File Management

Arnav Gupta

April 9, 2024

Contents

1 Overview **2**

1.1 Files and File Systems . . . 2

1.2 File Structure . . . 3

1.3 File Management Systems . . . 4

1.3.1 File System Architecture . . . 5

1.3.2 File Management Functions . . . 6

2 File Organization and Access **6**

2.1 The Pile . . . 7

2.2 The Sequential File . . . 7

2.3 The Indexed Sequential File . . . 8

2.4 The Indexed File . . . 8

2.5 The Direct or Hashed File . . . 9

3 B-Trees **9**

4 File Directories **11**

4.1 Contents . . . 11

4.1.1 Basic Info . . . 11

4.1.2 Address Info . . . 11

4.1.3 Access Control Info . . . 12

4.1.4 Usage Info . . . 12

4.2 Structure . . . 12

4.3 Naming . . . 13

5 File Sharing **13**

5.1 Access Rights . . . 13

5.2 Simultaneous Access 14

6 Record Blocking 14

7 Secondary Storage Management 15

7.1 File Allocation 15

7.1.1 Preallocation vs Dynamic Allocation 16

7.1.2 Portion Size 16

7.1.3 File Allocation Methods 17

7.2 Free Space Management 17

7.2.1 Bit Tables 17

7.2.2 Chained Free Portions 18

7.2.3 Indexing 18

7.2.4 Free Block List 18

7.3 Volumes 19

7.4 Reliability 19

1 Overview

1.1 Files and File Systems

File system allows user to create files with desirable properties:

- **long-term existence**: files stored on disk or secondary storage and do not disappear when a user logs off
- **shareable between processes**: files have names and access permissions for sharing
- **structure**: files have internal structure convenient for particular applications, can be organized hierarchically or have other relations

Functions can be performed on files:

- **create**: new file defined within structure of files
- **delete**: file removed from structure and destroyed
- **open**: existing file declared opened, allowing process to perform functions on the file
- **close**: file is closed for a process, so the process cannot perform further operations on it
- **read**: process reads data in a file

- **write**: process updates a file, either adding new data or changing existing data

1.2 File Structure

A **field** is the basic element of data:

- contains a single value
- has a length and data type
- may be fixed or variable length
- could have subfields of value, name, and length

A **record** is a collection of related fields:

- fixed or variable length depending on if fields are variable length or number of fields varies
- each field has a field name
- has a length field

A **file** is a collection of similar records:

- a single entity referenced by name
- may be created or deleted
- has access control restrictions
- can also be a collection of fields

A **database** is a collection of related data:

- relationships that exist among elements of data are explicit
- consists of 1+ types of files
- DBMS is independent of OS

Operations to use files are:

- `Retrieve_All`: retrieve all file records, required for applications that process all file info at once, similar to *sequential processing*, since records are accessed in sequence
- `Retrieve_One`: requires retrieval of a single record

- `Retrieve_Next`: requires retrieval of the next record in some logical sequence, next to most recently retrieved record
- `Retrieve_Previous`: retrieve previous record to currently accessed record
- `Insert_One`: insert a record into the file, can be at a particular position
- `Delete_One`: delete an existing record, could require updating data structures to preserve sequencing
- `Update_One`: retrieve a record, update 1+ fields, and rewrite updated record to file, preserving sequencing
- `Retrieve_Few`: retrieve a number of records

1.3 File Management Systems

A set of system software that provides services to users and applications in the use of files.

Only way user or application may access files, so system has consistent, well-defined control over files.

Objectives for an FMS:

- meet data management needs and user requirements, including data storage and ability to perform operations
- guarantee that file data are valid
- optimize performance for the system (throughput) and user (response time)
- provide IO support for a variety of storage device types
- minimize or eliminate potential for lost or destroyed data
- provide a standardized set of IO interface routines to user processes
- provide IO support for multiple users

For user requirements (assuming authorized):

1. each user should be able to create, delete, read, write, and modify files
2. each user may have controlled access to other users' files

3. each user may control what types of accesses are allowed to the user's files
4. each user should be able to move data between files
5. each user should be able to backup and recover files in case of damage
6. each user should be able to access files by name rather than numeric ID

1.3.1 File System Architecture

At the lowest level, **device drivers**

- communicate with peripheral devices or their controllers or channels
- are responsible for starting IO operations on a device and processing completion of an IO request

Basic file system is also called physical IO:

