small writes are more likely to expose the file system to serious inconsistency). However, it has not been widely used because it is **highly incompatible** with existing file systems. In addition, the cleaner thread takes **additional** CPU time

2 Journaling file systems

Journaling file systems aim at increasing the **resilience** of file systems against crashes by **recording** each update to the file system as a **transaction**.

2.1 Concept

The key idea behind a *journaling file system* is to **log all events** (transactions) **before** they take place.

- Write the actions that should be undertaken to a log file
- Carry them out
- Remove/commit the entries once completed

If a crash happens in the **middle** of an action (e.g., deleting a file) the entry in the log file **will remain** present after the crash. The log can be examined after the crash and used to **restore** the consistency of the file system. **NTFS** and **EXT3-4** are examples of journaling file systems.

3 Virtual file systems

Multiple file systems usually **coexist** on the same computer. These file systems can be seamlessly integrated by the operating system (e.g. Unix / Linux). This is usually achieved by using **virtual file systems** (VFS). VFS relies on standard object oriented principles (or manual implementations thereof), e.g. **polymorphism**. We devine a **generalised** interface that abstracts different file type implementations.

In a similar way, Unix and Linux **unify** different file systems and present them as a **single hierarchy** and hides away / abstracts the implementation specific details for the user. The VFS presents a **unified interface** to the outside. File system specific code is dealt with in an **implementation layer** that is clearly separated from the interface.

The VFS interface commonly contains the **POSIX** system calls (open, close, read, write, . . .). Each file system that meets the VFS requirements provides an **implementation** for the system calls contained in the interface. Note that implementations can be for **remote file systems** (e.g. sshfs), i.e. the file can be stored on a different machine

3.1 Real word applications

Every file system, including the **root file system**, is **registered** with the VFS.

- A list / table of addresses to the VFS function calls (i.e. function pointers) for the specific file system is provided
- Every VFS function call corresponds to a specific entry in the VFS function table for the given file system
- The VFS maps / translates the POSIX call onto the native file system call

A virtual file system is essentially **good programming practice**

Reference section

placeholder