File Systems

OPS Lecture 14, G53OPS/G52OSC

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

University Of Nottingham
United Kingdom

2015

Recap
Last Lecture

1. **User view** of file systems
   - System calls
   - Structures, organisation, file types
2. **Implementation view** of file systems
   - Disk and partition layout
   - File tables
   - Free space management
   - ...

Recap
Last Lecture

1. **User view** of file systems
   - System calls
   - Structures, organisation, file types
2. **Implementation view** of file systems
   - Disk and partition layout
   - File tables
   - Free space management
   - ...
3. There is a lot more happening than expected at first sight!

Goals for Today

Overview

- File system **implementations**
  1. Contiguous
  2. Linked lists
  3. File Allocation Table (FAT)
  4. I-nodes (linking files, the Unix V7 file system)
- The **log structured** file system paradigm

Contiguous Allocation
Concept

- **Contiguous file systems** are similar to **dynamic partitioning** in memory allocation:
  - Each file is stored in a single group of **adjacent blocks** on the hard disk
  - E.g. 1k blocks, 100k file, 100 contiguous blocks
- Allocation of free space can be done using **first fit**, **best fit**, **next fit**, etc.

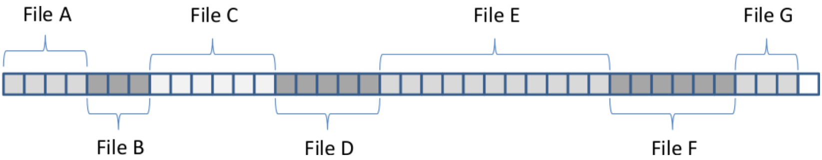

Figure: External Fragmentation when Removing Files

Contiguous Allocation
Advantages

- **Simple to implement**: only location of the first block and the length of the file must be stored
- **Optimal read/write performance**: blocks are co-located/clustered in nearby/adjacent sectors, hence the seek time is minimised (remember the example in lecture on disks!)

Contiguous Allocation
Disadvantages

- Disadvantages of contiguous file systems include:
  - The **exact size** of a file (process) is not always known beforehand: what if the file size exceeds the initially allocated disk space
  - **Allocation algorithms** used to decide which free blocks to allocate to a given file (e.g., first fit, best fit, etc.)
  - Deleting a file results in **external fragmentation**: de-fragmentation must be carried out regularly (and is slower than for memory)
- Contiguous allocation is still in use: **CD-ROMS/DVDs**
  - External fragmentation is less of an issue here since they are write once only

Linked Lists
Concept

- To avoid external fragmentation, files are stored in **separate blocks** (similar to paging) that are **linked to one another**
- Only the **address of the first** block has to be stored to locate a file
- Each block contains a **data pointer** to the next block (which takes up space)

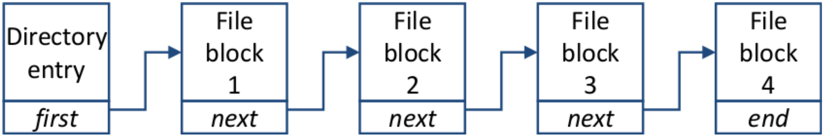

Figure: Linked List File Storage

Linked Lists
Advantages

- **Easy to maintain**: only the first block has to be maintained in the directory entry
- File sizes can **grow dynamically** (i.e. file size does not have to be known beforehand): new blocks/sectors can be added to the end of the file
- Similar to paging for memory, every possible block/sector of disk space can be used: i.e., there is **no external fragmentation**!
- **Sequential access is straightforward**, although **more seek operations/disk access** may be required

Linked Lists
Disadvantages

- **Random access is very slow**, to retrieve a block in the middle, one has to walk through the list from the start
- There is **internal fragmentation**, on average the last half of the block is left unused
  - Internal fragmentation will reduce for **smaller block sizes**
- May result in **random disk access**, which is very slow (remember the example in lecture on disks)
  - **Larger blocks** (containing multiple sectors) will be faster

Linked Lists

Disadvantages (Cont'ed)

- Space is lost within the blocks due to the pointer, the data in a **block is no longer a power of 2**!
- **Diminished reliability**: if one block is corrupt/lost, access to the rest of the file is lost

File Allocation Tables

Concept

- Store the linked-list pointers in a **separate index table**, called a **File Allocation Table (FAT)**

