TDIU11

Operating Systems

File-System Implementation

[SGG7/8/9] Chapter 11

Christoph Kessler, IDA,
Linköpings universitet.

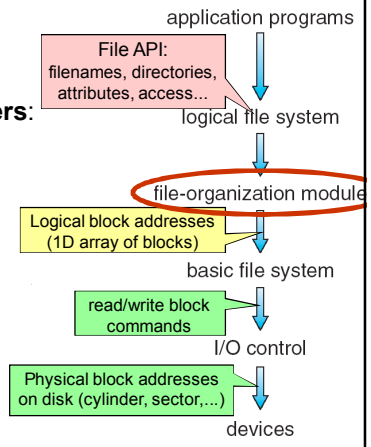
File System Implementation

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Recovery
- Log-Structured File Systems

File-System Structure

- File system resides on secondary storage (disks)
- File system organized into **layers**:
- **File control block** (FCB) – (at the logical FS layer) storage structure consisting of information about a file

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

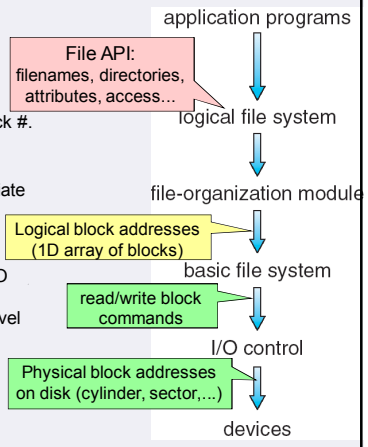


Layered File System

File-System Structure

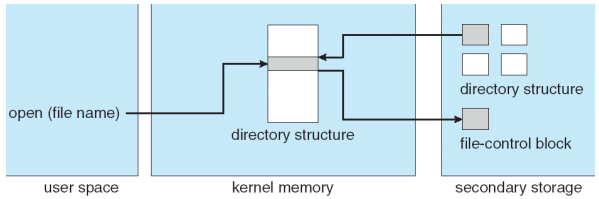
- **File organization module** implements the File system interface:
  - understands files/directories, knows logical addresses, and physical blocks.
  - Translates logical block # to physical block #. Manages free space, disk allocation
- **Basic file system**
  - issues generic commands to the appropriate device driver.
  - Manages memory buffers and caches (allocation, freeing, replacement)
- **Device drivers** manage I/O devices at the I/O control layer.
  - Given "retrieve block 123", outputs low-level commands to hardware controller
- **Disk** provides in-place rewrite and random access. I/O transfers performed in blocks of sectors (usually 512 bytes)

Layering is useful for reducing complexity and redundancy (e.g. several FSs), but adds overhead and can decrease performance.

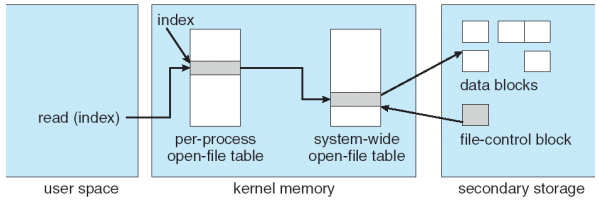


Layered File System

In-Memory File System Structures



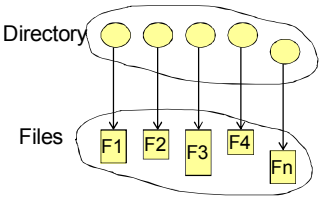
(a) Opening an existing file



(b) Reading an open file

Directory Implementation

Name	File pointer
File1	F1
File2	F2
File3	F3
…	…



- Directories contain pointers to files
- **Linear list** of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute
- Possible ways to speed up lookups
  - Caching lookups
  - Sorting directory entries
  - Hash table structure

Directory Implementation – Details

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - 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 of fixed size, or use a chained-overflow method

Partitions and Mounting

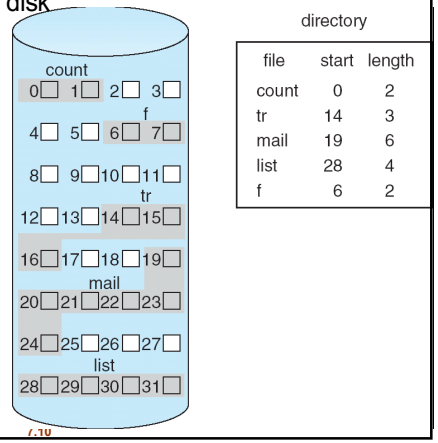
- **Partition** can be a volume containing a file system ("**cooked**") or **raw** – just a sequence of blocks with no file system
- **Boot block** can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system.
  - At boot time, the system does not know FS code yet.
- **Root partition** contains the OS, other partitions can hold other OSes, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system reads device directory and consistency is checked

File Allocation Methods

- An **allocation method** refers to how disk blocks are allocated for files
  - **Contiguous allocation**
  - **Linked allocation**
  - **Indexed allocation**

