Operating Systems and Concurrency

File Systems 4
COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture: file system implementations

- **Contiguous implementations** are easy, fast, but result in external fragmentation
- **Linked lists** are sequential, and have block sizes $\neq 2^n$ (page sizes are 2^n)
- **FAT** have block sizes $= 2^n$, but the table becomes prohibitively large
- **I-nodes** are only loaded when the file is open, contain attributes and multiple block levels

Goals for Today

Overview

- File system **paradigms** (which use an underlying file system)
  - Log structured file systems
  - Journaling file systems
- The Unix/Linux **virtual file system**

Log Structured File System

Context

- Consider the creation of a **new file** on a Unix system:
  1. **Allocate**, **initialise** and **write** the **i-node** for the file
     - i-nodes are usually located at the start of the disk
  2. **Update and write the directory** entry for the file
  3. **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 across the disk**

Log Structured File System

Context

- Device characteristics:
  - Due to seek and rotational delays, **hard disks are slow**
  - **SSDs** suffer from **write amplification** (block must be erased), **write disturbance**, and **wear out**
- A **log structured file system** copes better with inherent **device characteristics**:
  - Aims to improve speed of a file system on a traditional hard disk by **minimising head movements** and **rotational delays**
  - Reduces **write amplification**, **disturbance** and **wear out**

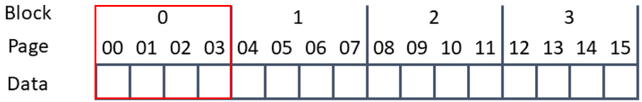

Figure: SSD Write Operation

Log Structured File System

Context

- Device characteristics:
  - Due to seek and rotational delays, **hard disks are slow**
  - **SSDs** suffer from **write amplification** (block must be erased), **write disturbance**, and **wear out**
- A **log structured file system** copes better with inherent **device characteristics**:
  - Aims to improve speed of a file system on a traditional hard disk by **minimising head movements** and **rotational delays**
  - Reduces **write amplification**, **disturbance** and **wear out**

Block	0			
Page	00	01	10	11
Data	00101001	10100101	11101011	00001010

Figure: SSD Write Operation

Log Structured File System

Context

- Device characteristics:
  - Due to seek and rotational delays, **hard disks are slow**
  - **SSDs** suffer from **write amplification** (block must be erased), **write disturbance**, and **wear out**
- A **log structured file system** copes better with inherent **device characteristics**:
  - Aims to improve speed of a file system on a traditional hard disk by **minimising head movements** and **rotational delays**
  - Reduces **write amplification**, **disturbance** and **wear out**

Block	0			
Page	00	01	10	11
Data	11111111	11111111	11111111	11111111

Figure: SSD Write Operation

Log Structured File System

Context

- Device characteristics:
  - Due to seek and rotational delays, **hard disks are slow**
  - **SSDs** suffer from **write amplification** (block must be erased), **write disturbance**, and **wear out**
- A **log structured file system** copes better with inherent **device characteristics**:
  - Aims to improve speed of a file system on a traditional hard disk by **minimising head movements** and **rotational delays**
  - Reduces **write amplification**, **disturbance** and **wear out**

Block	0			
Page	00	01	10	11
Data	10101010	11111111	11111111	11111111

Figure: SSD Write Operation

Log Structured File System

Concept

- A **log** is a data structure that is **written to at the end** and treated as a **circular buffer**
- Log structured file systems **buffer read and write operations** (i-nodes, data, etc.) in memory
    - Enables to **write “larger volumes”**
- 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

Log Structured File System

Structure

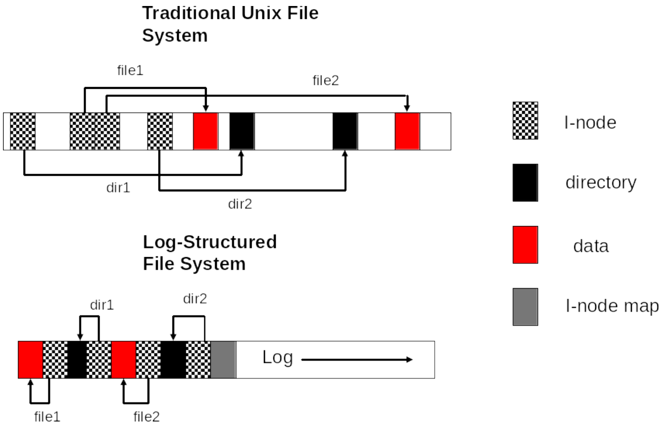

Figure: Blocks written to create two 1-block files: dir1/file1 and dir2/file2

Log Structured File System

Concept: external fragmentation

- A **cleaner thread** (for deleted files) is running in the background and spends its time scanning the log circularly and **compacting** it
- **Deleted** files are marked as free segments and **files being used** right now are written at the end of the log

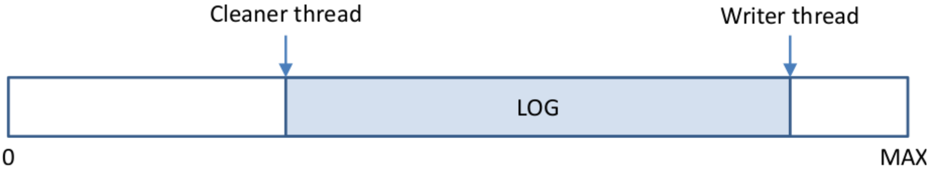

Figure: Log Structured File System

Log Structured File System

Advantages and Disadvantages

- It greatly **increases disk performance** on writes, file creates, deletes but the **cleaner thread** takes additional CPU time
- 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)
- Less frequently used for **regular file systems** (because it is highly incompatible with existing file systems), more frequently used for **SSDs**