- primary interface with the environment outside computer
- deals with blocks of data exchanged with disk or tape
- concerned with placement of blocks on secondary storage and buffering of blocks in main memory
- does not understand content of data or file structure

Basic IO supervisor

- responsible for file IO initiation and termination
- maintains control structures that deal with device IO, scheduling, and file status
- selects device on which file IO is to be performed
- schedules disk and tape access to optimize performance
- IO buffers are assigned and secondary memory allocated

Logical IO

- allows users and applications to access records
- deals with file records

- provides general-purpose record IO capability and maintains basic file data

Access Method

- closest to user, provides a standard interface between applications and file system and devices that hold data
- reflect different file structures and different ways to access and process data
- can optimize access depending on application needs:
  - sequential access (one after another)
  - direct access (direct address)
  - indexed access (use an ID)

1.3.2 File Management Functions

Users and application programs send commands for file operations.

File system must identify and locate selected file. Uses a directory to describe location of all files and their attributes.

Shared systems must enforce user access control.

Basic operations are at the record level. To translate user commands to specific file manipulation commands, appropriate access method for file structure must be employed.

Records or fields of a file must be organized as a sequence of blocks for output and unblocked after input.

Secondary storage must be managed, allocating files to free blocks and managing free storage so available blocks are known for new files and growth in file size.

Optimization (scheduling) depends on file structure and access patterns.

2 File Organization and Access

File organization is logical structuring of records.

Important criteria for choosing file organization are:

- short access time, needed when accessing a single record
- ease of update, some files won't be updated
- economy of storage, should have minimum redundancy, but this can be used to increase data access speed
- simple maintenance
- reliability

2.1 The Pile

Variable-length records, variable set of fields, chronological order.

Data collected in order of arrival. Each record consists of a burst of data.

Purpose is to accumulate the mass of data and save it.

Records may have different or similar fields (in different orders). Each field should be self-describing with name and value, with length implicit with delimiter.

Record access is by exhaustive search, must examine each record.

Useful when data is collected and stored before processing or when data is not easy to organize. Uses space well when data varies in size and structure, is adequate for exhaustive search, and is easy to update.

2.2 The Sequential File

Most common form of file structure. Fixed format used for records, so all records of the same length, having the same number of fixed-length fields in a particular order.

Only field values are stored:

- first field is **key field** with IDs the record, with records stored in key sequence

Useful for applications that process all records, can be easily stored on tape and disk.

Single file access requires sequential search, unless entire file can be brought into main memory at once.

Physical organization of the file matches the logical organization so updates are difficult. A separate pile file is used (log/transaction file) and periodically, a batch update merges log file with master file to produce file in correct key sequence.

Alternative is to organize sequential file as linked list, so each block contains a pointer to the next. Insertion of new records involves pointer manipulation but new records need not occupy a particular physical block.

2.3 The Indexed Sequential File

Records organized in sequence based on key field.

New features are:

- index to support random access, provides lookup capability
- overflow file, similar to log file but integrated so a record in the overflow file is located by following a pointer from its predecessor

Index is a simple sequential file, where each record has a key field and a pointer to the main file. To find a specific field, index is searched to find highest key value that is equal to or precedes the desired key value. Search continues in the main file at the location indicated by the pointer.

Each record in the main file contains a pointer to the overflow file. New records are added to the overflow file with the record in the main file that immediately precedes the new record in logical sequence being updated to contain a pointer to the new record in the overflow file (if it is in the overflow file, that record is updated).

Main file occasionally merged with the overflow file in batch mode.

Greatly reduces single-record access time. Processing entire file sequentially requires processing main file in sequence until a pointer to overflow file is found, then that is followed until a null pointer is encountered.

Multiple levels of indexes for the same key field can increase efficiency. Lowest level treated as sequential file and higher-level index file created for that file.

2.4 The Indexed File

Indexed sequential file inadequate for search based on anything other than key field.

Sequentiality and a single key are abandoned, so records are only accessed through their indexes.

Records can be placed anywhere as long as a pointer in at least 1 index refers to that record. Can also have variable-length records.

Exhaustive index contains 1 entry for every record in the main file, where the index is a sequential file for ease of searching.

Partial index contains entries to records where field of interest exists.

Some records do not contain all fields. When a new record is added to the main file, all index files must be updated.

Mostly used where timelines are important and data is not processed exhaustively.

2.5 The Direct or Hashed File

Directly access any block of a known address, so key field required in each record.