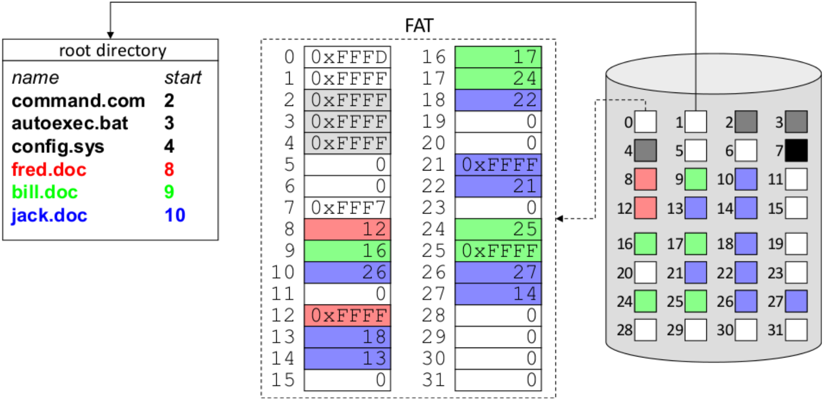

Figure: File Allocation Tables

File Allocation Tables

Advantages

- **Block size remains power of 2**, i.e., no space is lost due to the pointer
- **Index table** can be **kept in memory** allowing fast non-sequential/random access (one still has to walk through the table though)

File Allocation Tables

Disadvantages

- The **size of the file allocation** table grows with the number of blocks, and hence the size of the disk
- For a 200GB disk, with a 1K block size, 200 million entries are required. Assuming that each entry is 3 bytes, this requires **600Mb of main memory**!

I-nodes
Concept

- File attributes and block pointers are stored in separate data structures, called **I-nodes**
  - In contrast to FAT, I-nodes are **only loaded when the file is open** (stored in system wide open file table)
  - If every I-node consists of n bytes, and at most k files can be open at any point in time, at most $n \times k$ bytes of main memory are required
- I-nodes can use **direct block pointers** (usually 12), **indirect block pointers**, or a combination thereof (e.g., similar to **multi-level page tables**)

I-nodes
Concept

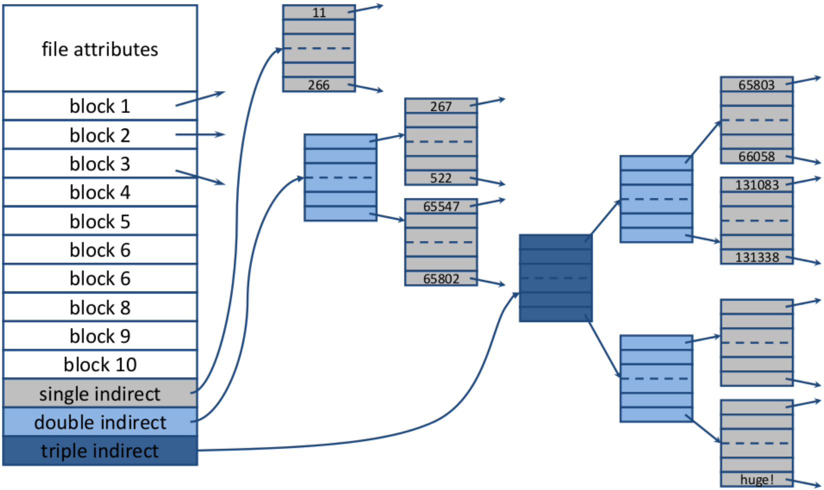

Figure: File Storage Using I-Nodes

File System Comparison

Contiguous vs. Linked vs. Indexed

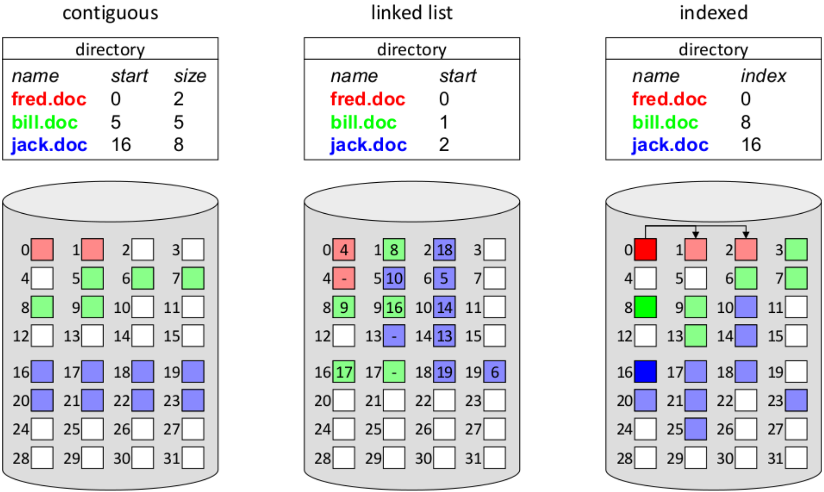

Figure: Contiguous vs. Linked List (or FAT) vs. I-nodes (or indexed)