File System Implementations

Journaling File Systems: Example

- **Deleting a file** consists of the following actions:
    1. **Remove** the file's **directory** entry
    2. Add the file's **i-node** to the **pool of free i-nodes**
    3. Add the file's **disk blocks** to the **free list**
- It can **go wrong**, for instance:
    - **Directory** entry has been **deleted** and a crash occurs $\Rightarrow$ i-nodes and disk blocks become inaccessible
    - The **directory** entry and **i-node** have been released and a crash occurs $\Rightarrow$ disk blocks become inaccessible

File System Implementations

Journaling File Systems: Example

- **Changing the order** of the events does not necessarily resolve the issues
- Journaling file systems aim at **increasing the resilience** of file systems against **crashes** by recording each update to the file system as a transaction

File System Implementations

Journaling File Systems: 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 the **entry in the log file will remain present**
- The **log is examined** after the **crash** and used to restore the consistency
- **NTFS**, **ext3**, and **ext4** and are examples of journaling file systems

File System Implementations

Virtual File Systems: Concept

- **Multiple file systems** usually co-exist (e.g., NTFS and ISO9660 for a CD-ROM, NFS)
- File systems are **seamlessly integrated** by the operating system's **virtual file systems** (VFS)
- VFS relies on **standard object oriented** principles (or manual implementations thereof), e.g. polymorphism

File System Implementations

Virtual File Systems: Concept

- Consider some code that you are writing, **reading "data records"** (DataObject) from a file
- These records can be stored in **CSV file**, or **XML File**
- How would you make your code resilient against **changes in the underlying data structure**?

File System Implementations

Virtual File Systems: Concept

- Consider some code that you are writing, **reading “data records”** (DataObject) from a file
- These records can be stored in **CSV file**, or **XML File**
- How would you make your code resilient against **changes in the underlying data structure**?
    - You would hide the **implementation** behind **interfaces** using **Data Access Objects** (DAOs)

File System Implementations

Virtual File Systems: Concept

- We can define a **generic interface**, e.g. `DataReader`, containing a method `public DataObject readData();`
  - In the case of file systems, this would be the **POSIX interface** containing reads, writes, close, etc.
- You would hide the CSV and XML code in **specific implementations** of the `DataReader` interface, e.g. `CSVDataReader` and `XMLDataReader`
  - In the case of file systems this would be the file system implementations

File System Implementations

Virtual File Systems: Concept (Cont'ed)

- You would rely on **polymophism** to call the correct method
  - `DataReader dr = new CSVDataReader()`
    `dr.readData() // reads data from CSV`
  - `DataReader dr = new XMLDataReader()`
    `dr.readData() // reads data from XML`

File System Implementations

Virtual File Systems: Concept

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ClientApplication {
  public static void main(String[] args) throws IOException {
    System.out.println("Choose CSV or XML?");
    BufferedReader br
        = new BufferedReader(new InputStreamReader(System.in));
    String type = br.readLine();
    br.close();

    DataReader reader = null;
    if (type.equals("CSV")) {
      reader = new CSVDataReader();
    } else if (type.equals("XML")) {
      reader = new XMLDataReader();
    }

    if (reader != null)
      System.out.println(reader.readData());
  }
}
```


File System Implementations

Virtual File Systems: Concept

- 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**

File System Implementations

Virtual File Systems: Concept

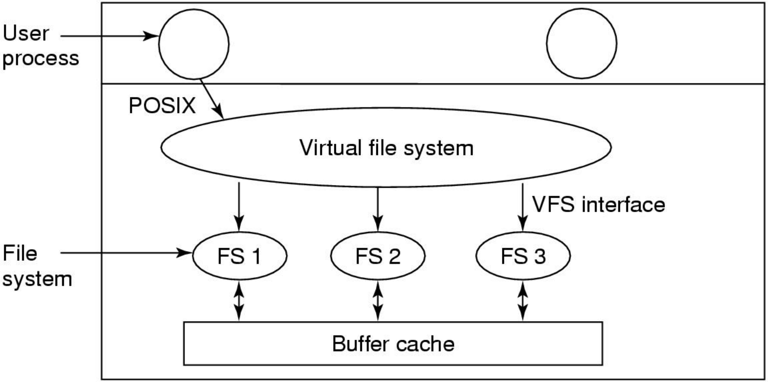

Figure: Virtual File System (Tanenbaum)

File System Implementations

Virtual File Systems: Concept

- 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

File System Implementations

Virtual File Systems: In practice

- 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**

File System Implementations

Virtual File Systems: Example

```
$ df -Th
Filesystem    Type         Size  Used Avail Use% Mounted on
devtmpfs      devtmpfs      32G     0   32G   0% /dev
/dev/sda3     btrfs        215G   82G  132G  39% /
tmpfs         tmpfs         32G  8.6M   32G   1% /tmp
/dev/sda3     btrfs        215G   82G  132G  39% /home
/dev/sda1     ext4         477M  166M  282M  38% /boot
/dev/sdb      ext4         917G  165G  706G  19% /data
pszit@bann:   fuse.sshfs   700M  282M  418M  41% /mnt/bann
```

Summary

Take-Home Message

- File system **paradigms**:
  - Logs: store everything as close as possible
  - Journaling: apply the transaction principle
- VFS: apply good software design (polymorphism)

Test Your Understanding

File Systems

Exercises

Following up the previous question on FAT-32, we mentioned that FAT-32 has severe limitations (e.g. the file allocation table could be too big)

- Why do you think that this file system is still in use in most of flash drives, cameras or MP3 players?
- When would you consider formatting your flash drive with NTFS or ext3?
- If you format your flash drive with (Windows) NTFS, will it work (directly) in Unix systems?