No concept of sequential ordering. Done by hashing on the key value.

Used where rapid access is required, where fixed-length records are used and records are accessed one at a time.

3 B-Trees

For efficient index files, two-level organization is used, where the original file is broken into sections and the upper level consists of a sequenced set of pointers to the lower-level sections. This can be extended to 3+ levels, resulting in a tree. This should be balanced (with good index) to give best performance.

B-tree provides efficient searching, adding, and deleting.

A B-tree is a tree structure (no closed loops) such that:

1. the tree consists of a number of nodes and leaves
2. each node contains 1+ keys that uniquely identify a file record, and more than 1 pointer to child nodes or leaves
    (a) variable number of keys and pointers per node

3. each node is limited to the same number of maximum keys
4. keys in a node are stored in nondecreasing order
    (a) each key has an associated child that is the root of a subtree containing all nodes with keys $\leq$ the key but $\geq$ the preceding key
    (b) a node has a rightmost child that is the root for a subtree containing all keys greater than any keys in the node, so each node has 1 more pointer than keys

A B-tree has minimum degree d and satisfies the following:

1. every node has at most $2d - 1$ keys and $2d$ children (or $2d$ pointers)
2. every node except the root has at least $d - 1$ keys and d pointers
    (a) every internal node is at least half full and has at least d children
3. root has at least 1 key and 2 children
4. all leaves appear on the same level and contain no info, they are simply logical to terminate the tree
5. a non leaf node with k pointers contains $k - 1$ keys

A B-tree has a large branching factor (many children) and so tree has low height. Means searching terminates quickly.

To search for a key, start at the root node, if it's not there, go down 1 level and one case will occur:

1. desired key is $<$ smallest key in this node, so take leftmost pointer to the next level
2. desired key is $>$ largest key in this node, so take rightmost pointer to the next level
3. desired key is between two adjacent keys in this node, so take the pointer between these keys down to the next level

Insertion must maintain a balanced tree:

1. search tree for the key, if it is not in the tree, then a node at the lowest level is reached
2. if this node has $< 2d - 1$ keys, insert the key into this node in the proper sequence

3. if the node is full, split the node around its median key into 2 new nodes with $d-1$ keys each and promote the median key to the next higher level
    (a) insert the new key into the appropriate new node
    (b) original node has now been split into 2 nodes with $d-1$ and d keys
4. promoted node is inserted into the parent node following the rules of step 3 (could be split)
5. if promotion reaches the root and root is already full, a new root is created and height increases

4 File Directories

4.1 Contents

File directory contains info about files, including attributes, location, and ownership. Info is managed by OS, with the directory being a file accessible by file management routines.

Directory maps file names to files with file entries. File location and size is also important, as well as access controls and usage info.

4.1.1 Basic Info

File name: name chosen by creator (user or program), must be unique in a directory

File type: text, binary, load module, etc.

File organization: for systems that support different organizations

4.1.2 Address Info

Volume: indicates device on which file is stored

Starting address: starting physical address on secondary storage

Size used: current size of file in bytes, words, or blocks

Size allocated: maximum size of the file

4.1.3 Access Control Info

Owner: user assigned control of the file, the owner can grant/deny access to other users and to change these privileges

Access info: could have user's name and password for each authorized user

Permitted actions: controls reading, writing, executing, and transmitting over a network

4.1.4 Usage Info

Date created: when file was placed in directory

Identity of creator: usually owner, but not always

Date last read access: date of last time a record was read

ID of last reader: user who did last reading

Date last modified: date of last update, insertion, or deletion

ID of last modifier: user who did the modifying

Date of last backup: date of last time file was backed up on another storage medium

Current usage: info about current activity on file, such as process or processes that have the file open, whether locked by a process, and whether file has been updated in main memory but not disk

4.2 Structure

Some info stored in header record associated with file (less storage needed for directory).

Simplest directory structure is a list of entries, 1 for each file, which can be done with a sequential file with filename as key. Provides no help in organizing files and forces user to be careful not to use the same name for 2 different types of files. Inadequate for multiuser system or single user with multiple files. Difficult to conceal portions of directory from users when there is no directory structure.

