OPERATING SYSTEMS

File System Implementation

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

中山大學
SUN YAT-SEN UNIVERSITY

File System Implementation

Contents

File System Structure

Overview

- File structure
  - Logical storage unit
  - Collection of related information
- *File system* resides on secondary storage (disks).
  - Provides user interface to storage, mapping logical to physical.
  - Provides efficient and convenient access to disk by allowing data to be stored, located and retrieved easily.
- The disk provides in-place rewrite and random access.
  - I/O transfers are performed in *blocks* of *sectors* (sector size is usually 512 bytes).
    - In Linux ext2/ext3, block size = 4096 bytes by default.

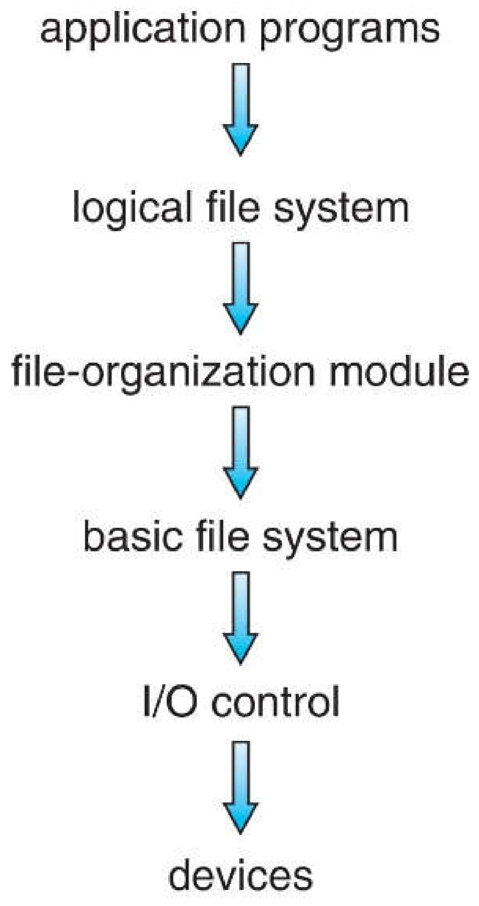
```
$/sbin/tune2fs -l /dev/sda1 | grep "Block size“
```

- *File control block* – (FCB) storage structure consisting of information about a file.
- *Device driver* controls the physical device.

File System Structure

File System Layers

- File system is organized into layers.
  - Each level uses the features of lower levels to create new features for use by higher levels.



File System Structure

File System Layers

- *I/O control* level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system.
  - Device drivers manage I/O devices. Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060”, a device driver outputs low-level hardware specific commands to hardware controller.
- *Basic file system* (called the “block I/O subsystem” in Linux) needs only to issue generic commands to the appropriate device driver to read and write blocks on the storage device. It issues commands to the drive based on logical block addresses like “retrieve block 123”.
  - Also manages memory buffers and caches (allocation, freeing, replacement)
    - Buffers hold data in transit
    - Caches hold frequently used data
- *File organization module* understands files, logical address, and physical blocks.
  - translates logical block number to physical block number
  - manages free space, disk allocation.

File System Structure

File System Layers

- *Logical file system* manages metadata information including all of the file-system structure except the actual data (or contents of the files).
  - Translates file name into file number, file handle, location by maintaining file control blocks (inodes in UNIX)
  - Directory management
  - Protection
- Layering is useful for reducing complexity and redundancy, but adds overhead and can decrease performance.
  - Logical layers can be implemented by any coding method according to OS designer.
- Many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has UFS, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types, with *extended file system* ext2 and ext3 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE.

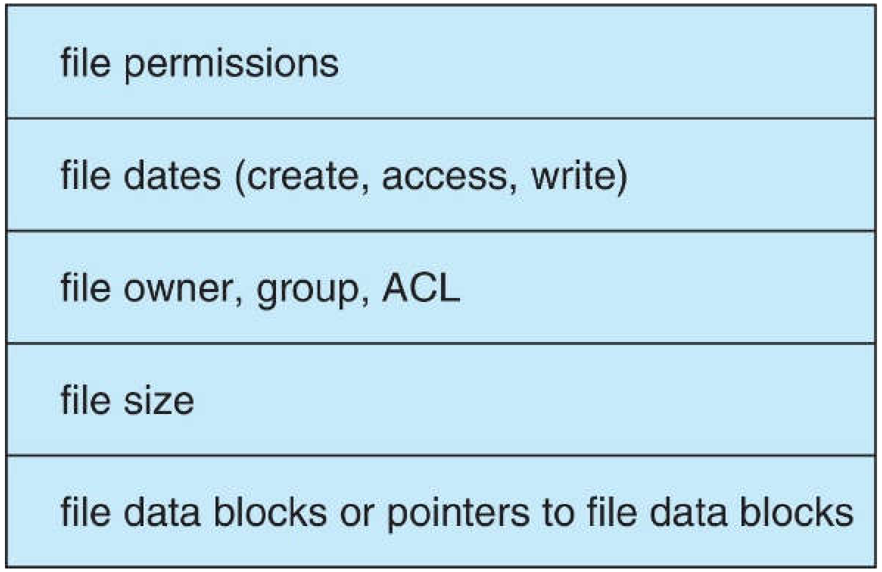
File System Structure

On-Storage Structures

- On storage, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.
- On-storage Structures
  - *Boot control block* (per volume) contains info needed by system to boot OS from that volume.
    - Needed if volume contains OS, usually first block of volume.
    - Also called *boot block, partition boot sector*.
  - *Volume control block* (per volume) contains volume details
    - Total number of blocks in the volume, block size, free-block count and pointers, free-FCB count and pointers.
    - Also called superblock, master file table.
  - *Directory structure* (per file system) is used to organize the files.
    - In UFS, including file names and associated inode numbers.
    - In NTFS, stored in the master file table.

File System Structure

- **On-Storage Structures**
  - On-storage Structures (cont.)
    - *FCB* (per file) contains many details about the file.
      - It has a unique identifier number to allow association with a directory entry.
      - inode number, permissions, size, dates, etc. in UNIX.
      - NTFS stores the information in master file table using relational DB structures.


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

A Typical File Control Block (FCB)

File System Structure

In-Memory Structures

- The in-memory information is 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. Several types of structures may be included.
  - *Mount table* contains information about each mounted volume.
  - *Directory-structure cache* holds the directory information of recently accessed directories.
  - *System-wide open-file table* contains a copy of the FCB of each open file, as well as other information.
  - *Open-file table* (per process) contains pointers to the appropriate entries in the system-wide open-file table, as well as other information, for all files the process has open.
  - *Buffers* hold file-system blocks when they are being read from or written to a file system.