I-nodes

Lookups

- **Opening a file** requires the disk blocks to be located
  - **Absolute file names** are located relative to the root directory
  - **Relative file names** are located based on the current working directory
- E.g. Try to locate `/usr/gdm/mbox`

/ inode 1

size mode times
2

/ contents 2

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

/usr inode 6

size mode times
132

/usr contents 132

6	.
1	..
19	fred
30	bill
51	jack
26	gdm
45	cfi

/usr/gdm inode 26

size mode times
406

/usr/gdm contents 406

26	.
6	..
64	research
92	teaching
60	mbox
17	grants

Figure: Locating a File

I-nodes
Lookups

Locate the root directory of the file system

- Its i-node sits on a fixed location at the disk (the directory itself can sit anywhere)

Locate the directory entries specified in the path:

- Locate the i-node number for the first component (directory) of the path that is provided
- Use the i-node number to index the i-node table and retrieve the directory file
- Look up the remaining path directories by repeating the two steps above

Once the file's directories have been located, locate the file's i-node and cache it into memory

I-nodes

Hard and Soft Links

- Two approaches exist to **share a file**, e.g. between directory B and C (owner):
  - **Hard links**: maintain two (or multiple) a reference to the same i-node in B and C
    - the i-node link reference counter will be set to 2
  - **Symbolic links**:
    - The owner maintains a reference to the i-node in, e.g., directory A
    - The "referencer" maintains a small file (that has its own i-node) that contains the location and name of the shared file in directory C
- What is the **best approach** $\Rightarrow$ both have advantages and disadvantages

I-nodes
Hard Links

- Disadvantages of hard links:
  - Assume that the owner of the file **deletes** it:
    - If the i-node is also deleted, any hard link will, in the best case, **point to an invalid i-node**
    - If the i-node gets **deleted** and “**recycled**” to point to an other file, the hard links will **point to the wrong file**!
    - The only solution is to **delete the file**, and **leave the i-node intact** if the “**reference count**” is larger than 0 (the original owner of the file still gets “charged” for the space)
- Hard links are the **fastest way of linking files**!

I-nodes
Hard and Soft Links

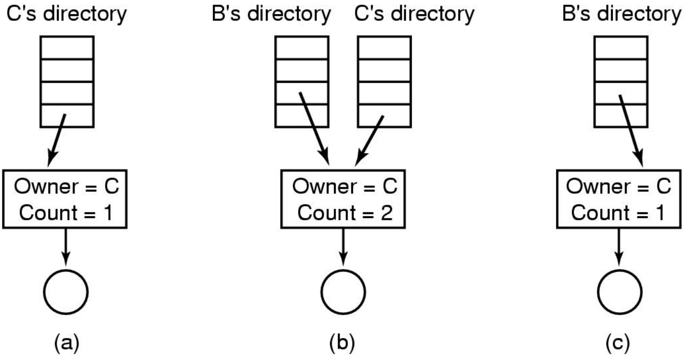

Figure: Hard Links (Tanenbaum)

I-nodes
Symbolic Links

- Disadvantages of symbolic links:
  - They result in an **extra file lookup** (once the link file has been found, the original file needs to be found as well)
  - They require an **extra i-node** for the link file
- Advantages of symbolic links:
  - They can **cross the boundaries of machines**, i.e. the linked file can be located on a different machine
  - There are **no problems with deleting** the original file $\Rightarrow$ the file simply does not exist any more

The Unix V7 File System
Implementation

- **Tree structured** file system with links
- Directories contain **file names** and **i-node numbers**
- I-nodes contain **user attributes** and **system attributes** (e.g. count variable)
- One single, double, and triple **indirect blocks** can be used

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**!

Log Structured File System

Context

- **Disks are slow** compared to other components in a computer (e.g. CPU)
    - Is there any way in which we can make them faster?
    - I.e., can we develop a file system that copes better with the inherent delays of traditional disks
- This is what log structured file systems try to achieve: 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**

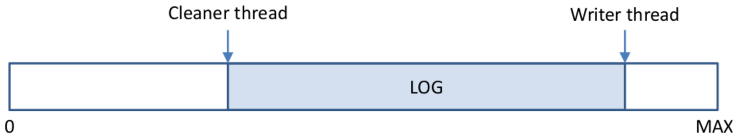

Figure: Log Structured File System

Recap

Take-Home Message

- File system **implementations**
- The **log structured** file system paradigm

Next Lecture

File System Paradigms

- Journaling file systems
- Virtual file system
- Consistency checking/fixing