# File Systems

## Agenda

- Basics
  - File system implementation
  - Directory implementation
  - File allocation algorithms
- Introduction to UNIX file system
- Memory-mapped files

## File system implementation

- A disk usually contains multiple partitions
  - These are created when installing the OS
  - Each partition can either be a raw partition or hold a file system
  - Raw partitions (containing unformatted characters) are often used to create swap space in UNIX
  - A file system partition contains four types of information
    - A boot control block or boot block or boot sector: the first block of the partition
    - A superblock or volume control block or master file table: holds various metadata of the file system
    - A directory structure
    - The data blocks containing the file data

## Directory implementation

- A directory is a list of files
  - Implementing a directory is equivalent to implementing a list that can support search, add, and delete operations
  - A linear list is the simplest, but the least efficient
  - Hashing or balanced search trees are expected to perform better

## File allocation

- File allocation is done through a mapping from logical file blocks to physical disk sectors
  - Contiguous or linear allocation: logical file blocks are laid out contiguously on the disk
    - Directory entry contains the block address of the first block and the length of the file (block address vs. disk address?)
    - Offers easy random access, but suffers from external fragmentation due to size over-estimate
    - Growing a file beyond its allocated space is impossible
  - Linked allocation: each file block has a pointer to the next block; no external fragmentation
    - Directory entry contains the addresses of the first and the last blocks
    - File allocation table (FAT) of MS-DOS uses a linked allocation with a table to improve random access latency

## File allocation

- Mapping from file blocks to disk sectors
  - Indexed allocation: each file has its own index block, which stores the list of block addresses corresponding to the file blocks
    - Directory entry stores the index block number
    - Fast random access and no external fragmentation
    - Small files waste a lot of space in the index block
    - Large files may need a linked list of multiple index blocks with the directory entry storing the head index block id
    - For large files, instead of having a linked list of index blocks, it is possible to have multiple levels of index blocks
    - UNIX uses a multi-level index block scheme; each file has an inode (index node) storing ten direct data block addresses, one single-indirect pointer, one double-indirect pointer, and one triple-indirect pointer

## UNIX file system

- Every file has a unique inode
  - Contains information about owner, group, permissions, time of last access, size, data block addresses, and few more pieces of information
  - A directory is treated just like a file where the data blocks hold the file name to inode map for each file under that directory
  - UNIX inodes may reside in disk or in the in-memory inode cache
  - The disk inodes of a partition are arranged in an array of pre-defined size
    - This array is placed between the superblock and the data blocks
    - The size of the array determines the maximum number of files in the file system hosted by the partition

## UNIX file system

- Meaning of directory access permissions
  - Read permission means the data blocks of the directory can be read
    - Data blocks of a directory contain the name to inode maps of all the files under the directory
  - Write permission means the data blocks of the directory can be modified
    - Existing maps can be modified and new maps can be added
    - Still cannot use write() call; there is a set of special functions to write to the data blocks of a directory
  - Execute permission allows one to search the directory and access the directory
    - Can read and write the inodes of the files under the directory

## UNIX file system

- UNIX chdir (or cd) command requires execute permission on the directory
  - Almost no meaningful operation on a file can be done without reading its inode; so changing to a directory is meaningless in this case
- Most useful file operations would require execute permission on the directory
- When an inode is read from the disk, it is copied into a free inode in the in-memory inode cache
  - Inode cache is a hash table
  - The hash key is a function of inode disk partition number and the inode number
  - A free list chains up all free inodes in the inode cache

## UNIX file system

- Every file session starts with a path name
  - The first step is to convert the path name to an inode number
  - Every directory component of the path is traversed checking appropriate permissions
  - Within each directory a linear search is performed to locate the next component of the path (demarcated by /)
    - Once the inode number of the next component is obtained, the inode needs to be read to traverse the next component
    - The process continues until the desired inode number is obtained
    - How to handle file deletion in a directory? What happens to the map? Special meaning of inode number zero

## Memory-mapped files

- Mapping files in memory can speed up file accesses
  - Once a file is mapped to memory, there is no need to invoke file I/O system calls for accessing it; all subsequent accesses will generate standard memory instructions
  - The mmap() call is used to map a file to memory
  - The virtual memory abstraction helps multiple processes share a memory-mapped file and communicate through it
  - Note that physical file on disk may not be coherent with the copy in memory until the file is closed through a munmap() call