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
- Simple – need only starting location (block index) and length (# of blocks)
- Random access
- Wasteful of space (dynamic storage-allocation problem)
- Files cannot grow easily
- Works well on CD-ROM



directory

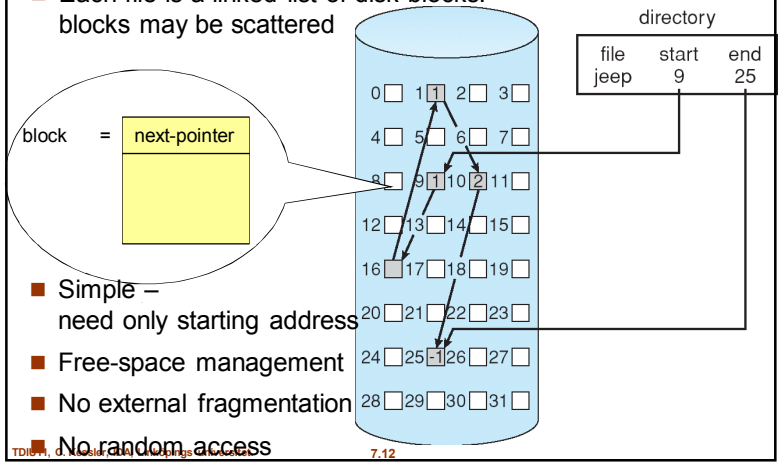
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-Based Systems

- Many newer file systems (e.g., the 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.

Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered
- Simple – need only starting address
- Free-space management
- No external fragmentation
- No random access

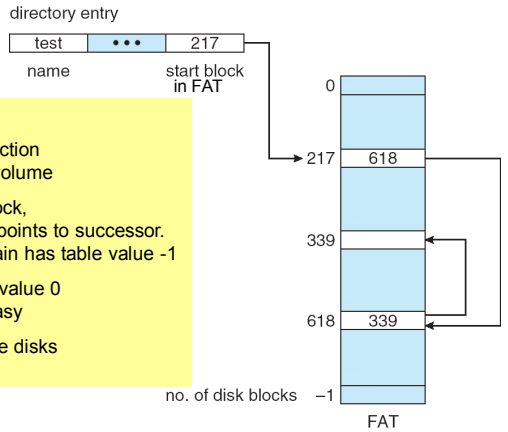


directory

file	start	end
jeep	9	25

File-Allocation Table (FAT)

File-allocation table (FAT) – disk-space allocation used by MS-DOS and OS/2.



Variant of linked allocation:

FAT resides in reserved section at beginning of each disk volume

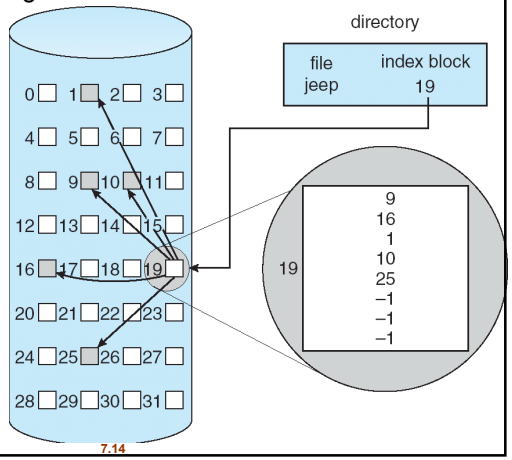
One entry for each disk block, indexed by block number, points to successor. Entry for last block in a chain has table value -1

Unused blocks have table value 0
→ Finding free blocks is easy

Does not scale well to large disks or small block sizes

Indexed Allocation

- Brings all pointers together into an ***index block***

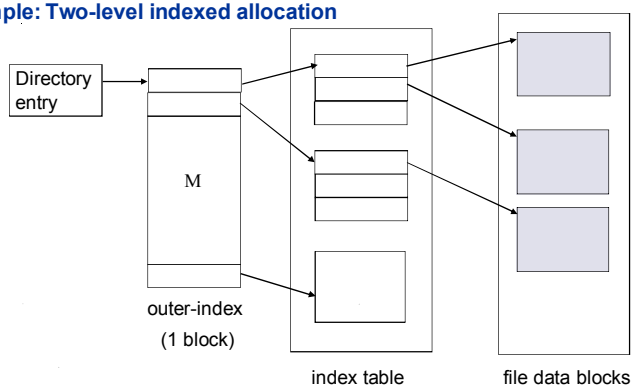


Indexed Allocation (Cont.)

