Operating Systems Principles

cosc1112/cosc1114
School of Science
Semester 2, 2017

Lecture 09 – File system

Dr. Ke Deng

ke.deng@rmit.edu.au

RMIT UNIVERSITY

Outline

- File Concept
- Access Methods
- Disk and Directory Structure
- File-System Mounting
- File Sharing
- Protection
- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance

Objectives

- To explain the function of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures
- To explore file-system protection
- To describe the details of implementing local file systems and directory structures
- To describe the implementation of remote file systems
- To discuss block allocation and free-block algorithms and trade-offs

File Concept

- Contiguous logical address space
- Types:
  - Data
    - Numeric
    - Character
    - Binary
  - Program
- Contents defined by file's creator
  - Many types
    - Consider **text file, source file, executable file**

File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum
- Information kept in the directory structure

File Operations

- File is an **abstract data type**
- **Create**
- **Write –** at **write pointer** location
- **Read –** at **read pointer** location
- **Reposition within file - seek**
- **Delete**
- **Truncate –** allows all attributes to remain unchanged while set file length to be 0 and file space released
- ***Open(F_i)*** – search the directory structure on disk for entry ***F_i***, and move the content of entry to memory
- ***Close (F_i)*** – move the content of entry ***F_i*** in memory to directory structure on disk

Open Files

- **Open-file table**:

  Several pieces of data are needed to track each open file:

  - **File pointer**: pointer to last read/write location, per process that has the file open
  - **File-open count**: counter of number of times a file is open – to allow remove the file's entry from open-file table when last processes closes it
  - **Disk location of the file**: cache of data access information
  - **Access rights**: per-process access mode information

Open File Locking

- Similar to reader-writer locks

Provided by some operating systems and file systems

  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock

File Locking Example – Java API

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
        public static final boolean EXCLUSIVE = false;
        public static final boolean SHARED = true;
        public static void main(String arsg[]) throws IOException {
            FileLock sharedLock = null;
            FileLock exclusiveLock = null;
            try {
                    RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
                    // get the channel for the file
                    FileChannel ch = raf.getChannel();
                    // this locks the first half of the file - exclusive
                    exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
                    /** Now modify the data . . . */
                    // release the lock
                    exclusiveLock.release();
```

File Locking Example – Java API (Cont.)