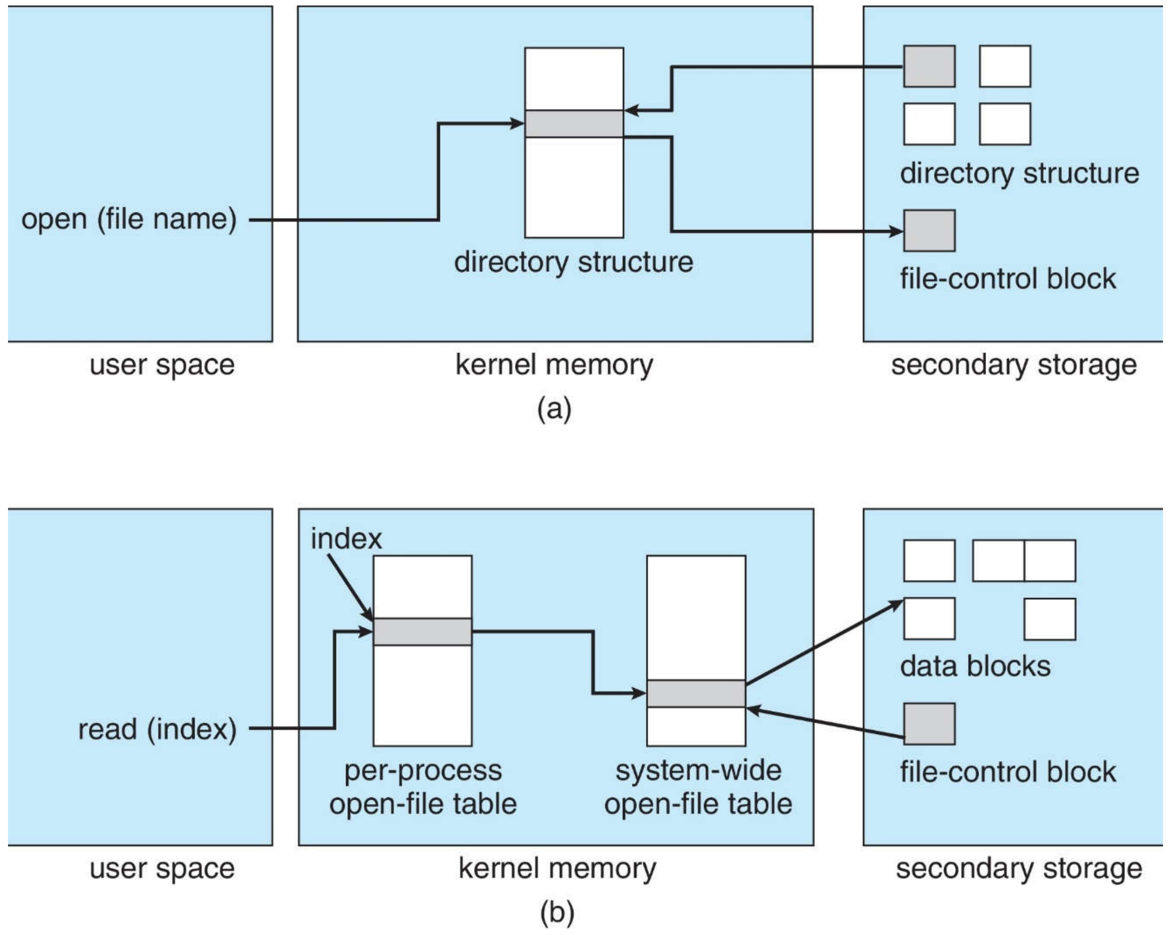
File System Structure

Usage of File System Structures

- The following figure illustrates the necessary file system structures provided by the operating systems.
  - Figure (a) refers to opening a file
  - Figure (b) refers to reading a file
- Plus buffers hold data blocks from secondary storage.
- Open returns a file handle for subsequent use
- Data from read is eventually copied to specified user process memory address.

File System Structure

Usage of File System Structures



(a) File open. (b) File read.

File System Structure

- **Partitions and Mounting**
  - *Partition* can be a volume containing a file system ("cooked") or *raw*.
    - Raw partition is 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.
    - It may also be a boot management program for multi-OS booting.
  - *Root partition* contains the operating system, other partitions can hold other operating systems, other file systems, or just be raw.
    - Root partition is mounted at boot time.
    - Other partitions can mount automatically or manually.
  - At mount time, file system consistency is checked.
    - Is all metadata correct?
      - If not, fix it, try again.
      - If yes, add to mount table, allow access.

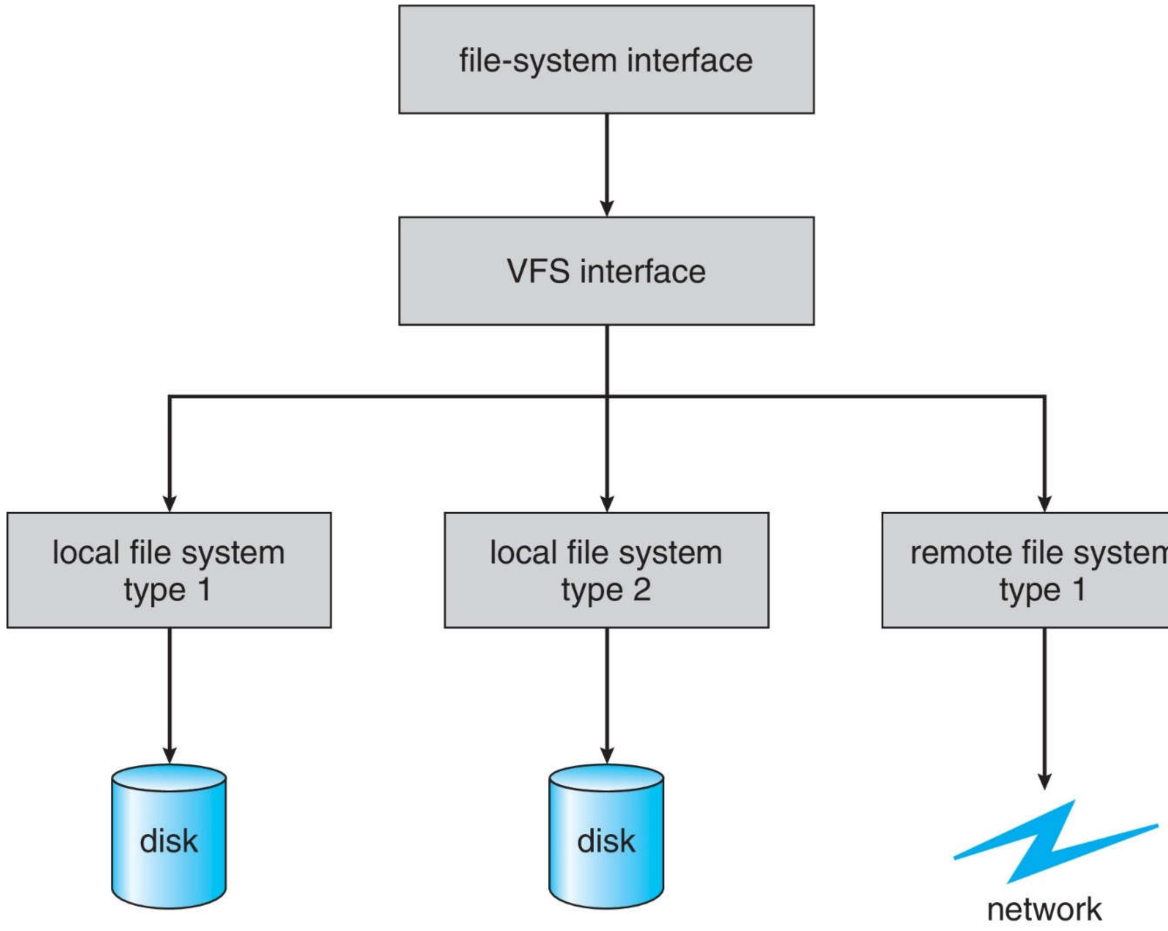
File System Structure

- **Virtual File Systems**
  - Virtual File Systems (VFS) on UNIX provide an object-oriented way of implementing file systems.
  - VFS allows the same system call interface (the API) to be used for different types of file systems.
    - VFS separates file-system generic operations from implementation details.
    - Implementation can be one of many file systems types, or network file system.
      - Implements vnodes which hold inodes or network file details.
    - Then dispatches operation to appropriate file system implementation routines
  - The API is to the VFS interface, rather than any specific type of file system.

File System Structure

Virtual File Systems

- Schematic View of a Virtual File System.



Virtual File Systems

- Virtual File System Implementation
  - For example, Linux has four object types:
    - inode, file, superblock, dentry
  - VFS defines set of operations on the objects that must be implemented.
    - Every object has a pointer to a function table.
      - Function table has addresses of routines to implement that function on that object.
    - For example:

```
int open()          open a file
int close()         close an already-open file
ssize_t read()      read from a file
ssize_t write()     write to a file
int mmap()          memory-map a file
```

File System Structure

Directory Implementation

- The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system.
- Linear List
  - A linear list of file names is used 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
  - A 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

