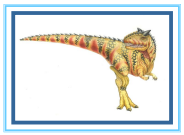
Chapter 11: Implementing File Systems



Chapter 11: Implementing File Systems

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery

Objectives

- To describe the details of implementing file systems and directory structures
- To discuss block allocation and free-block algorithms and trade-offs

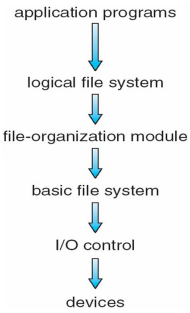
File-System Structure

- Disks provide most of the secondary storage on which file systems are maintained.
- Two characteristics of disks make them convenient for this usage:
  - A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back onto the same place on the disk
  - A disk can access directly any block of information it contains. Thus it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for disk to rotate – details in Chapter 12
- To improve I/O efficiency, I/O transfers between memory and disk are performed in units of **blocks**. Each block has one or more sectors. A sector size varies from 32 bytes to 4,096 bytes (4KB), usually 512 bytes (0.5 KB)

File-System Structure (Cont.)

- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks)
  - This provides efficient and convenient access to disk by allowing data to be stored, located, and retrieved easily
  - It provides user interface: file and file attributes, operations on files, directory for organizing files
  - It provides data structures and algorithms for mapping logical file system to physical secondary storage devices
- File systems are organized into different layers

Layered File System



File System Layers

- **I/O control** and **device drivers** manage I/O devices at the I/O control layer
  - It consists of device drivers and interrupt handlers to transfer information between memory and disks
  - Given commands like "read drive1, cylinder 72, track 2, sector 10" (disk physical address), into memory location 1060" outputs low-level hardware specific commands to hardware controller
- **Basic file system** issues generic commands to the appropriate device driver to read and write physical blocks on the disk
  - gives commands like "retrieve block 123" translates to a specific device driver
  - It also manages memory buffers and caches that hold various file-system, directory, and data block. (allocation, freeing, replacement)
  - Buffers hold data in transit. A block in memory buffer is allocated before the transfer of a disk block occurs.
  - Caches hold frequently used file-system metadata to improve performance
- **File organization module** knows about files, and their logical blocks, as well as physical blocks
  - Translates logical block # (address) to physical block #, pass this to basic file system for transfer
  - Manages free disk space, disk block allocation

File System Layers (Cont.)

- **Logical file system** manages metadata information
  - **Metadata** includes all of the file-system structure except the actual data (or the contents of files)
  - It manages directory structure to provide the information needed by file-organization module.
  - Translates file name into file number or file handle, location by maintaining file control blocks
  - A file control block (**FCB**) (an inode in Unix file systems) contains all information about a file including ownership, permissions, and location of the file contents (on the disk)
  - It is also responsible for protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Many file systems are in use today, and most operating systems support more than one file system
  - Each with its own format - CD-ROM is ISO 9660; Unix has **UFS** (Unix File System) based on FFS; Windows has FAT, FAT32, NTFS (or Window NT File System) as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types of file systems, with **extended file system** ext2 and ext3; plus distributed file systems, etc.
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

File-System Implementation

Several on-disk and in-memory structures are used to implement a file system.

- On-disk structure, it may contain information about how to boot an operating system stored there, the total number of blocks, number and location of free blocks, directory structure, and individual files
- In-memory information used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount time.

On-Disk File-System Structure

- **Boot control block** (per volume) contains info needed by system to boot OS from that volume
  - If the disk does not contain an OS, this block can be empty
  - Usually the first block of a volume. In UFS, it is called the boot block. In NTFS, it is the partition boot sector
- **Volume control block** (per volume) contains volume (or partition) details
  - Total # of blocks, # of free blocks, block size, free block count and pointers, a free FCB count and pointer
  - In UFS, this is called superblock. In NTFS, it is stored in the master file table
- A **directory structure** (per file system) is used to organize the files
  - In UFS, this includes file names and associate inode numbers (FCB in Unix). In NTFS, it is stored in the master file table
- Per-file **File Control Block (FCB)** contains many details about the file
  - It has a unique identifier number to associate with a directory entry.
  - In UFS, inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structure, with a row per file