Operations on directory:

- **search**: when file referenced, directory must be searched to find the corresponding file entry

- **create file**: when new file created, an entry must be added to the directory
- **delete file**: when file deleted, an entry must be removed from the directory
- **list directory**: all or part of the directory may be requested, request results in a listing of all files owned by a user, plus some attributes
- **update directory**: change in file attributes requires change in corresponding directory entry

Two-level scheme has 1 directory per user and a master directory. Master directory has an entry for each user directory, providing address and access control info. Each user directory is a simple list of user files. Names must only be unique within the files of a single user, and access restriction can be easily enforced. No help in structuring collections of files.

Hierarchical/tree-structure has a master directory that has user directories. This is recursive (subdirectories have subdirectories and files). Can store directories as sequential files or hashed structure.

4.3 Naming

Each file must have a unique symbolic name. This can be achieved in a tree-structured directory by following a path from the master directory down the branches to a file, giving a pathname for a file. Files must only have unique pathname, not unique file name.

User or process has a current **working directory** associated, where files are referenced relative to the working directory.

5 File Sharing

5.1 Access Rights

File system should allow file sharing options to change access controls.

Access rights that a user can be assigned for a file are:

- **none**: user may not know file exists and cannot access it, to enforce, the user would not be allowed to read the directory that includes the file

- **knowledge**: user can determine file exists and its owner to ask owner for access rights
- **execution**: user can load and execute a program but not copy, used for proprietary programs
- **reading**: user can read file for any purpose (including copying and execution), copy could potentially be disallowed
- **appending**: user can add data to a file at the end, but not modify or delete any contents, useful for collecting data
- **updating**: user can modify, delete, and add to file data, which includes writing the file initially, rewriting it, and removing all/part of the data
- **changing protection**: user can change access rights granted to other users, usually only given to owners, owner can also specify which rights can be changed
- **deletion**: user can delete file from system

Rights can be considered a hierarchy. Owner has all access rights and may grant rights to others. Access can be provided to different user classes:

- **specific user**: designated by user ID
- **user groups**: set of users not individually defined, system keeps track of membership
- **all**: all users who have access to this system (public files)

5.2 Simultaneous Access

User may lock an entire file when it must be updated or lock individual records during update.

Issues of mutual exclusion and deadlock must be addressed in designing shared access.

6 Record Blocking

For IO, records must be organized as blocks.

On most systems, blocks have fixed-length (simplifies IO, buffer allocation, and block organization).

The larger the block, the more records that are passed in 1 IO operation. Good for files being processed or searched sequentially, bad for records being accessed randomly (no locality).

IO transfer time is reduced by using larger blocks, but this requires larger IO buffers.

Given the size of a block, 3 methods of blocking are:

1. **fixed blocking**: fixed-length records used and an integral number of records stored in a block, potential for internal fragmentation in each block
    (a) common for sequential files with fixed-length records
2. **variable-length spanned blocking**: variable-length records used and are packed into blocks with no unused space, so some records span two blocks with a continuation pointer
    (a) storage efficient and does not limit record size
    (b) difficult to implement; records that span 2 blocks require 2 IO operations and difficult to update
3. **variable-length unspanned blocking**: variable-length records used, but no spanning, so wasted space in most blocks
    (a) limits record size to the size of a block

Record blocking interacts with virtual memory if employed. A page cannot be treated as a block.

7 Secondary Storage Management

Space on secondary storage must be allocated to blocks. Must keep track of space available for allocation.

7.1 File Allocation

A **portion** is a contiguous set of allocated blocks, whose size and range from a single block to an entire file.

A **file allocation table (FAT)** keeps track of the portions assigned to a file.

7.1.1 Preallocation vs Dynamic Allocation

Preallocation requires the max file size be declared at the time of file creation, which can be difficult to reliably estimate. Overestimating is wasteful of space.

Dynamic allocation allocates space to a file in portions as needed.

7.1.2 Portion Size

In choosing portion size, there is a trade-off between efficiency from the POV of a single file vs overall system efficiency. 4 items to consider are:

1. contiguity of space increases performance (especially for `Retrieve_Next` and for a transaction-oriented OS)
2. having many small portions increases table size to manage allocation info
3. having fixed-size portions (blocks) simplifies the reallocation of space
4. having variable-size or small fixed-size portions minimizes waste of unused storage due to overallocation

Two major alternatives:

1. **variable, large contiguous portions**: better performance since variable size avoids waste and has small allocation tables, but reuse is difficult
   (a) file is preallocated one contiguous group of blocks, so no table needed, only pointer to first block and number of blocks allocated
   (b) must be concerned about fragmentation of free space, could be done using:
      i. **first fit**: choose first unused contiguous group of blocks of sufficient size from a free block list
      ii. **best fit**: choose smallest unused group of sufficient size
      iii. **nearest fit**: choose unused group of sufficient size closest to previous allocation for file to increase locality
2. **blocks**: small, fixed portions provide flexibility, but require large tables or complex structures, less contiguity as blocks are allocated as needed

(a) all required portions allocated at once, so file allocation table is of fixed size since number of blocks allocated is fixed

Both are compatible with preallocation or dynamic allocation.