Allocation Methods

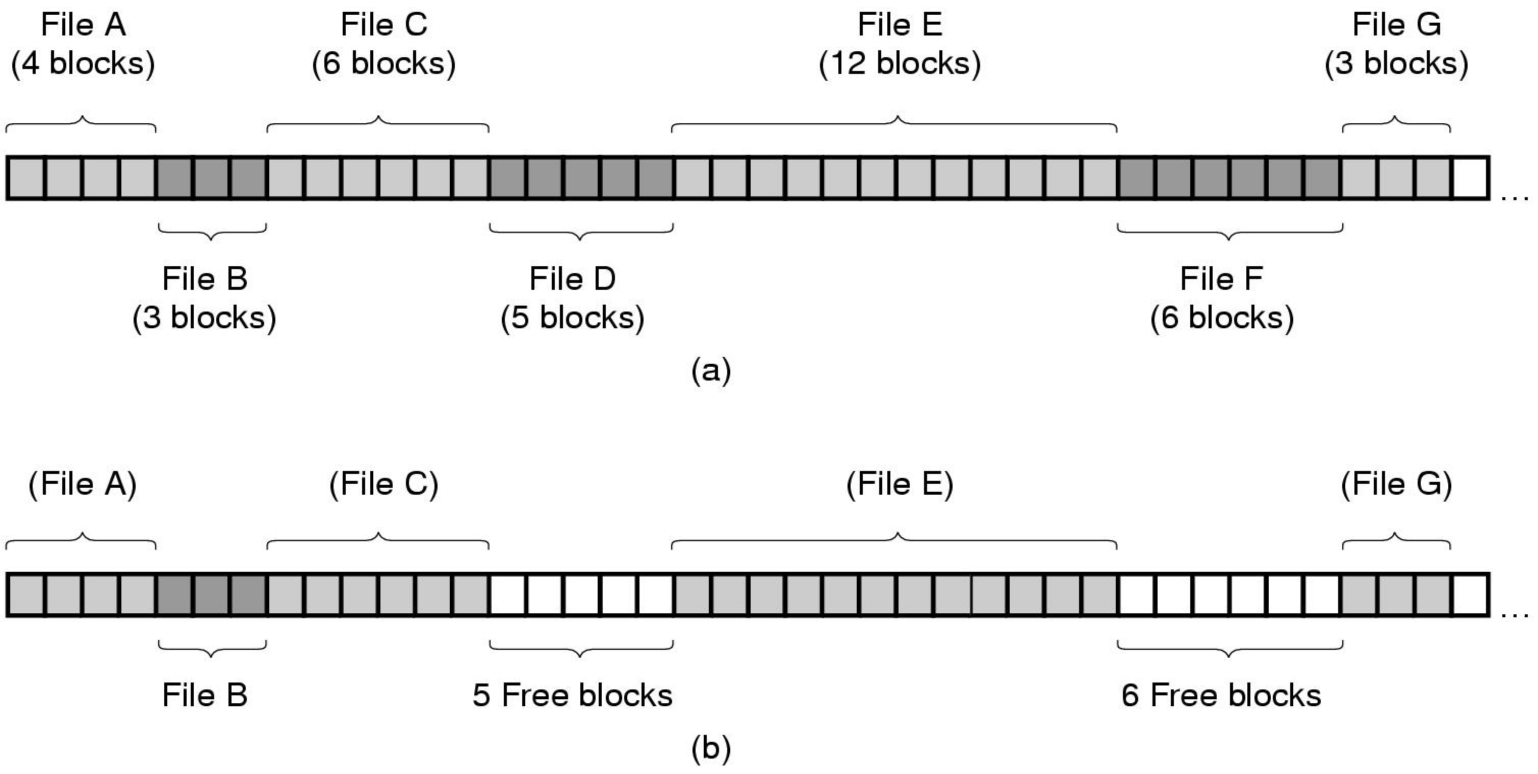
Allocation Methods

- An allocation method refers to how disk blocks are allocated for files.
  1. (1) Contiguous allocation
  2. (2) Linked allocation /Chained allocation
  3. (3) Indexed allocation
  4. (4) Combined schemes

Allocation Methods

Contiguous Allocation

- Example.



(a) Contiguous allocation of disk space for 7 files.
(b) The state of the disk after files D and F have been removed.

Allocation Methods

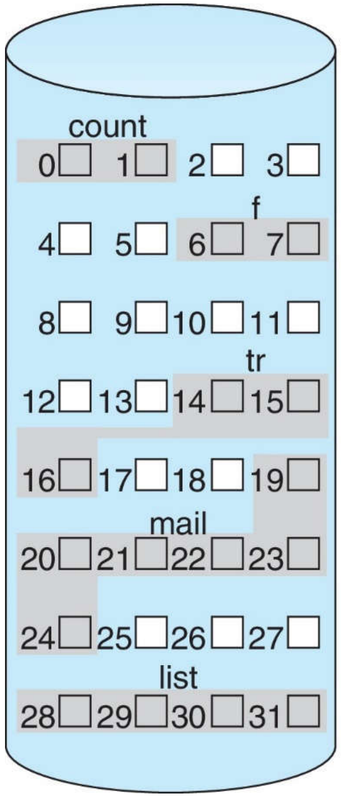
- **Contiguous Allocation**
  - Each file occupies a set of contiguous blocks on the disk.
    - Simple: only starting location (block number) and length (number of blocks) required
    - Enables random access
    - Best performance in most cases.
  - Disadvantages:
    - Wasteful of space (dynamic storage-allocation problem).
    - Files cannot grow.
  - Problems include finding space for file, knowing file size, external fragmentation, need for *compaction off-line* (down time) or *on-line.*

Allocation Methods

- **Contiguous Allocation**
  - Mapping from logical to physical addresses (512 bytes per block)

    $$\texttt{logical\_address} = Q \times 512 + R$$ (assembled in bytes)

    - Block number to be accessed is Q + starting address (base address).
    - Displacement (offset) into block of file is R.
  - Example.

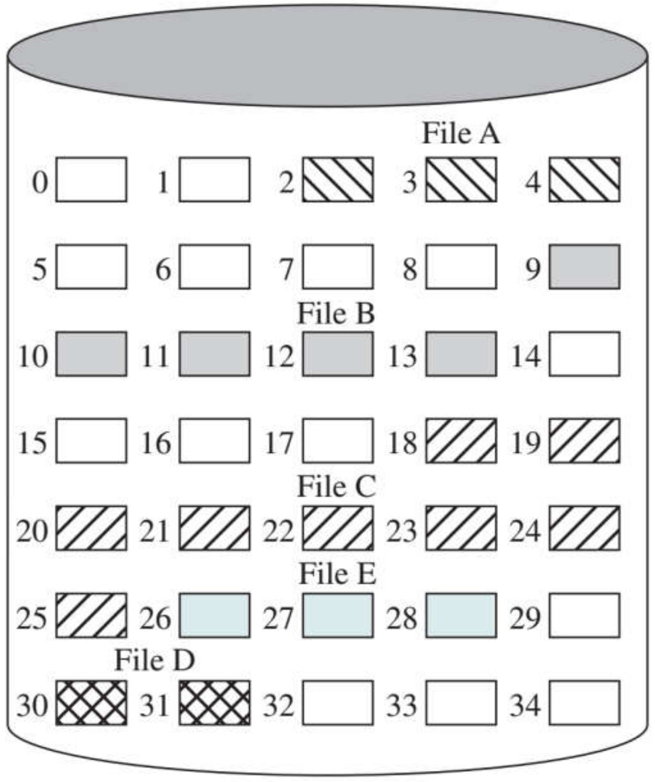


directory

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

Allocation Methods

- **Contiguous Allocation**
  - Example.

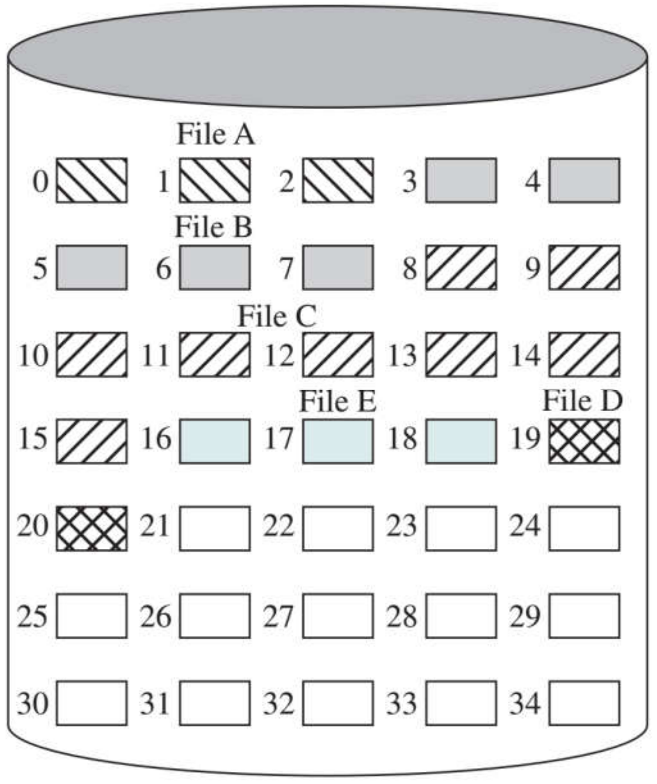


File allocation table

File name	Start block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Allocation Methods

- **Contiguous Allocation**
  - Example (after compaction).



File allocation table

File name	Start block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Allocation Methods

- **Contiguous Allocation**
  - Extent-Based File Systems
    - Many newer file systems (SPARC Veritas file system, Linux 2.6.19 EXT4, etc.) 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.

Allocation Methods

Linked Allocation

- Linked allocation aka chained allocation
  - Each file is a linked list of disk blocks
    - These blocks may be scattered anywhere on the disk.
    - Each block contains a pointer to the next block.
    - File ends at a NIL pointer.
- Advantages:
  - No compaction, no external fragmentation.
  - Free space management is called when new block needed.
  - Efficiency is improved by *clustering* blocks into groups, but internal fragmentation increases.
  - Simple: need only starting address
  - Free-space management: no waste of space
- Disadvantages:
  - No random access
  - Locating a block can take many I/O times and disk seeks.
  - Some systems periodically consolidate (合并) files.
  - Reliability can be a problem.