A Typical File Control Block

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

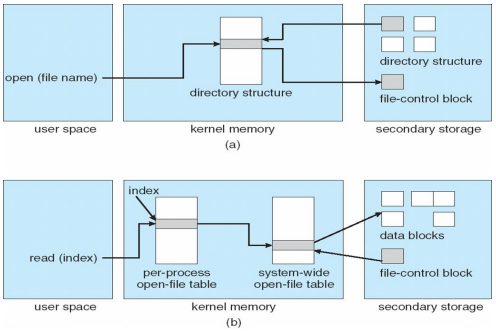
In-Memory File System Structures

- An in-memory **mount table** contains information about each mounted volume
- An in-memory directory-structure cache holds the directory information of recently accessed directories.
- The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information
- The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file tale, as well as other information
- Buffers hold file-system blocks when they are being read from disk or written to disk

In-Memory File System Structures

- Figure 11-3(a) refers to opening a file
  - The open() operation passes a file name to the logical file system.
  - If the file is already in use by another process. A per-process open-file table entry is created pointing to the existing system-wide open-file table.
  - Otherwise, search the directory structure (part of it cached in memory) for the given file name. Once the file is found, the FCB is copied into the system-wide open-file table in memory. This table not only stores the FCB but also tracks the number of processes that have this file open
  - The other fields in the per-process open-file table may include a pointer to the current location in the file (for next read() or write() operation) and access mode in which the file is open.
- Figure 11-3(b) refers to reading a file
  - The open() call returns a pointer to the appropriate entry in the per-process open-file table. All file operations are then performed via this pointer. UNIX systems refer to it as a **file descriptor**; Window refers to it as a **file handle**.
  - Data from read eventually copied to specified user process memory address

In-Memory File System Structures



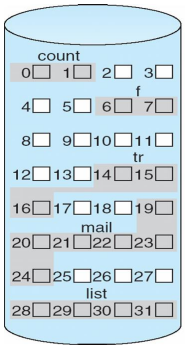
Directory Implementation

- The selection of directory-allocation and directory management algorithms significantly affects the efficiency, performance, and reliability of the file system.
- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
  - The major disadvantage of a linear list is that finding a file requires a linear search time.
    - Cache in memory the frequently used directory information
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - Collisions – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method
  - The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

Allocation Methods - Contiguous

- An allocation method refers to how disk blocks are allocated for files, so the disk space is utilized effectively and files can be accessed quickly.
- There are three major methods of allocating disk space that are widely in use, contiguous, linked and indexed.
- **Contiguous allocation** – each file occupies a set of contiguous blocks of the disk
  - Best performance in most cases – support sequential and direct access easily
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems with finding space for a new file, and when file size grows
  - This is also a dynamic storage-allocation problem discussed earlier, which involves how to satisfy a request of size n (variable) from a list of free holes – external fragmentation exists
  - Best-fit and first-fit are common strategies, and shown to be more efficient than worst-fit.
  - The cost of compaction is particularly high for large disk, which may take hours. Some system require that compaction be one only when off-line, with the file system unmounted
  - File size must be known at the time of file creation – overestimation leads to large amount of internal fragmentation

Contiguous Allocation of Disk Space



directory

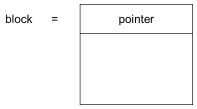
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-Based Systems

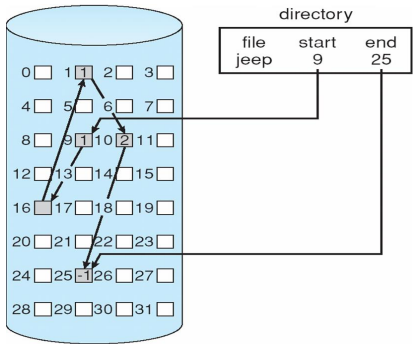
- Some newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents
  - The location of a file's blocks is recorded as a location and a block count, plus a link to the first block of the next extent
  - Internal fragmentation can still be a problem if the extents are too large
  - External fragmentation can become a problem if extents of varying size are allocated and deallocated

Allocation Methods - Linked

- **Linked allocation** – each file consists of a linked-list of blocks
  - Each file is a linked list of disk blocks, which may be scattered anywhere on the disk
  - The directory contains a pointer to the first and last blocks of the file
  - File ends at null pointer (the end-of-list pointer value)
  - Each block contains pointer to next block
  - No compaction needed, and no external fragmentation
  - A file can continue to grow as long as free blocks are available
  - It is inefficient to support direct access of the file, only good for sequential access
  - Extra disk space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78% of the disk space is being used for pointers.
  - Reliability can be a problem; for instance what happen if a pointer is lost or damaged.



