Lecture 15: File Management

Adam Hawley

March 26, 2019

Contents

1 File Management Outline **2**

2 File Concept **2**

3 File operations **2**

4 File Types **3**

5 Internal File Structure **3**

6 Directories **3**

6.1 Tree-Based Directories 3

6.2 Non Tree-Based Directories 4

7 Access Methods **4**

7.1 Sequential Access 4

7.2 Direct Access 4

8 Access Control **4**

9 Access Granularity **5**

10 File System Mounting **5**

11 Allocation of Files **5**

11.1 Continuous Allocation 5

11.2 Linked Allocation 6

11.3 Indexed Allocation 6

12 Free Block Management 6

13 Efficiency & Performance 7

14 Recovery 7

1 File Management Outline

- Permanent storage of data and programs (files and directories)
- File systems
  - Information and mechanisms used to handle files and directories on disk.
  - File protection
  - Examples: FAT (win9x), NTFS (winNT), ext family, etc.

2 File Concept

A file is a *logical storage unit* where the data is non-volatile. Files tend to have attributes such as name, type, location, size, protection, time of creation and modification etc. File information is kept in the directory.

3 File operations

A file is an abstract data type that can be manipulated by the applications which can be thought of as an array of bytes. The OS provides an API to operate on files, including functions for:

- File creation
- Opening and closing files
- Read and writing files
- Repositioning within a file
- Deleting or resetting
- Appending
- Renaming

- Copying
- Changing privileges
- Memory-mapped files

4 File Types

File types determine which applications can understand the contents of the file. In Unix and Windows, file types correspond to the file name extensions but in old MacOS it was a separate attribute.

5 Internal File Structure

Files are stored on devices such as HDD or SDD. Files are stored in physical blocks HDs without a file structure is just an array of blocks. No more than one file can use an individual block so this results in internal fragmentation. It is important to note that **BLOCK** $\neq$ **SECTOR**.

6 Directories

A directory is a special file which holds information of other files which is owned and managed by the OS. It provides a name space for the file names — all names in name space need to be unique (effectively maps names to the files themeselves). File operations may also update directories.

6.1 Tree-Based Directories

Tree-based directories contain root directories and subdirectories. Files can be accessed using an **absolute path** by prefixing the filename with the sequence of all directories drom the *root* e.g. `\cs\usr\me\lecturing\osi.html`.

They can also be accessed using a **relative path** which is similar to using the absolute path but it makes an assumption that there is a **current directory** e.g. when in `\cs\usr\me` (current) we can just refer to `lecturing\osi.html`. The advantage of this is that there is improved simplicity when dealing with deep trees (normal case).

6.2 Non Tree-Based Directories

Non Tree-Based directories allow files to be reached from multiple directories. To do this we have to use aliasing where multiple names point to the same file. This creates extra challenge where extra care should be taken with the aliases to avoid broken links.

7 Access Methods

Once a file is opened it can be accessed. During the opening of a file, a type of access method is selected: sequential or direct.

7.1 Sequential Access

The file is produced in order, one record after the other. Sequential access is used for applications needing to read files from the start to the end. The API for sequential access typically includes `read_next()`, `write_next()` and `reset()` calls. Writes are usually performed only at the end; data is not always deleted and is instead flagged as invalid.

7.2 Direct Access

File is made up of fixed-length logical records. Records can be read and written in any order (non-sequentially). Therefore the API typically includes `read(n)` and `write(n)` calls. Before a read/write operation is performed, the current file pointer has to be moved to the desired position, **SEEK**.

8 Access Control

Access control is the control over whether and how users/processes can access a file. Types of access: read, write, execute, travers, list, rename and delete. Permissions on each file are typically given at three levels of privileges:

- Owner
- Group (Groups of users, e.g. people in one department)
- Universe (Any other user)

9 Access Granularity

Read-write operations are done in units of **blocks** which are usually one or more sectors of a storage device. Can be from 32 bytes to 64kb but a typical value is 4kb. The OS does the buffering so writes are not done immediately to the disk, only when the block is full, the file is closed or when there is an explicit **flush/synch**.

10 File System Mounting

Before a storage device can be used, it should be **mounted** as part of a file system. It verifies that the information in the disk is a valid file system. Before a storage device can be physically removed it should be **unmounted** which involves checking that all files are closed and all information is written to the disk (no information is cached).

11 Allocation of Files

Files are stored in storage devices as blocks. There are three main mechanisms to allocate blocks to files:

- Continuous Allocation
- Linked Allocation
- Indexed Allocation

11.1 Continuous Allocation

Each file occupies a set of contiguous blocks of disk. This means that the file handler needs to know the first block and the number of blocks.

- Number of disk seeks is minimised since the disk head does not need to move very far.
- Access to file is very easy

However

- Finding space is more difficult (similar problems as external fragmentation in dynamic partitions such as best-fit vs. first-fit policies and relocation of files).

- Allocating file size is also very difficult if files are allowed to grow.

Continuous allocation is used by the IBM VM/CMS operating system for its efficiency.

11.2 Linked Allocation

Each file is a linked list of blocks (each block contains the block number of the next block in the list). This allows blocks scattered throughout the disk. The directory holds a pointer to the first and last block of the disk (this makes reading and accessing the end of the file easier).

- Very easy to accommodate growing files.

However

- A major problem is that access to the i$^{\text{th}}$ block requires scanning all blocks from the first one this can lead to poor worst-case access times $\Longrightarrow$ especially dependent of disk scheduling.

A possible solution to the problem is to group the blocks continuously in clusters.

There is another major problem with linked allocation, its reliability. If one block gets damaged then the rest of the file is lost!

For example, consider FAT (File Allocation Table): Same idea as a linked list, but links are stored at the beginning of each partition for the whole disk rather than the next block number being stored in the current block. What happes if the FAT becomes corrupted?

11.3 Indexed Allocation

Previous methods do not nicely support direct access. In indexed allocation, the first block of a file (the index block) contains a list of the blocks used by the file (similar to the paging memory allocation). This makes direct access easy. The index block is usually cached. Obviously this leads to some wasted space (an additional full block is used for each file). Index blocks can be linked for large files.

12 Free Block Management

When a file needs additional blocks it needs to use the ones which are free. A good allocation policy results in improved performance. Methods could be:

- Bit Vector (i.e. 0001000100100111000, where 1 = free and 0 = busy)
- Linked list containing the free blocks
- Grouping to prepare for files which will never have such a small size.

Allocate a free block which is close to the last block on the file to minimise seek times.

13 Efficiency & Performance

- Efficiency: How Well Available Space is Used
  - Lost space due to pointer sized (12/16/32 bits)
  - Size of index blocks and allocations tables.
  - Internal fragmentation
- Performance
  - Access time
  - Update time: storing *last update time* requires update of directory entry.
  - Adequate usage of cache is essential
  - Read-ahead improves performance in sequential access.

It is difficult to compare because it depends on how the files will be used. If accesses will be mostly sequential then continuous is best. While if accesses will be mostly direct access then indexed is best. Generally it is a combination of approaches that leads to the best performance.

14 Recovery

Data on disk is essential and its loss can cause great harm. In case of a computer failure the data on the disk may be inconsistent e.g. broken links, bad index info. **Consistency checking** is performed when the file system is not properly unmounted. Backup and restore:

- Complete backup or incremental (modified files only, *diff* outputs)
- Using sets of backup devices (tapes, HDs)
- RAID Configs