Allocation Methods

- **Linked Allocation**
  - Mapping from logical address to physical addresses (512 bytes per block, 4 bytes for block address pointer)

```
logical_address = Q × (512 – 4) + R.
```

    - Block to be accessed is the Q-th block in the linked list of blocks representing the file.

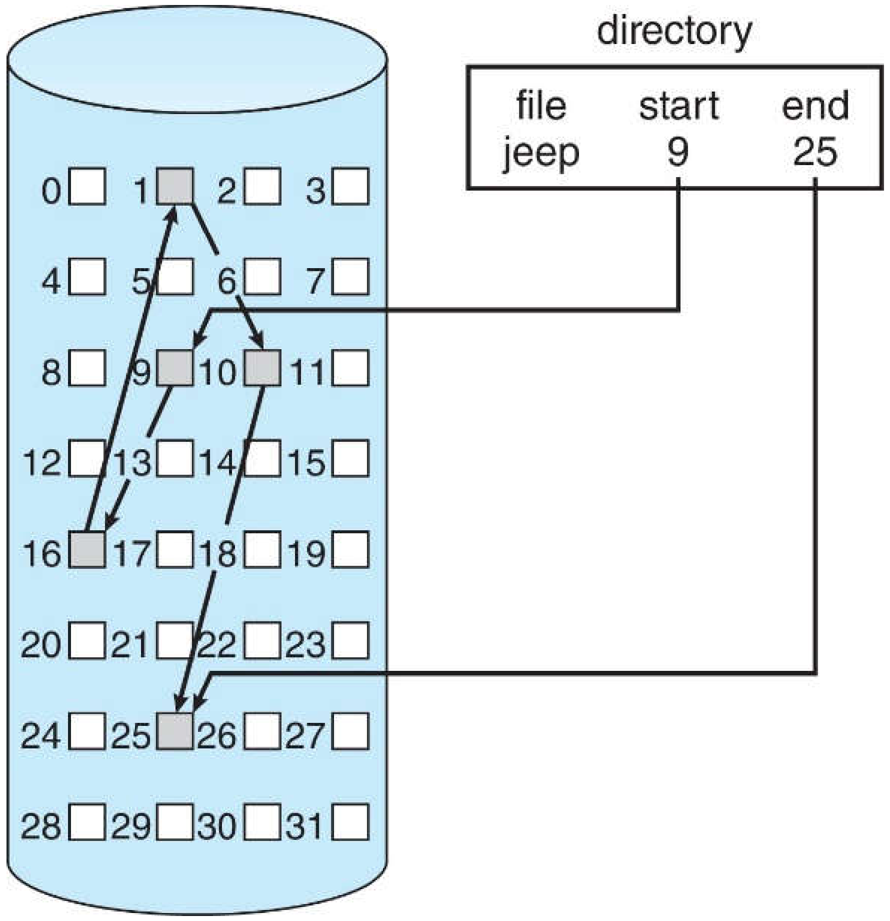
```
Q = logical_address % (512 – 4).
```

    - The first 4 bytes of each block are reserved for the block pointer. Displacement into block of file is

```
offset = R + 4.
```

Allocation Methods

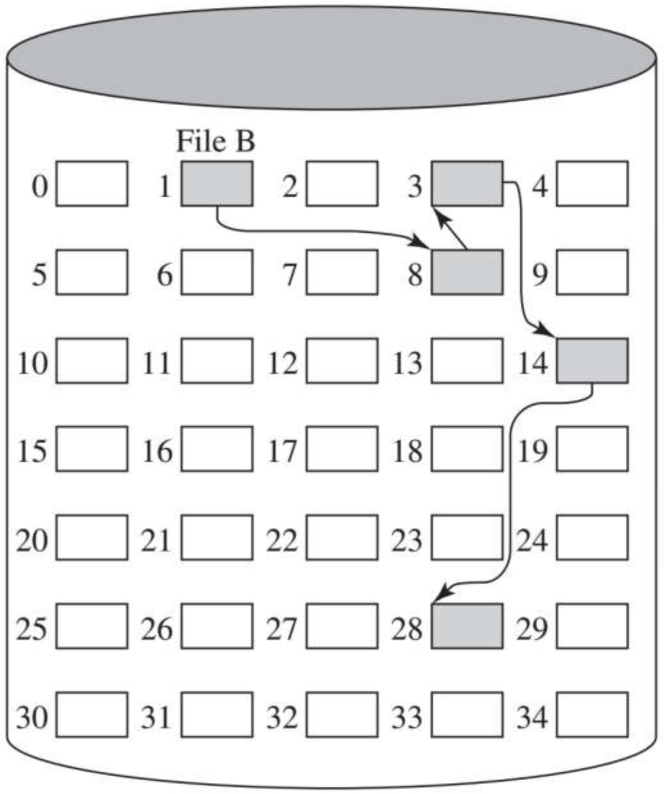
- **Linked Allocation**
  - Example.



Allocation Methods

Linked Allocation

- Example.

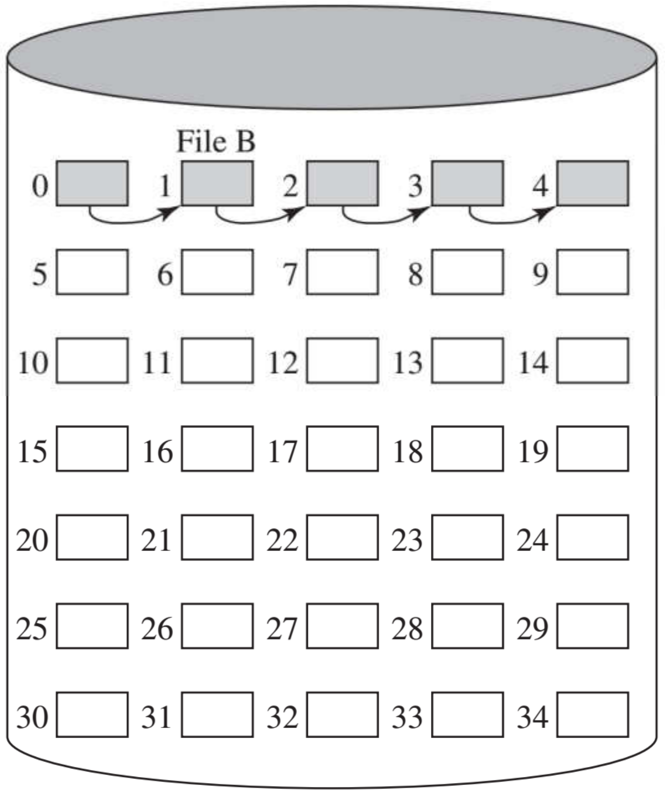


File allocation table

File name	Start block	Length
● ● ●	● ● ●	● ● ●
File B	1	5
● ● ●	● ● ●	● ● ●

Allocation Methods

- **Linked Allocation**
  - Example (after consolidation).



