File System

Computer Operating Systems
BLG 312E

2017-2018 Spring

Longterm Storage

- need to store very large amounts of data
- stored data should not be lost after process terminates
- processes should be able to share access to the stored data

File System Functions

- file naming
- file access
- file use
- protection and sharing
- implementation

File System Properties

- from the point of view of the user
  - file contents
  - file names
  - file protection and sharing
  - file operations
  - ...

⇒ User interface

- from the point of view of the designer
  - implementation of files
  - free space handling
  - logical block size
  - ....

⇒ File system implementation

File Types

- Files
  - ASCII files
  - binary files
- Directories
  - in most operating systems directory ≈ file

Access within a File

- sequential access
- random access

File Attributes

- information stored in directory structure (resides in secondary storage)
- directory entry: file name and unique id (used to locate file attributes)
  - name: symbolic file name
  - identifier: unique tag used for identification in file system
  - type: for systems that support different types of files
  - location: pointer to device and location of file on device
  - size: current size of file (in bytes, words or blocks) and maximum allowed size
  - protection: access control information (who can read/write/execute, etc)
  - time, date and used identification: for creation, last modification, last use

File Operations

- create / delete
- rename
- open / close / truncate
- read / write / append
- position the file pointer
- query/change file attributes

⇒ through system calls (*open*, *creat*, *read*, *write*, *close*, ...)

Operating System Tables

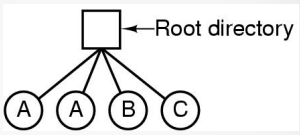
- operating system keeps *open-file* table
  - *system-wide table:* contains process independent info (e.g. location of file on disk, access dates, file size, open count, ...)
  - *per-process table*: keeps track of all files opened by a process (info stored: current file pointer, access rights, accounting info, ...)
- each entry in the per-process table, points to an entry in the system-wide open-file table
- when a process opens a file
  - an entry is added to the system-wide open-file table
  - open count is incremented
  - an entry is added to the per-process open-file table, pointing to the entry in the system-wide open-file table
- upon each file close
  - open count is decremented
  - pointer in the per-process open-file table is removed
  - if open count is zero, the entry is removed from the system-wide open-file table

Directories

- can be viewed as a symbol table that translates file names into their directory entries
- operations:
  - searching for a file
  - create / delete a file
  - list a directory
  - rename a file
  - traverse the file system
- logical structure of a directory: single-level, two-level, tree structure, ...

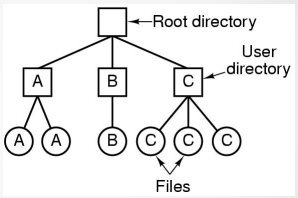
Single-Level Directory Systems

- Provides fast access
- Not suitable for multi-user systems (problem if different users create files with same name)
- May be suitable for embedded systems (e.g. store driver profiles in a car)



Two-Level Directory Systems

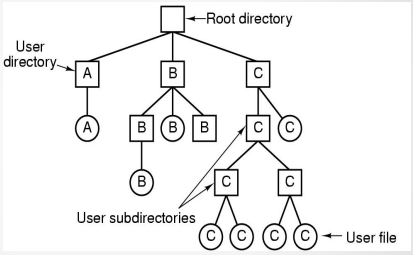
- A directory per user (hence users may have files with same name)
- May be suitable personal computers with multi-users
- System login with a user name and password may be possible



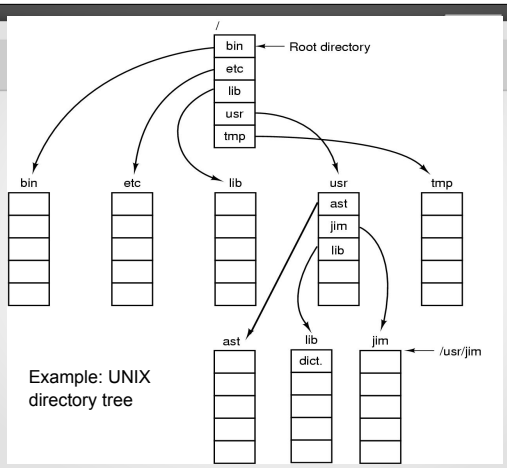
(**Note:** letters show the owners of the files/directories)

Hierarchical Directory Systems (tree structure)