- Direct access once index block is loaded
  - without external fragmentation,
  - but overhead of index block.
- All block pointers of a file must fit into the index block
  - How large should an index block be?
    - Small – Limits file size
    - Large – Wastes space for small files
  - Solution: Multi-level indexed allocation →

Multilevel-indexed allocation

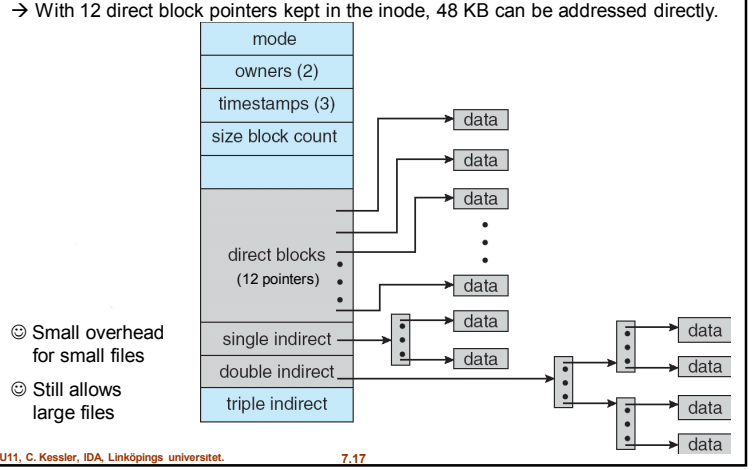
Example: Two-level indexed allocation



With 4K-blocks, could store up to 1,024 four-byte pointers in the outer-index block
→ up to 1024 inner-index blocks
→ up to 1,048,567 data blocks and file size of up to 4GB

Combined Scheme: UNIX inode

Block size 4 KB
→ With 12 direct block pointers kept in the inode, 48 KB can be addressed directly.

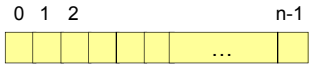


☺ Small overhead for small files

☺ Still allows large files

Free-Space Management (1)

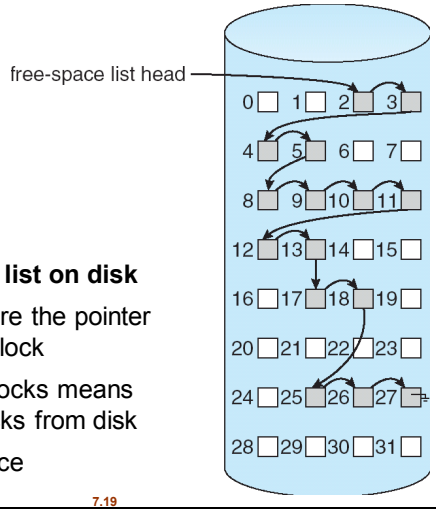
- **Bit vector** (of size *n* for *n* blocks)



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

  - Number of first free block =
    (number of bits per word) * (number of 0-value words) + offset of first 1 bit
- Easy to get contiguous files
- Bit map requires extra space
  - Example: block size = 1 KB = 2^{10} bytes
    disk size = 68 GB ~ 2^{36} bytes
    $n = 2^{36}/2^{10} = 2^{26}$ bits (or 67 MB)
- Inefficient unless entire bit vector is kept in main memory

Free-Space Management (2)

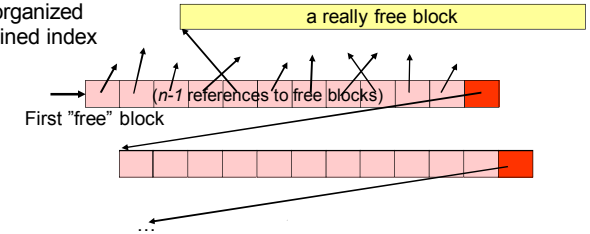


- **Linked free space list on disk**
  - Only need to store the pointer to the first free block
  - Finding *k* free blocks means reading in *k* blocks from disk
  - No waste of space

Free-Space Management (3)

- **Grouping**

Free-list organized using chained index blocks



- **Counting**
  - Often, multiple subsequent blocks are allocated/freed together
  - For sequences of free blocks located subsequently on disk, keep only reference to first one and length of sequence

Efficiency and Performance

Efficiency dependent on:

- File allocation method and directory implementation
- Types of data kept in file's directory entry: e.g., last access
- Pre-allocation (Unix *inodes*) or as-needed metadata allocation
- Varying-size data structures: e.g., clusters of varying sizes

Performance:

- Keeping data and metadata close together to reduce seek time
- **Buffer cache** – in 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

Recovery

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- 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

File System Consistency

- We know: Which blocks are used by each file and which blocks are free
- If a partition was not cleanly unmounted (crash, power failure) these can become inconsistent
- We can try to repair (fsck, scandisk)
  - For each block
    - Find which files use the block
    - Check if the block is marked as free
  - The block is used by 1 file xor is free – OK
  - Two files use the same block – BAD: duplicate the block and give one to each file
  - The block is both used and is marked free – BAD: remove from free list
  - The block is neither free nor used – Wasted block: mark as free

Log Structured File Systems

- **Log structured** (or **journaling**) file systems record each metadata update to the file system as a **transaction**
- All transactions are written to a log
  - A transaction is considered *committed* once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are *asynchronously* written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed

☺ Faster recovery from crash,
☺ removes risk of inconsistency of metadata