File allocation table

File name	Start block	Length
• • •	• • •	• • •
File B	0	5
• • •	• • •	• • •

Allocation Methods

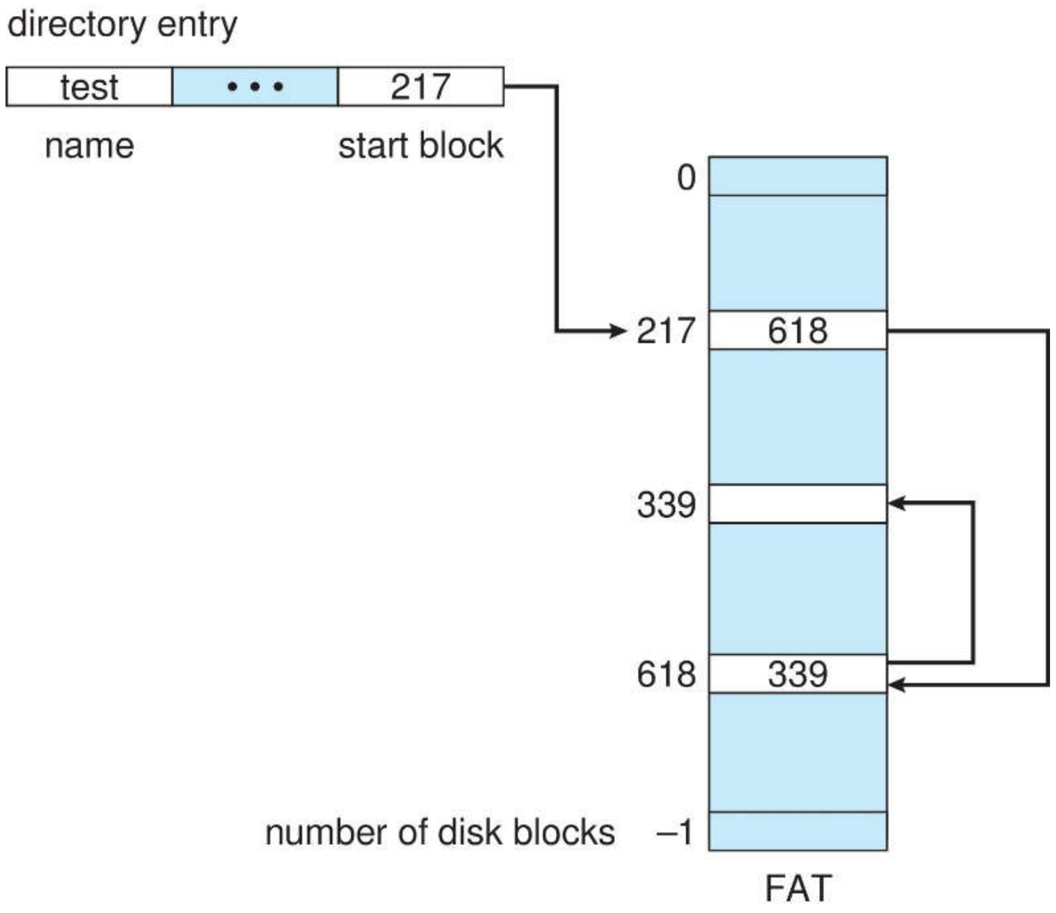
- **Linked Allocation**
  - File Allocation Table (FAT)
    - An important variation on linked allocation is the use of a *file-allocation table* (FAT). A section of storage at the beginning of each volume is set aside to contain FAT (a volume-wide data structure).
    - FAT has one entry for each block in the volume and is indexed by block number.
      - The directory entry contains the block number of the first block of the file.
      - The FAT entry indexed by that block number contains the block number of the next block in the file.
      - This chain 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 value of 0.
      - Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value.

Allocation Methods

- **Linked Allocation**
  - File Allocation Table (FAT)
    - The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached.
      - The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks.
    - A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

Allocation Methods

- **Linked Allocation**
  - File Allocation Table (FAT)
    - An illustrative example: a file consisting of disk blocks 217, 618, and 339.

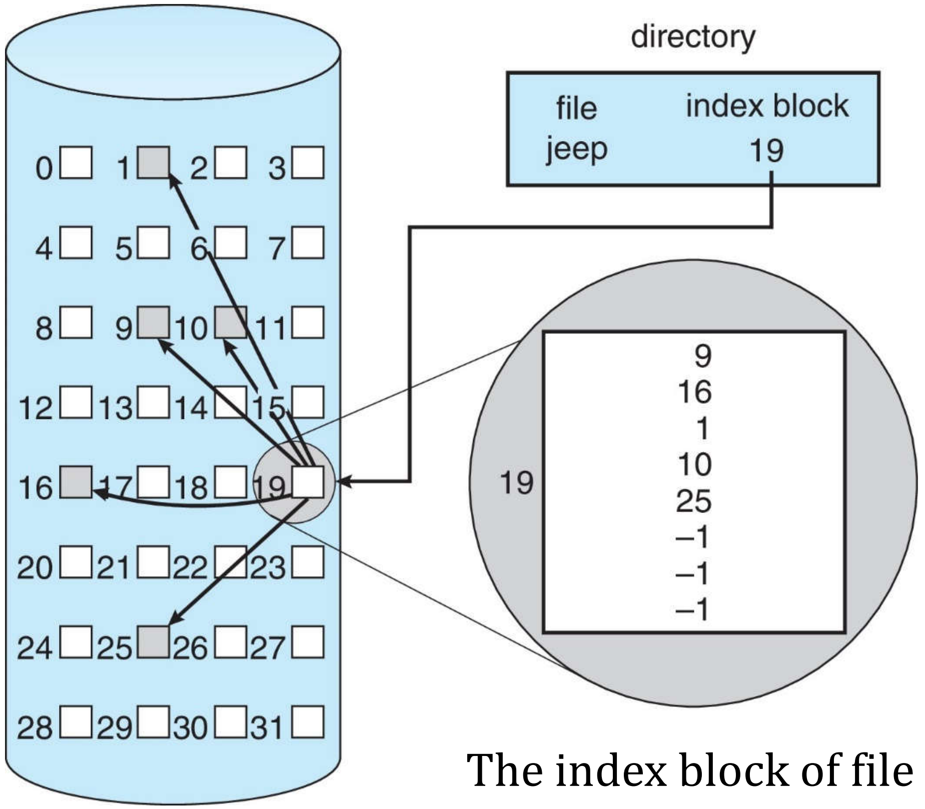


Allocation Methods

- **Indexed Allocation**
  - Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation.
    - In the absence of a FAT, linked allocation cannot support efficient direct access, since the blocks must be retrieved in order.
  - *Indexed allocation* solves this problem by bringing all the pointers together into one location: the *index block*.
    - Each file has its own index block, which is an array of storage-block addresses. The i-th entry in the index block points to the i-th block of the file.
    - The address of the index block is involved in the directory structure.
  - Indexed allocation supports direct access without suffering from external fragmentation, but does suffer from wasted space.
    - An entire index block must be allocated, even if the file has only one or two blocks. This point raises the question of how large the index block should be, as small as possible?

Allocation Methods

- **Indexed Allocation**
  - Example.

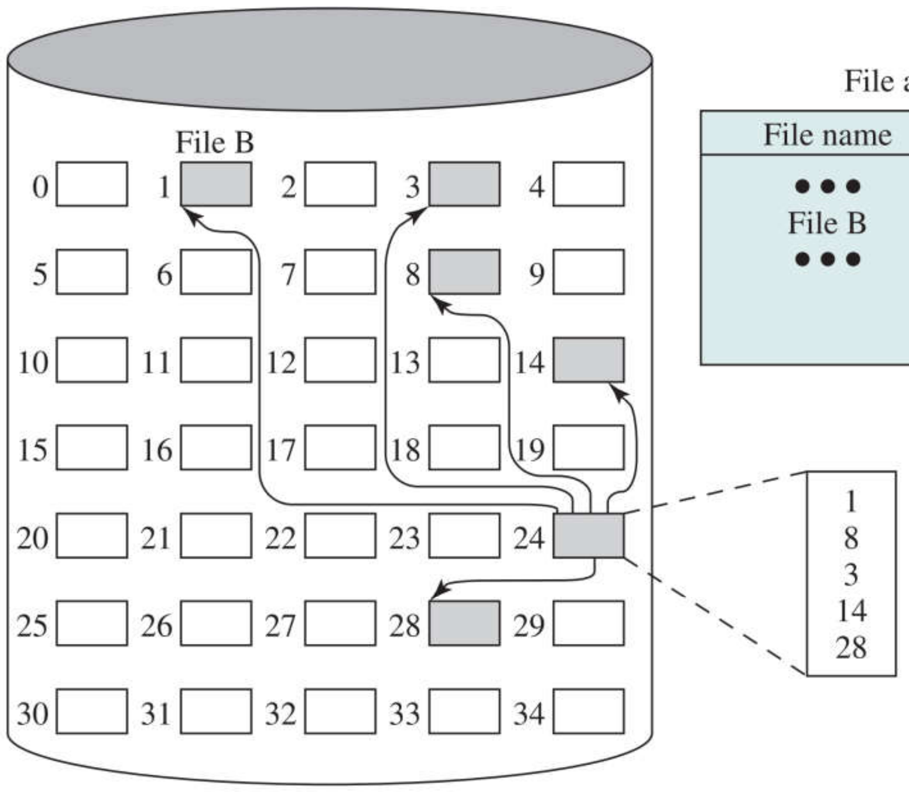


The index block of file jeep

Allocation Methods

Indexed Allocation

- Example (block-size portion version).