- users wish to keep their files in a logical grouping
- directory tree
- used in modern operating systems



(**Note:** letters show the owners of the files/directories)



Example: UNIX directory tree

File System Implementation – Layered Structure

- file system has a layered structure:
  - application programs *(top level)*
  - logical file system
  - file-organization module
  - basic file system
  - I/O control *(lowest level)*
  - devices
- duplication of code minimized: I/O control and sometimes the basic file system can be used by multiple file systems
- introduces operating system overhead, decreasing performance

I/O Control Level

- consists of device drivers and interrupt handlers
- device driver translates high-level commands into hardware-specific instructions used by hardware controller (interface of I/O device to system)

Basic File System

- issues generic commands to appropriate device driver
- manages memory buffers and caches holding file-system, directory and data blocks
- a block in the buffer is allocated before a disk block transfer can occur

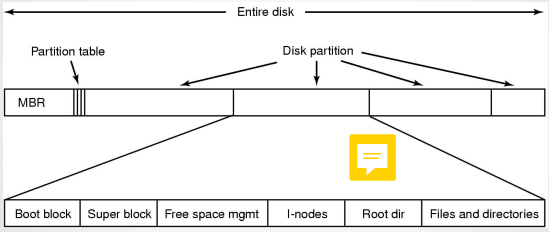
File-Organization Module

- knows about files' logical and physical blocks
- translates logical block addresses to physical block addresses
- also manages free space: keeps track of unallocated blocks

Logical File System

- manages meta-data information
  - meta-data: all of the file system structure except the contents of the files
- manages the directory structure
  - provides the file-organization module with the necessary info when given a symbolic file name
- maintains file structure via file control blocks (FCB)
  - a.k.a. *inode* in UNIX systems
  - FCB contains info on file, such as ownership, permissions, location of file contents, …
- also responsible for protection and security

File System Implementation



Example file system structure (UNIX file system UFS)

File System Implementation

- Boot control block (per volume)
  - info needed by system to boot an operating system from that volume
  - if no operating system on volume, block is empty (raw disk, e.g. swap space in UNIX can use a raw partition)
  - typically is the first block of a volume
  - in UFS: boot block

File System Implementation

- Volume control block (per volume)
  - contains volume (or partition) details (e.g. no of blocks in partition,
  - size of blocks, free block count, free block pointers, free FCB count and free FCB pointers, ...
  - in UFS: superblock

File System Implementation

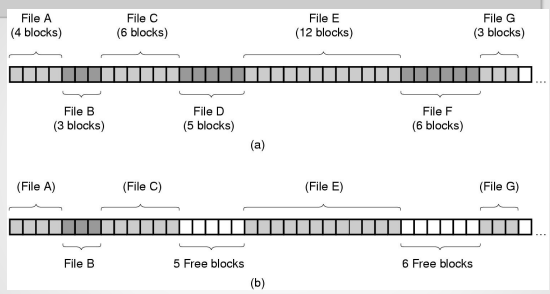
- Directory structure (per file system)
  - for organizing files
  - in UFS: includes file names and associated inode numbers

File System Implementation

- per-file FCB
  - contains details about file
  - has a unique id to associate with a directory entry
  - inodes in UFS

File System Implementation

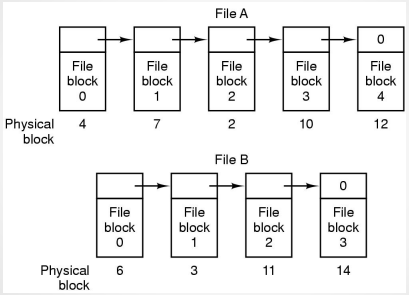
- using contiguous allocation
  - disk addresses define a linear ordering on the disk
  - keep a list of addresses of first blocks and number of blocks for each file
  - advantages
    - easy implementation
    - more efficient “read” operation
  - disadvantages
    - fragmentation on disk (need to compact disk)
    - keep a list of free spaces
      - file size must be known at creation (cannot change)
      - limited maximum file size
  - good for CD-ROM file systems (only one write)



(a) contiguous allocation example: 7 files
(b) view of the disk after files *D* and *E* have been deleted