7.1.3 File Allocation Methods

With **contiguous allocation**, a single continuous set of blocks is allocated to a file at the time of file creation. File allocation table needs a single entry for each file with starting block and length. Best from POV of a sequential file. Multiple blocks can be read in at a time for sequential processing and easy to retrieve single block. Causes external fragmentation, so need to perform compaction. Further, need to declare size of file at creation time.

Chained allocation is done on an individual block basis. Each block contains a pointer to the next block on the chain. File allocation table need a single entry for each file, with starting block and file length. Preallocation possible but more common to allocate blocks as needed, any free block added to the chain. No external fragmentation. Best suited for sequential files to be processed sequentially. Does not accommodate for locality, so even sequential processing accesses different parts of disk.

Indexed allocation has the file allocation table contain a separate 1-level index for each file. The index has 1 entry for each portion allocated to the file. File indexes not physically stored on file allocation table but in separate block, and entry for a file in the file allocation table points to that block. Allocation can be fixed blocks or variable size portions. Blocks eliminates external fragmentation, but portions improves locality. Supports sequential and direct file access.

7.2 Free Space Management

Need a **disk allocation table** to know which blocks on disk are available.

7.2.1 Bit Tables

Use a vector containing 1 bit for each block on the disk, where 0 is free and 1 means in use.

Easy to find contiguous group of free blocks, and small as possible.

Can be sizable, so having disk table in memory or disk is tricky.

When only few free blocks left, exhaustive search of table slow performance, so other data structures are needed.

7.2.2 Chained Free Portions

Free portions may be chained with a pointer and length value in each free portion. Less space overhead and works with all file allocation methods.

With block allocation, choose the free block at the head of the chain and adjust first pointer and length.

With portion allocation, first-fit may be used to find next suitable free portion in chain, with pointer and length values adjusted.

Disk becomes fragmented after some use and when allocating a block, must read block to find pointer to head, slowing file creation.

7.2.3 Indexing

Treats free space as a file and uses an index table. Index should be on the basis of portions rather than blocks, so every table entry is a free portion of the disk.

7.2.4 Free Block List

All blocks assigned numbers and list of numbers of free blocks is maintained in a reserved portion of the disk.

Larger than bit table, so much be stored on disk rather than main memory.

To store part of list in main memory:

1. list can be treated as a push-down stack, with first few thousand elements kept in main memory
    (a) when a new block is allocated, it is popped from the top of the stack (in main memory), and pushed onto stack for dealloc
    (b) only need to transfer between disk and main memory when in-memory portion of stack is full or empty
2. list can be treated as a FIFO queue, with entries from head and tail in main memory
    (a) when new block is allocated, first entry from head is taken, and end of tail is taken for delete from queue

   (b) only need to transfer between disk and main memory when in-memory portion of head or tail of queue is full or empty

7.3 Volumes

A collection of addressable sectors in secondary memory that an OS or application can use for data storage. Sectors in a volume need not be consecutive on a physical storage device, but must appear that way to OS or application. Can come from assembling and merging smaller volumes.

Disk can be partitioned, where each is a separate volume.

7.4 Reliability

To prevent error, the following steps can be performed when a file allocation is requested:

1. lock disk allocation table on disk, so no other user can alter table until this allocation is complete
2. search disk allocation table for available space, assuming a copy is always in main memory (if not, read in)
3. allocate space, update disk allocation table, and update disk, which involved writing disk allocation table back to disk
   (a) for chained disk allocation, involves updating pointers on disk
4. update file allocation table and update disk
5. unlock disk allocation table

When small portions allocated frequently, performance impact is substantial. Batch storage allocation scheme can avoid this, by obtaining a batch of free positions on disk for allocation that are marked "in use". Allocations using this batch may proceed in main memory. When the batch is exhausted, disk allocation table updated on disk and new batch may be acquired. If a system crash occurs, portions on the disk marked "in use" must be cleaned up before they can be reallocated, which depends on file system characteristics.