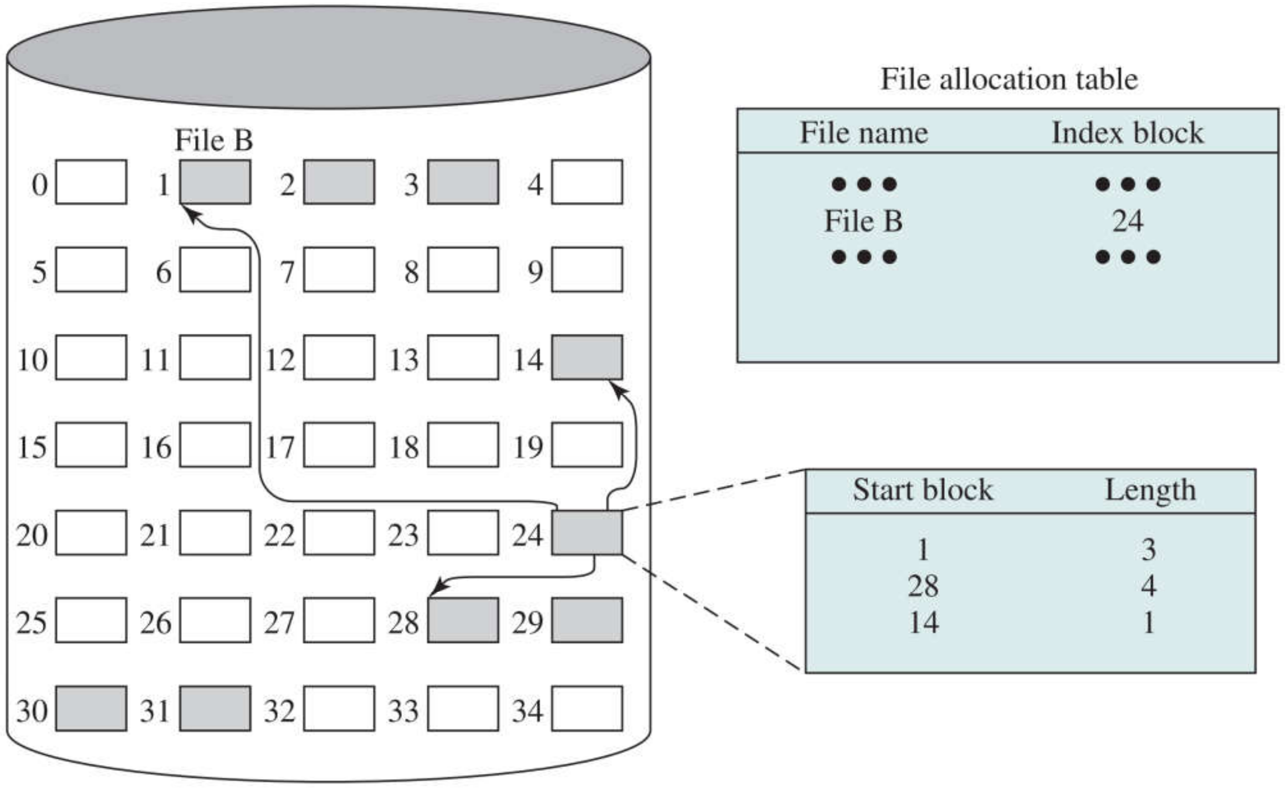


File allocation table

File name	Index block
•••	•••
File B	24
•••	•••

Allocation Methods

- **Indexed Allocation**
  - Example (variable-size portion version).



File allocation table

File name	Index block
● ● ●	● ● ●
File B	24
● ● ●	● ● ●

Start block	Length
1	3
28	4
14	1

Allocation Methods

- **Indexed Allocation**
  - Index Block Schemes
    - One block scheme example
      - Mapping from logical to physical addresses in a file of maximum size of 256K bytes (it is 512 blocks with block size of 512 bytes), we need only 1 block (index block) for index table with 512 entries (1 byte for each entry):

        ```
        logical_address = Q×512 + R.
        ```

      - Q = displacement into index block of index table
      - R = displacement into block of file

Allocation Methods

- **Indexed Allocation**
  - Index Block Schemes
    - Linked scheme
      - Mapping from logical to physical addresses in a file of unbounded size and block size of 512 bytes, we use linked scheme to link blocks of index table.

$$\texttt{logical\_address} = Q_1 \times (512 \times 511) + Q_2 \times 512 + R_2$$

      - Q_1 = block number of index table

$$Q_1 = \texttt{logical\_address} \ \% \ (512 \times 511) + R_1$$

      - Q_2 = displacement into index block of index table

$$Q_2 = R_1 \ \% \ 512 + R_2$$

      - R_2 = displacement into block of file

Allocation Methods

- **Indexed Allocation**
  - Index Block Schemes
    - Two-level index scheme
      - 4K-byte blocks could store 2^{10} of 4-byte pointers in outer index
        - number of blocks of file being represented up to

          $$2^{10} \times 2^{10} = 1{,}048{,}567$$
        - file size up to

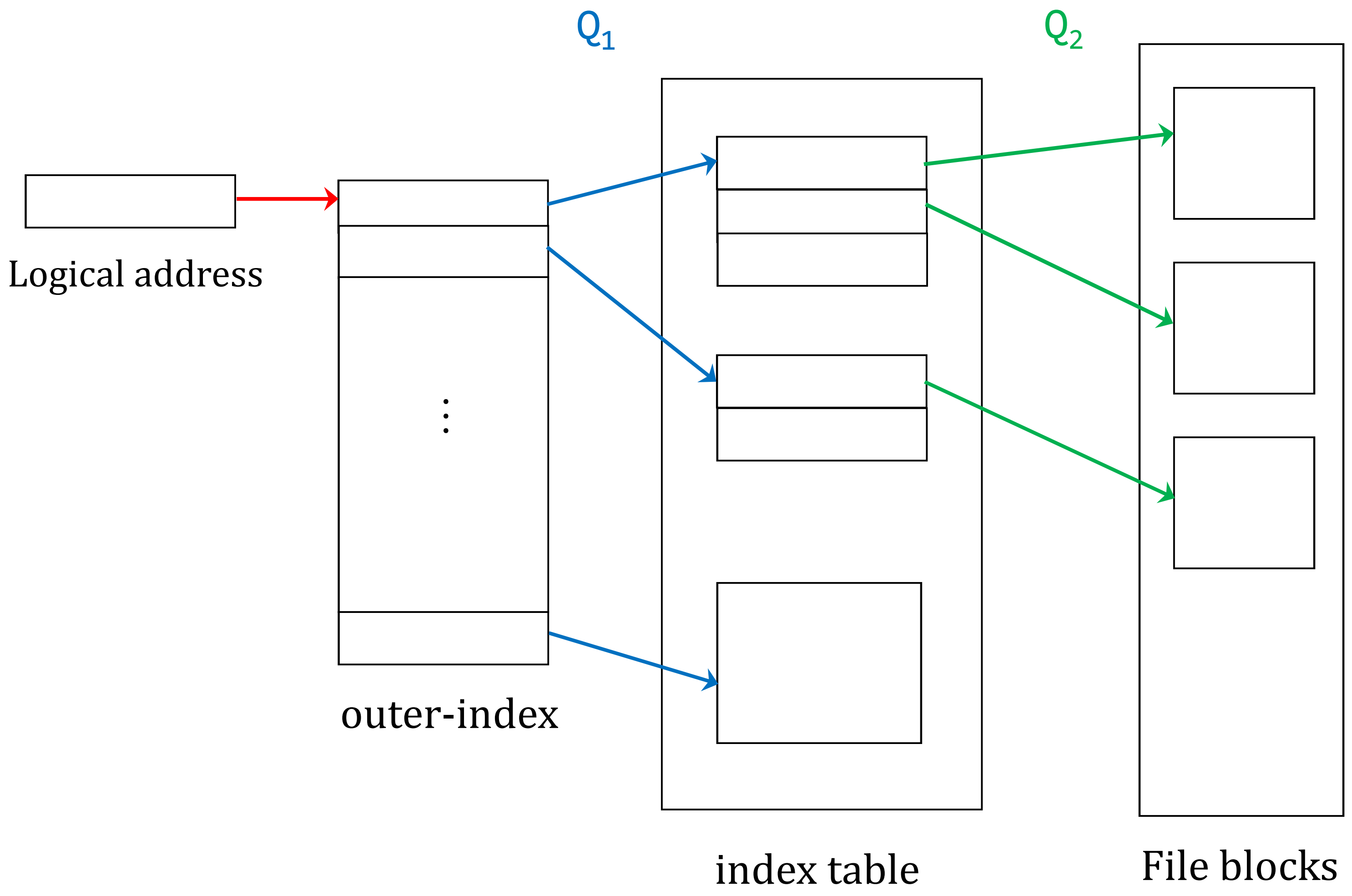
          $$4\ (KB) \times 2^{10} \times 2^{10} = 4\ (GB)$$
      - Mapping from logical to physical addresses

        $$\text{logical\_address} = Q_1 \times (1024 \times 1024) + Q_2 \times 1024 + R_2$$
      - Q_1 = displacement into outer-index
      - Q_2 = displacement into block of index table
      - R_2 = displacement into block of file