File System Implementation

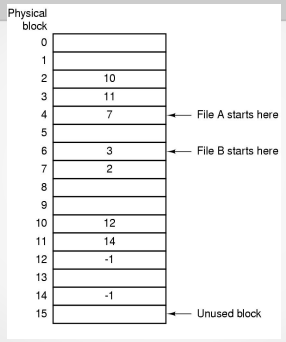
- using linked lists
  - first word of each block is a pointer to the next block
  - no fragmentation (internal fragmentation only in the last block)
  - only the address of the first block of a file is kept
  - access to data in a file: easy sequential access; random access is harder
  - data size in blocks are no longer a power of 2: few bytes taken up by pointer
  - most reads performed in sizes as powers of 2 (need to read two blocks to achieve the required amount of data)



Using linked lists

File System Implementation

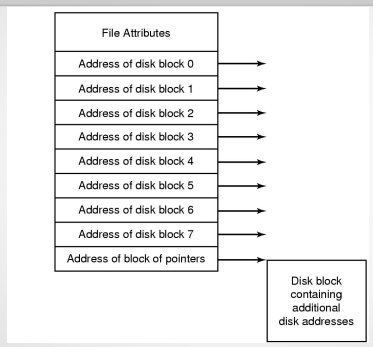
- using file tables in memory
  - keep the pointers in a table in memory (instead of in the blocks on the disk)
  - FAT (File Allocation Table) (used e.g. in MS-DOS)
  - section of disk at the beginning of each volume set for FAT
  - easier random access
    - since table is in memory
  - only need to know the address of the starting block
  - the whole table must be in memory
  - size of table depends on size of disk
    - e.g.: for a 20 GB disk and a block size 1K: need 20 million records of a minimum of 3 bytes in the table (20MB)



using file tables in memory

File System Implementation

- keep an i-node (index-node) for each file
  - contains file attributes
  - contains disk addresses of blocks
- keep only the i-nodes of open files in memory
  - total memory size needed is proportional to the number of maximum files allowed to be open at the same time
- in the simplest implementation, the maximum number of blocks for a file is limited
  - solution: reserve the last entry of the i-node for a pointer to a block containing more block addresses

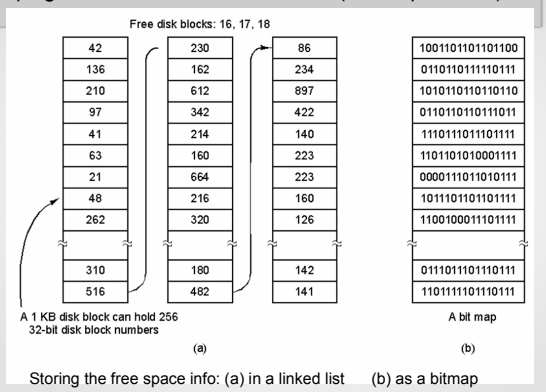


example i-node

Disk Space Management

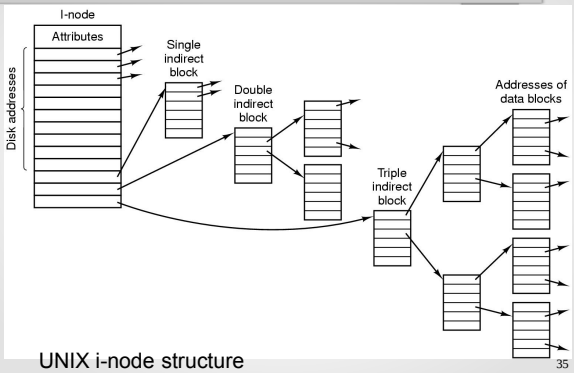
- files split up into blocks of fixed size which do not need to be adjacent on disk
- what should the block size be (unit of allocation) ?
  - same as sector, track, cylinder size?
    - device dependent
  - selection of the size of blocks is crucial
    - performance and efficient disk space usage are contradictory objectives
    - better to choose depending on average file size
    - size is usually predetermined for each system
      - UNIX systems: usually 1K

Keeping Track of Free Blocks on Disk (Free-Space List)



Storing the free space info: (a) in a linked list (b) as a bitmap

UNIX File System (UFS)



UNIX i-node structure

Example:

Consider a UNIX-like file system that uses i-nodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double and triple indirect disk blocks (as shown in the previous slide).

What is the maximum size of a file that can be stored in this file system?