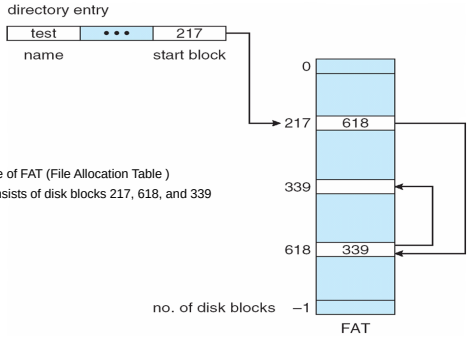
Linked Allocation



Allocation Methods - FAT

- **FAT** (**File Allocation Table**) – an important variation on linked allocation
  - This simple but efficient method of disk space allocation was used by the MS-DOS
  - A section of disk at the beginning of volume is set aside to contain a table called FAT.
  - The table has one entry for each disk block and is indexed by block number
  - The FAT is used in much the same way as a linked list.
  - The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This continues until it reaches the last block, which has a special end-of-file value as the table entry
  - An unused block is indicated by a table entry value 0. Allocating a new block to a file is a simple matter of finding the first 0-value table entry.
  - FAT can be cached. Random (direct) access time is improved, because the disk head can find the location of any block by reading the information in the FAT, instead of moving through blocks stored on the disk in the linked-allocation scheme.

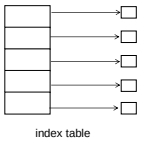
File-Allocation Table



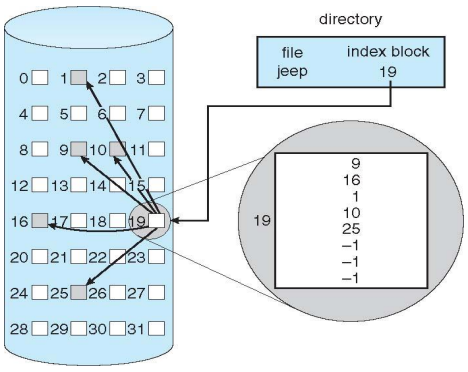
- An example of FAT (File Allocation Table)
- The file consists of disk blocks 217, 618, and 339

Allocation Methods - Indexed

- **Indexed allocation** – brings all the pointers together into one location, the index block
  - Each file has its own **index block**, which contains an array of pointers to its data blocks, or disk-block addresses
- Logical view



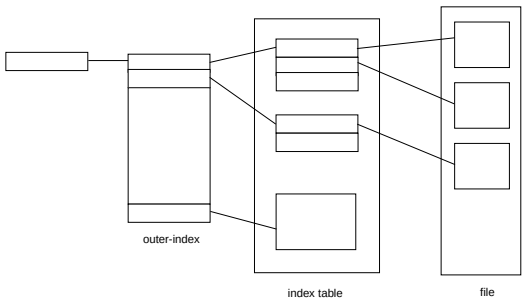
Example of Indexed Allocation



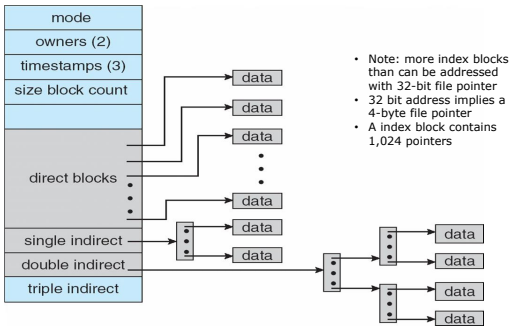
Indexed Allocation

- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space
- Indexed allocation suffers from some of the same performance problem as does linked allocation. Specifically, index block(s) can be cached in memory, but data blocks may be spread all over a volume
- Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead in the linked allocation
  - Suppose a file only has one or two blocks. Indexed allocation lose an entire index block, while linked allocation lose the space of only one pointer per block
- An index block is normally one disk block. What happen if the file is too large such that one index block is too small to hold enough pointers
  - **Linked scheme** – to link several together index blocks
  - **Multilevel scheme** – a first-level index block points to a set of second-level index blocks, which in turn point to the file blocks. This could be continued to a third or fourth level, depending on the desired maximum file size. With a 4,096-byte block, we could store 1,024 **four-byte** pointers in one index block. Two levels of index allow 1,048,576 data blocks, and a file size up to 4GB.
  - **Combined scheme** – direct blocks for small files, and indirect blocks (single indirect, double indirect, and triple indirect blocks) for larger files, used in UNIX-based file systems.

