File Systems

OPS Lecture 15, G53OPS/G52OSC

Geert De Maere
(Jason Atkin – OSC)
Geert.DeMaere@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2015

- 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
- **Hard vs. Symbolic/soft links** corresponding to i-node numbers and files containing paths

Goals for Today

Overview

- File system **paradigms**
  - Log structured file systems
  - Journalling file systems
  - The Unix/Linux virtual file system
- Restoring file system **consistency**

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 (directories are tables/files that map names onto i-nodes in unix)
  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 all over the disk**

Log Structured File System

Context

- Due to seeks and rotational delays, **disks are slow** compared to other components in a computer (e.g. CPU)
  - I.e., 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**

Log Structured File System
Concept

- Log structured file systems **buffer read and write operations** (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

Log Structured File System
Concept

- A **cleaner thread** is running in the background that retrieves data from the back of the log, removes the deleted files and places them in the buffer
- Once the buffer is full the **data retrieved at the end** will be **written to the front of the log**
- I.e., a hard drive is treated as a **circular buffer**

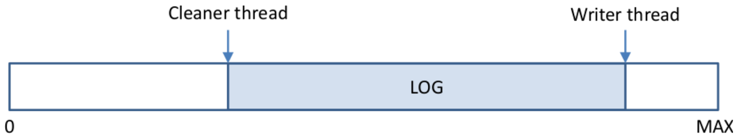

Figure: Log Structured File System

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
- Where can it **go wrong**, e.g.:
  - Directory entry has been deleted and a crash occurs $\Rightarrow$ i-nodes and disk blocks become inaccessible
  - The directory entry and i-nodes 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**

File System Implementations
Journaling File Systems: Concept

- The key idea behind a journaling file system is to **logs all events** before they take place
  - Write the actions that should be undertaken to the log
  - 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**

File System Implementations

Journaling File Systems: Concept (Cont'ed)

- 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

File System Implementations

Virtual File Systems: Concept

- **Multiple file systems** usually exist on the same computer (e.g., NTFS and ISO9660 for a CD-ROM, NFS)
- 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

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 datastructure**?

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 datastructure**?
  - You would hide the **implementation** behind **interfaces** using **Data Access Objects** (DAOs)

File System Implementations

Virtual File Systems: Concept

- A data access object defines 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`
- DAO's combine this with **factories** for object creation

File System Implementations

Virtual File Systems: Concept

- In a similar way, Unix and Linux **unify different file systems** and presents them as a single hierarchy and hides away / abstracts the implementation specific details for the user
- Unix / Linux achieve this by working with a **virtual file system (VFS)** which presents a common 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

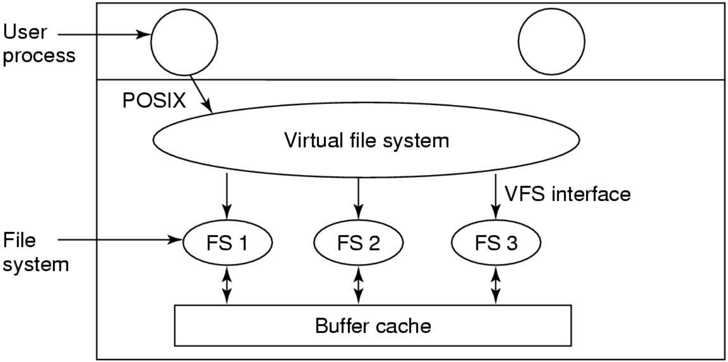

Figure: Virtual File System (Tanenbaum)

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**, 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"

File System Consistency
Restoring Consistency

- Regardless of journaling, etc, file systems can still be left in an **inconsistent state** due to crashes
- This can be problematic, in particular for **structural blocks** such as i-nodes, directories, and free lists
- **System utilities** are available to restore file systems, e.g.:
  - Scandisk
  - FSCK

File System Consistency
Restoring Consistency

- Block consistency is checked by building **two tables**:
  - Table one counts how often a **block is present in a file** (based on the i-nodes)
  - Table two counts how often a **block is present in the free list**
  - A consistent file system has a 1 in either of the tables for each block

File System Consistency

Restoring Consistency

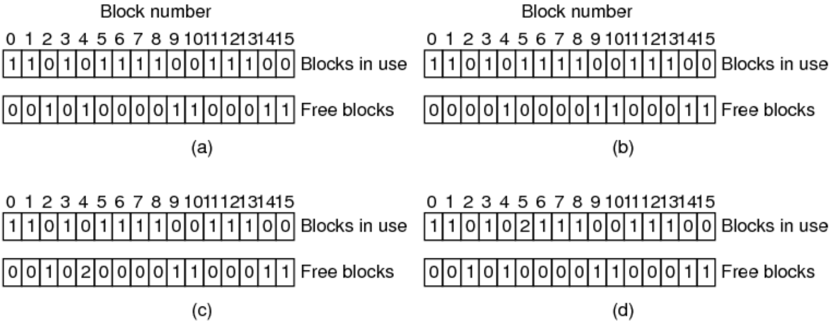

Figure5–18. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

Figure: Consistency checks (from Tanenbaum)

File System Consistency

Types of Inconsistency

- A **missing block**: it does not occur in any of the tables $\Rightarrow$ add it to the free list
- A block is **double counted** in the free list (“disaster” waiting to happen) $\Rightarrow$ re-build the free list
- A block is present in **two or more files**
  - Removing one file results in the adding the block to the free list
  - Removing both files will result in a double entry in the free list
  - Solution: use new free block and copy the content (the file is still likely to be damaged)

File System Consistency
Restoring Consistency

```
FSCK Algorithm:
 1. Iterate through all the i-nodes
    - retrieve the blocks
    - increment the counters
 2. Iterate through the free list
    - increment counters for free blocks
```

File System Consistency
Restoring Consistency

- Checking the directory system: are the **i-node counts correct**
- Where can it go wrong:
  - **I-node counter is higher** than the number of directories containing the file
    - Removing the file will reduce the i-node counter by 1
    - Since the counter will remain larger than 1, the i-node / disk space will not be released for future use
  - **I-node counter is less** than the number of directories containing the file
    - Removing he file will (eventually) set the i-node counter to 0 whilst the file is still referenced
    - The file / i-node will be released, even though the file was still in use

File System Consistency
Restoring Consistency

```
- Recurse through the directory hierarchy
  - Increment file specific counters
  - I.e. each file is associated with one counter
- One file may appear in multiple directories
  - Compare the file counters and i-node counters
  - Correct if necessary
```

Summary
Take-Home Message

- File system **paradigms**:
  - Logs: store everything as close as possible
  - Journalling: apply the transaction principle (similar to databases)
  - VFS: apply good software design (similar to data access objects)
- Restoring **consistency**

File System Implementations
Virtual File Systems: Concept

```
public interface DataReader {

        public DataObject readData();
}
```

File System Implementations

Virtual File Systems: Concept

```java
public class CSVDataReader implements DataReader {
        public DataObject readData() {
                int[] numbers = { 0, 1, 2, 3, 4 };
                String[] text = { "Hello World" };
                // Replace the above with code to read from CVS
                return new DataObject(numbers, text);
        }
}
```

```java
public class XMLDataReader implements DataReader {

        @Override
        public DataObject readData() {
                int[] numbers = { 0, 1, 2, 3, 4 };
                String[] text = { "Hello World" };
                // Replace the above with code to read from XML
                return new DataObject(numbers, text);
        }
}
```

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 CVS or XML?");
    BufferedReader br
        = new BufferedReader(new InputStreamReader(System.in));
    String type = br.readLine();
    br.close();

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

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