Allocation Methods

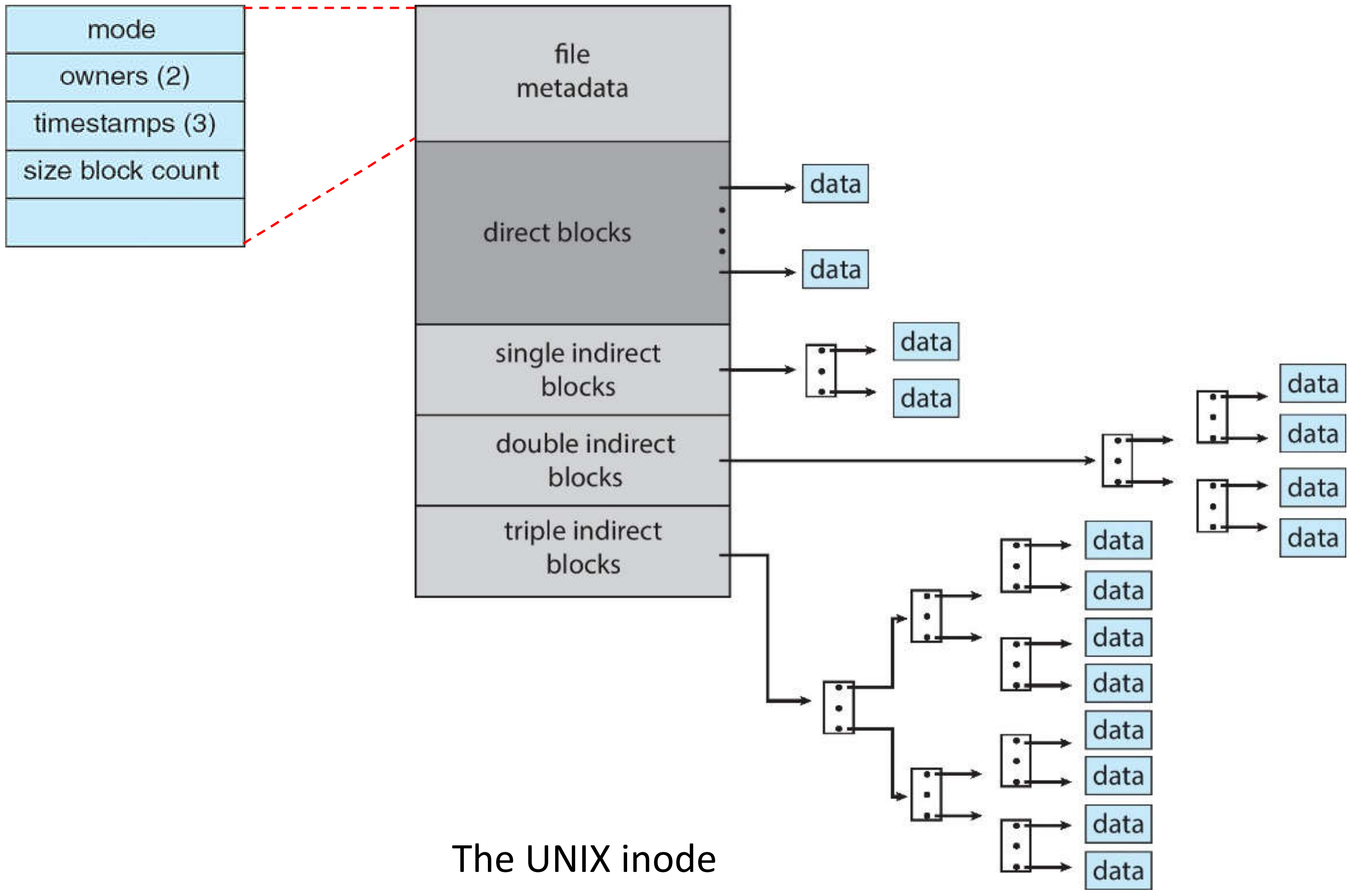
Indexed Allocation

- Two-level index.



Allocation Methods

- **Indexed Allocation**
  - Combined Scheme: UNIX UFS (4K bytes per block).



The UNIX inode

Allocation Methods

Comparison

- Comparison of file allocation methods.

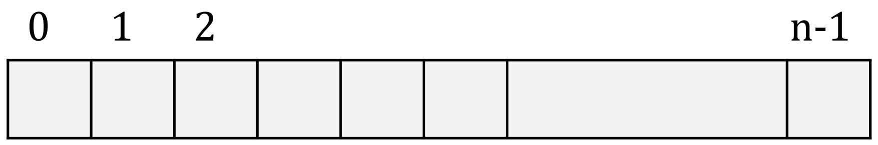
	Contiguous	Chained	Indexed	
Pre-allocation?	Necessary	Possible	Possible	
Fixed or variable size portions?	Variable	Fixed blocks	Fixed blocks	Variable
Portion size	Large	Small	Small	Medium
Allocation frequency	Once	Low to high	High	Low
Time to allocate	Medium	Long	Short	Medium
File allocation table size	One entry	One entry	Large	Medium

Allocation Methods

- **Performance**
  - The best method depends on the file access type.
    - Contiguous allocation is great for sequential and random.
    - Linked allocation is good for sequential, not for random.
    - If access type is declared at creation then we can select either contiguous or linked allocation.
  - Indexed allocation is more complex.
    - Single block access could require 2 index block reads then data block read.
    - Clustering can help improve throughput, reduce CPU overhead.
  - Adding thousands of extra instructions to the operating system to save just a few disk-head movements is reasonable.
    - Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS (million instructions per second)
    - A typical disk drive is at 250 IOPS (I/O times per second).
      - 159,000 MIPS / 250 = 630 million instructions per disk I/O.
    - Fast SSD drives provide 60,000 IOPS.
      - 159,000 MIPS / 60,000 = 2.65 MI per disk I/O.

Free-Space Management

- File system maintains *free-space list* to track available blocks/clusters
    - Using term “block” for simplicity
- Bitmap with n blocks
    - bitmap (位图) is aka bit vector, or bit table.



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

- Example. Consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be

```
001111001111110001100000011100000 ...
```

Free-Space Management

- Bitmap with n blocks
  - How to find the first free block on a system that uses a bit vector to allocate space?
    - Sequentially check each word in the bitmap to see whether that value of the word is not 0, since a 0-valued word contains only 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block.
  - The calculation of the located block number is

    (number_of_bits_per_word) * (number_of_0-value_words) + (offset_of_first_1_bit)

    - CPUs have instructions to return the offset of first “1” bit within a word.

Free Space Management

- **Free-Space Management**
  - Bitmap with n blocks
    - It requires extra space to save bitmap table on disk or in main memory.
    - Example.

$$\text{block size} = 512 \text{ bytes} = 2^9 \text{ bytes}$$
$$\text{disk size} = 2^{34} \text{ bytes (16 gigabyte)}$$
$$\text{number of blocks n} = 2^{34}/2^9 = 2^{25}$$
$$\text{bitmap size} = 2^{25} \text{ bits} = 2^{22} \text{ bytes} = 4\text{MB}.$$

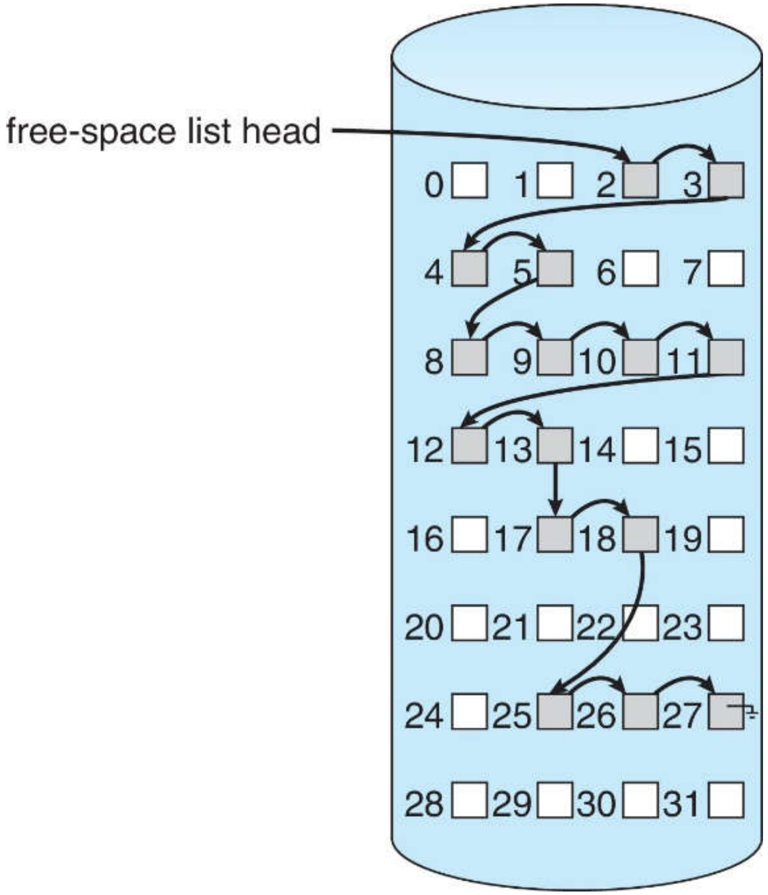
    - 4MB is a hefty chunk for main memory. The alternative is to put the bitmap table on disk.
      - A 4MB bit table would require 2^{13} = 8192 disk blocks. We can’t afford to search that amount of disk space every time a block is needed, so keeping a bit table resident in memory is indicated.
    - Even when the bit table is in main memory, an exhaustive search of the table can slow file system performance to an unacceptable degree.