Indexed Allocation – Multilevel Scheme



Combined Scheme: UNIX UFS (4K bytes per block, 32-bit addresses)



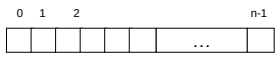
- Note: more index blocks than can be addressed with 32-bit file pointer
- 32 bit address implies a 4-byte file pointer
- A index block contains 1,024 pointers

Performance

- The allocation methods vary in the storage efficiency and data-block access times, both are important criteria
- Best method depends on file access type
  - Contiguous great for sequential and random – maximum file size must be declared when it is created
  - Linked allocation is good for sequential, not random
- Declare file access type at creation -> select either contiguous allocation for direct-access file or linked allocation for sequential-access file
- Indexed more complex
  - If the index block is already in memory, then access can be made directly. However, keeping index block in memory requires considerable space.
  - If the index block is not in memory, we have to read the index block first and then desired data block
  - The performance of indexed allocation depends on the index structure, on the file size, and on the position of the blocked desired

Free-Space Management

- File system maintains **free-space list** to track free disk space
  - (Using term "block" for simplicity)
- **Bit vector** or **bit map** (n blocks)



bit[i] = 1 → block[i] free
0 → block[i] occupied

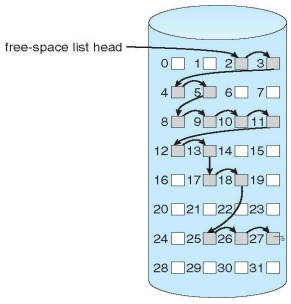
- The main advantage is its simplicity and efficiency in finding the first free blocks or n consecutive free blocks on the disk
- Bit vector is inefficient unless the entire vector can be kept in memory, but requires extra space

block size = 4KB = 2^{12} bytes
disk size = 2^{40} bytes (1 terabyte)
n = $2^{40}/2^{12} = 2^{28}$ bits (or 256 MB)

Linked Free Space List on Disk

Linked-List - link together all the free disk blocks

- Keeping a pointer to the first free block, in a special location on the disk, can be cached in memory
- The first block contains a pointer to the next free blocks, and so on
- Easily locate one free block, not easy to obtain contiguous blocks
- This is not efficient to traverse the entire list, since it must read each block, which requires substantial I/O time. Fortunately, this is not a frequent action



Free-Space Management (Cont.)

- **Grouping**
  - Modify linked-list to store addresses of n free blocks in first free block, The first n-1 of these blocks are actually free. The last block contains addresses of another n free blocks, and so on.
  - The addresses of a large number of free blocks can be found more quickly than linked-list
- **Counting** - Because several contiguous blocks may be allocated and freed simultaneously, particularly when contiguous-allocation algorithm or extents is used
  - By taking advantage of this, rather than keeping a list of n free disk addresses, we can keep address of first free block and count of following contiguous free blocks
  - Each entry in the free-space list consists of an disk address and a count
  - Although each entry requires more space than would a simple disk address, the overall list is shorter, as long as the count is generally greater than one
  - Note this method of tracking free space is similar to the extent method of allocating blocks.

Efficiency

- Efficient use of disk space are heavily dependent on:
  - Disk-allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures

Performance

- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes, as writing contents are usually buffered in the cache, and disk sorts its output queue according to disk address.

Recovery

- To ensure that a system failure does not result in loss of data or data inconsistency
  - A typical operation, such as creating a file, can involve many structural changes within the file system on the disk. Directory structure are modified, FCBs are allocated, data blocks are allocated, and free counts for all these blocks are decreased. These changes can be interrupted by a system crash, and inconsistencies among the structures can result.
- **Consistency checker** – a system program such as fsck in UNIX, compares data in directory structure with data blocks on disk, and tries to fix inconsistencies it finds
  - Scan all the metadata on each file system can confirm or deny the consistency of the system, but can be slow (minutes or hours)
- Use system programs to **back up** data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup