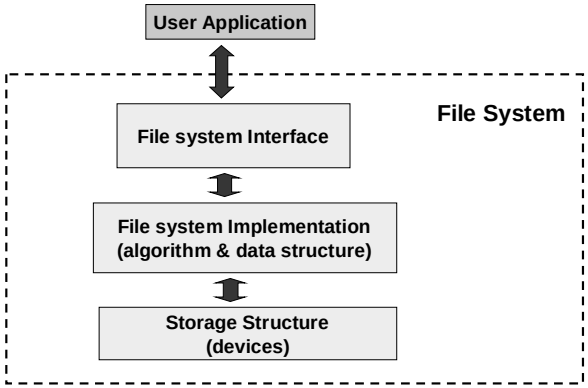
CSE 3221
Operating System Fundamentals

No.11

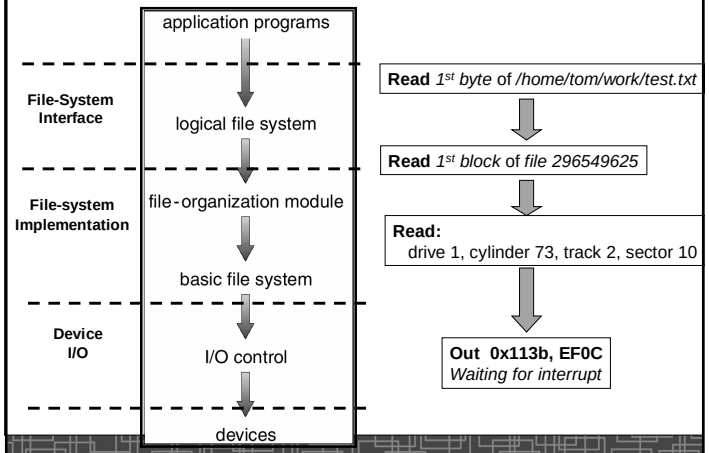
File System

Prof. Hui Jiang
Department of Computer Science and Engineering
York University

File System Overview



File-System Structure



File-System Interface

- ***File*: logical storage unit defined by OS**
  - **A file is a named collection of related information recorded on secondary storage, e.g., disk.**
  - **Files are mapped onto physical devices by the operating system.**
- **File attributes:**
  - **name, identifier, type, location,**
  - **size, protection, time, date and owner.**
- **File operations:**
  - **Creating a file**
  - **Writing a file**
  - **Reading a file**
  - **Reposition within a file**
  - **Deleting a file**
  - **Truncating a file**

File-System Interface (cont')

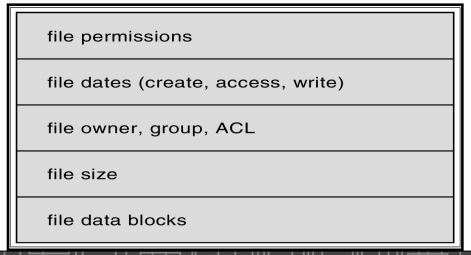
- File types
  - executable, object, source, batch
  - text, archive, library, multimedia
- File structure
- Directory: tree-structured, acyclic-graph, general graph
  - search for a file
  - list a directory
  - Rename a file
  - traverse the file system
- File sharing and protection
  - Consistency semantics
  - Access type: read,write,execute,append,delete,list
  - Access control: owner,group,universe

File-System Implementation

- A logical view of disk space is a linear array of logical blocks.
- OS maintains lots of data structure to implement a file-system
  - On-disk structures:
    - How to boot OS saved in the disk
    - Total number of blocks
    - Number and locations of all free blocks
    - Directory structure
    - Individual file information: FCB(file control block)
  - In-memory structures
    - On-line file-system management
    - Caching

On-disk Structure

- **Boot Control Block** (*boot block*)
  - Contains information to boot an operating system
- **Partition Control Block** *(superblock)*
  - Detailed information of this partition
- Directory structure: to organize files.
- A **FCB (File Control Block)** for each file (*inode*):



In-memory Structure

- An **in-memory partition table** for all mounted partition
- An **in-memory directory structure** for directory information of recently accessed directories (caching)
- The **system-wide open-file table** contains a copy of the FCB's of all open files as well as other information.
- The **per-process open-file table** contains all open files for each process (pointers to the appropriate entry in the system-wide open-file table)

File-System Implementation

- Directory implementation
- Allocation methods
- Free-space management

Directory Implementation

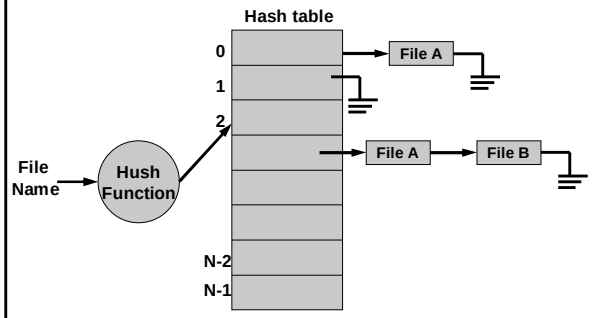
- A directory contains many files and possible sub-directories
- A directory structure need to store all index information about all files in this directory.
- Directory operations:
  - Search/Locate a file in a directory
  - Create a new file in a directory
  - Delete a file from the directory
- Directory implementation:
  - Linear list
  - Hash table

Linear List

- Use a linear list of file names with pointers to data blocks

- Data structures: linear array OR linked list
  - Search/locate a file: linear search of the whole list
  - Create a new file: linear search + adding a new entry
  - Remove a file: linear search + deleting the entry
- Disadvantage: low efficiency due to linear search
- Ordered list: complicating file creation/deletion

Hash Table



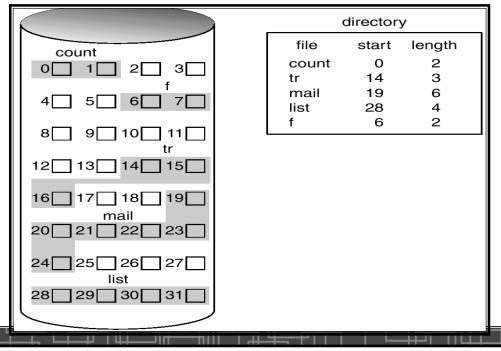
An example of hash function:
(add all characters in file-name) % N

Allocation Methods

- Concerns:
  - Use disk space effectively
  - Access each file efficiently
- A logical view of disk space is a linear array of logic blocks
- Allocation methods:
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation

Contiguous Allocation(1)

- Contiguous-allocation method requires each file to occupy a set of contiguous blocks on the disk.
- The directory entry for each file indicates the disk address of the starting block and the length (in block units).



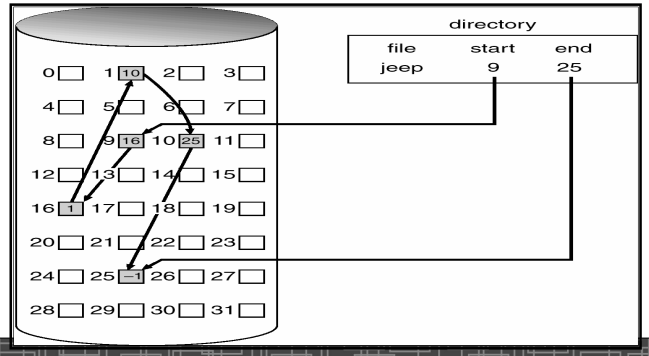
Contiguous Allocation (2)

- Advantages:
  - Disk access in accessing a file is minimal.
  - Both sequential and direct accesses can be done efficiently.
- Disadvantages:
  - *first-fit*, *best-fit* (external fragmentation)
  - Need to determine file size when creating it
- Modified contiguous-allocation scheme
  - When the space is not enough, another chunk of contiguous space, called an extent, is added to the initial allocation.
  - Location of a file includes:

    starting address + block count + link to next extent

Linked Allocation(1)

- Linked Allocation: each file is a linked list of disk blocks
  - Each directory entry has a pointer to the first block of the file
  - Each data block has a pointer to next block

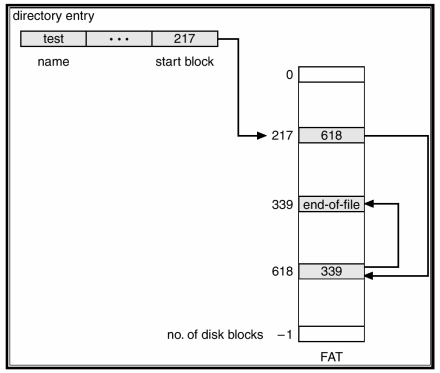


Linked Allocation (2)

- Advantage:
  - No external fragmentation
  - A file can grow without limit (unless disk is full)
- Disadvantages:
  - In sequential access, a disk seek is needed for reading next block. (head movement can be long)
  - In direct access, extremely inefficient.
  - Wasting space for pointers, e.g. 4 out of 512-byte (0.78%)
    - Clustering
  - Reliability: a middle block is lost or damaged → all following blocks can not be located
    - Doubly linked list
    - Save filename and relative block number in each block

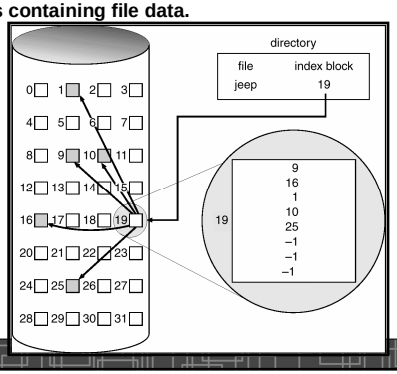
File-Allocation Table

- The table has one entry for each disk block
- Indexed by block number



Indexed Allocation(1)

- Indexed allocation: bringing all the pointers together into one location — an *index block*.
- Each file has its own index block, which is an array of disk-block addresses containing file data.



Indexed Allocation(2)

- Support efficient direct file access
- No external fragmentation
- Need cache index block for better performance
- Huge space waste:
  - One file has a index block
  - Most files on disk are only one or two blocks
- How large the index block should be??
  - Small → less waste
  - small → how to support a large file

EXAMPLE:

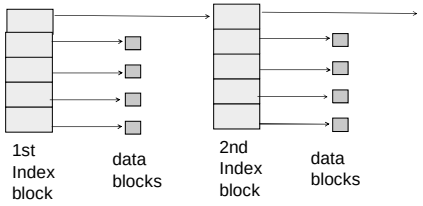
A 512-byte block, each pointer is 4-byte,

one index block can contain at most 128 pointers (512/4=128)

Largest file to be supported by one index block is 512-byte * 128 = 64 KB

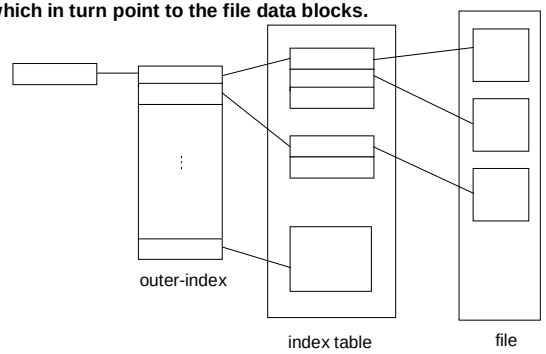
Indexed Allocation: Linked Scheme

- For large file, we can allocate several index blocks, which are also linked as a linked list
- In each index block:
  - A reserved pointer to next index block (*nil* for small files)
  - Other pointers for file data blocks



Indexed Allocation: multilevel index

- A first-level index point to a set of second-level index blocks, which in turn point to the file data blocks.



Previous EXAMPLE: 2-level index can support a file up to 8 MB

Summary: Allocation Methods

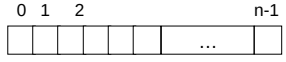
- Contiguous allocation
  - Quick access: only one disk read to reach data block
  - Prior size declaration
  - Serious external fragmentation
- Linked Allocation
  - *n* disk reads to reach *n*-th data block
  - Not proper for random access
- Indexed allocation
  - Two disk reads to reach data block
  - Multi-level indexing requires more disk reads
  - Large space waste due to index block
  - Need cache index block (require large memory space)

Free-Space Management

- File-system maintains a free-space list
  - Records all free disk blocks
- Two data-structures for free-space list
  - Bit vector
  - Linked list
  - Grouping
  - Counting

Free-space management: Bit Vector

- **Bit vector (Bit map) (*n* blocks)**



$$bit[i] = \begin{cases} 1 \Rightarrow block[i] \text{ free} \\ 0 \Rightarrow block[i] \text{ occupied} \end{cases}$$

- **Bit vector is stored word by word on disk**

  First free block number calculation:

  (number of bits per word)*(number of 0-value words starting from beginning) + offset of first 1 bit

- **Bit map requires extra space. Example:**

  block size = 2^{12} bytes

  disk size = $10*2^{30}$ bytes (10 gigabyte)

  n **= $10*2^{30}/2^{12} = 10*2^{18}$ bits (or 320K bytes)**

Free-space management: linked list

- **Link together all free blocks**
- **Keeping a pointer to the first free block (cached in memory)**
- **No need of extra storage.**
- **Cannot get contiguous space easily since traversing the list is not efficient.**