Free Space Management

- **Free-Space Management**
  - Free Block List (Linked List)
    - A Free Block List links together all the free blocks, keeping a pointer to the first free block in a special location in the file system and caching it in memory. Every free block contains a pointer to the next free block.
    - To traverse the list, we must read each block, which requires substantial I/O time on HDDs.
      - In fact, traversing the free list is not a frequent action. Usually, the operating system simply needs a free block and the first block in the free list is used.
    - Cannot get contiguous space easily
    - No waste of space

Free-Space Management

- Free Block List (Linked List)
  - Recall our earlier example, in which blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated.



Free-Space Management

- Grouping and Counting
  - Grouping
    - A modification of the free-list approach stores the addresses of n free blocks in the first free block. The first n - 1 of these blocks are actually free, and the last block contains the addresses of another n free blocks, and so on. The addresses of a large number of free blocks can now be found quickly.
  - Counting
    - It is the fact that space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering.
      - Keep the address of the first free block i and the number (n) of free contiguous blocks that follow block i.
      - Each entry in the free-space list then consists of a device address and a count.
    - These entries can be stored in a balanced tree, rather than a linked list, for efficient lookup, insertion, and deletion.

Free Space Management

- **Free-Space Management**
  - Space Maps
    - Oracle's ZFS (Zettabyte File System, Sun Solaris 10) was designed to encompass huge numbers of files, directories, and even file systems. On these scales, metadata I/O can have a large performance impact.
      - Full data structures like bitmaps couldn't fit thousands of I/Os.
    - Divides device space into *metaslab* units and manage them.
      - Given volume can contain hundreds of metaslabs.
      - Each metaslab has an associated space map.
        - Uses counting algorithm
      - But records to log file rather than file system.
        - Log of all block activity, in time order, in counting format
    - Metaslab activity is to load space map into memory in balanced-tree structure, indexed by offset.
      - Replay log into that structure
      - Combine contiguous free blocks into single entry.

Other Considerations

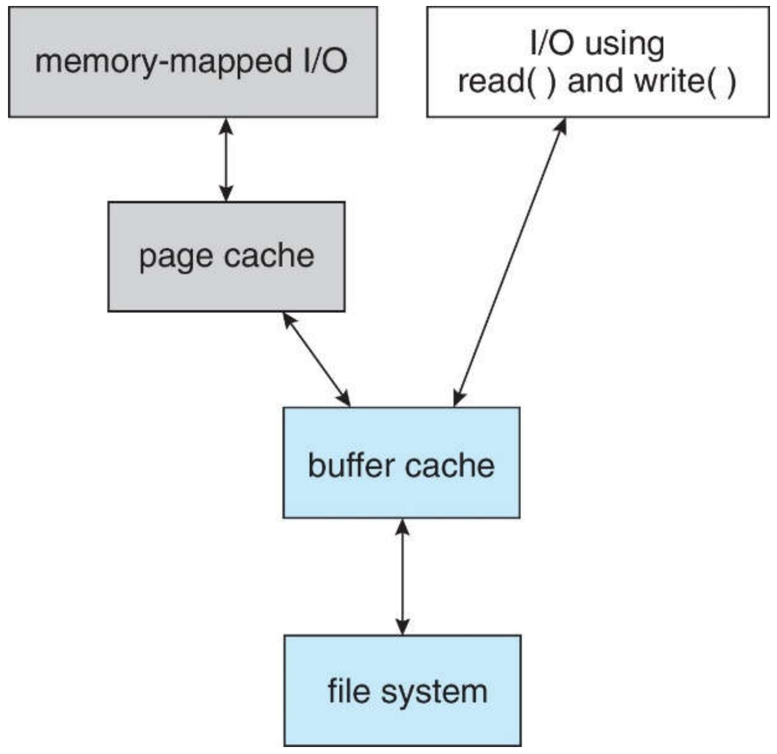
- **Efficiency and Performance**
  - Efficiency depends on
    - Disk allocation and directory algorithms
    - Types of data kept in file’s directory entry
    - Pre-allocation or as-needed allocation of metadata structures
    - Fixed-size or varying-size data structures.
  - Performance
    - Disk cache
      - separate section of main memory for frequently used blocks.
    - Free-behind and read-ahead
      - techniques to optimize sequential access.
    - Improve PC performance by dedicating section of memory as virtual disk or RAM disk.

Other Considerations

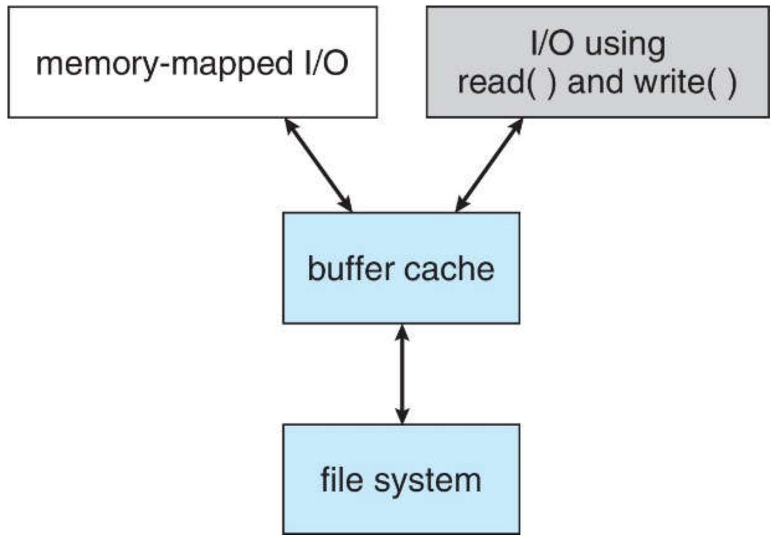
- **Efficiency and Performance**
  - Page-Cache
    - A page-cache caches pages rather than disk blocks using virtual memory techniques.
    - Memory-mapped I/O uses a page-cache.
    - Routine I/O through the file system uses the buffer(disk)-cache.
  - Unified Buffer Cache
    - A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid *double caching.*

Other Considerations

- **Efficiency and Performance**
  - Unified Buffer Cache.



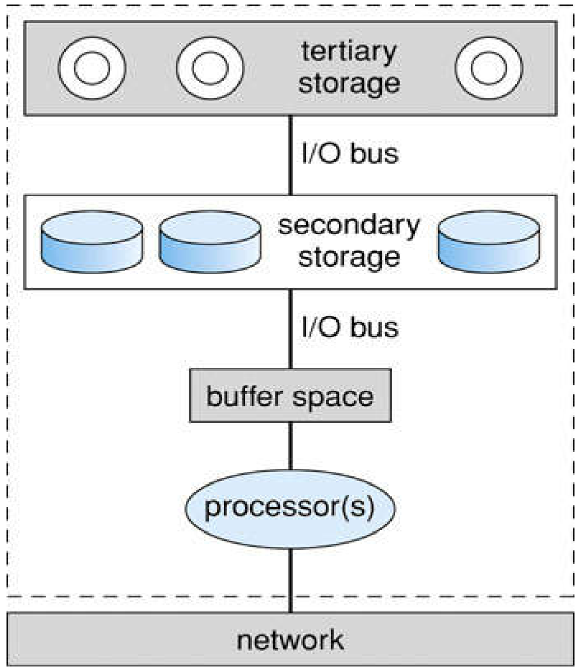
I/O without a unified buffer cache



I/O using a unified buffer cache

Other Considerations

- **Recovery**
  - Consistency checking
    - compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.
  - 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.



Storage Backup

Other Considerations

- **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 chance of inconsistency of metadata.