```
            // this locks the second half of the file - shared
            sharedLock = ch.lock(raf.length()/2+1, raf.length(), SHARED);
            /** Now read the data . . . */
            // release the lock
            sharedLock.release();
    } catch (java.io.IOException ioe) {
            System.err.println(ioe);
    }finally {
            if (exclusiveLock != null)
            exclusiveLock.release();
            if (sharedLock != null)
            sharedLock.release();
    }
  }
}
```

File Types – Name, Extension

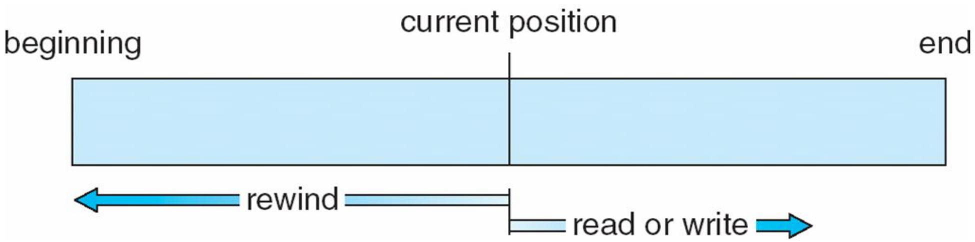
file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document

Access Methods

- **Sequential Access**
  - Read_next() - reads the next portion of the file and advances file pointer
  - Write_next() – appends to the end of the file and advances to the file end
  - reset() – to the beginning of the file



- **Direct Access** – for file is made up of fixed length logical records
  - read(n) – read block n
  - write(n) – write block n
  - an alternative approach is to retain read_next() and write_next()
    - position_file(n); read_next()
    - position_file(n); write_next()

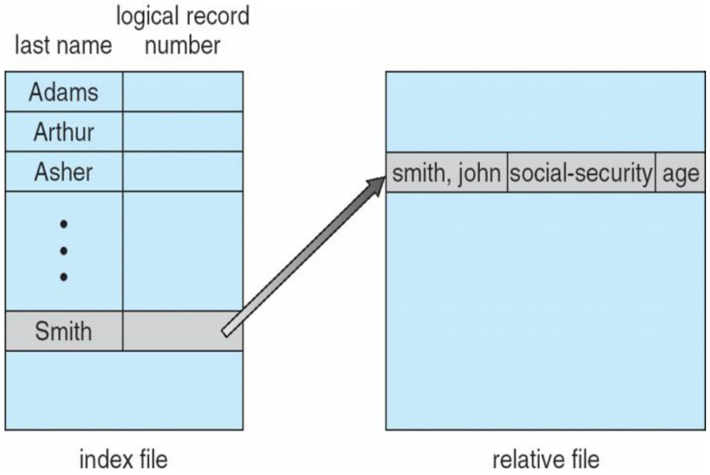
sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Simulation of Sequential Access on Direct-access File

Other Access Methods

Other access methods can be built on top of a direct-access method.

- Generally involve creation of an index for the file
- Keep index in memory for fast determination of location of data to be operated on (data are sorted)

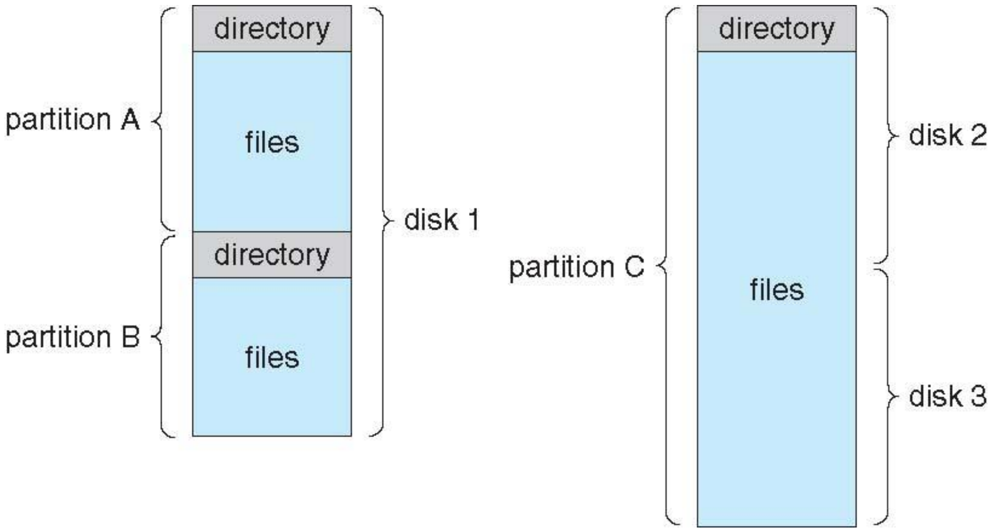


- If index file is too large, create an index for the index file
  - For example, IBM indexed sequential-access method (ISAM)
    - The file is kept sorted on a defined key
      - Use a master index that points to disk blocks of a secondary index
      - The secondary index blocks point to the actual file blocks.
    - All done by the OS

Disk Structure

A computer stores many files

- Disk can be subdivided into **partitions,** also known as minidisks, slices
  - Each can hold a file system; So, multiple file system types on the same device.
  - Leaving part of the device for other uses
    - Swap
    - Unformatted (raw) disk space
  - Disks or partitions can be **RAID** protected against failure



- Entity containing a file system known as a **volume**
  - Each volume can be thought as a virtual disk:
    - a subset of a device,
    - a whole device, or
    - multiple devices linked together into a RAID set
  - Each volume that contains a file system must also contain information about the files in the system. This information is stored in entries in a <u>device directory</u> or <u>volume table of contents</u>

Types of File Systems

As introduced above, a general-purpose computer system have multiple storage devices, those devices can be sliced up into volumes holding file systems.

- A computer system may have zero or more file systems which may be of varying types.
  - We talk of general-purpose file systems (there are many special purpose file system),
  - For example, Solaris has following file systems:
    - tmpfs – memory-based volatile FS for fast, temporary I/O
    - objfs – interface into kernel memory to get kernel symbols for debugging
    - ctfs – contract file system for managing daemons
    - lofs – loopback file system allows one FS to be accessed in place of another
    - procfs – kernel interface to process structures
    - ufs, zfs – general purpose file systems

Directory

Each volume that contains a file system must also contain information about the files in the system. This information is stored in entries in a device directory or volume table of contents

- Directory can be viewed as a table translates file names into their directory entries.
- The operations to be performed on a directory are
  - Search for a file – find the entry for a particular file
  - Create a file – new an entry in directory
  - Delete a file – remove entry in directory
  - List a directory – list the directory entry for each file
  - Rename a file
  - Traverse the file system

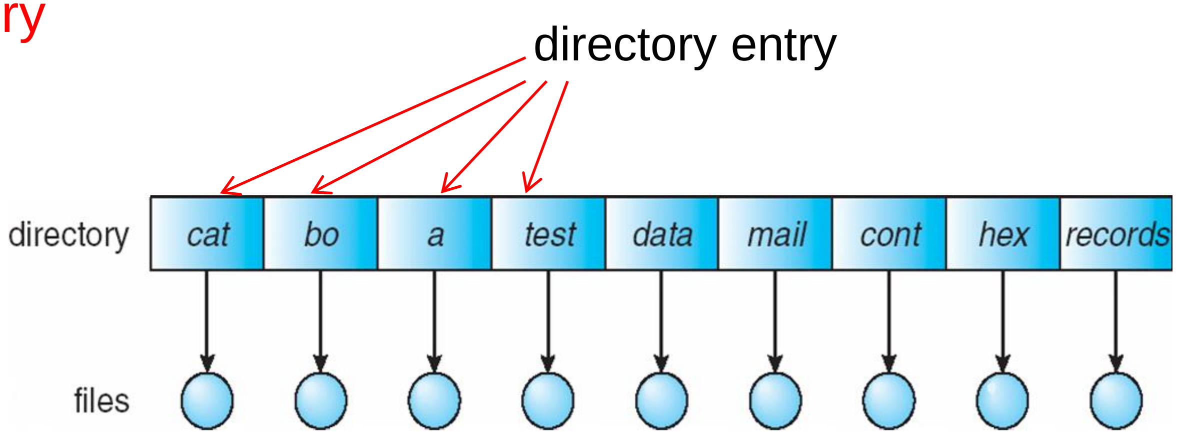
Directory Organization

The directory is organized logically to obtain

- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

Single-Level Directory

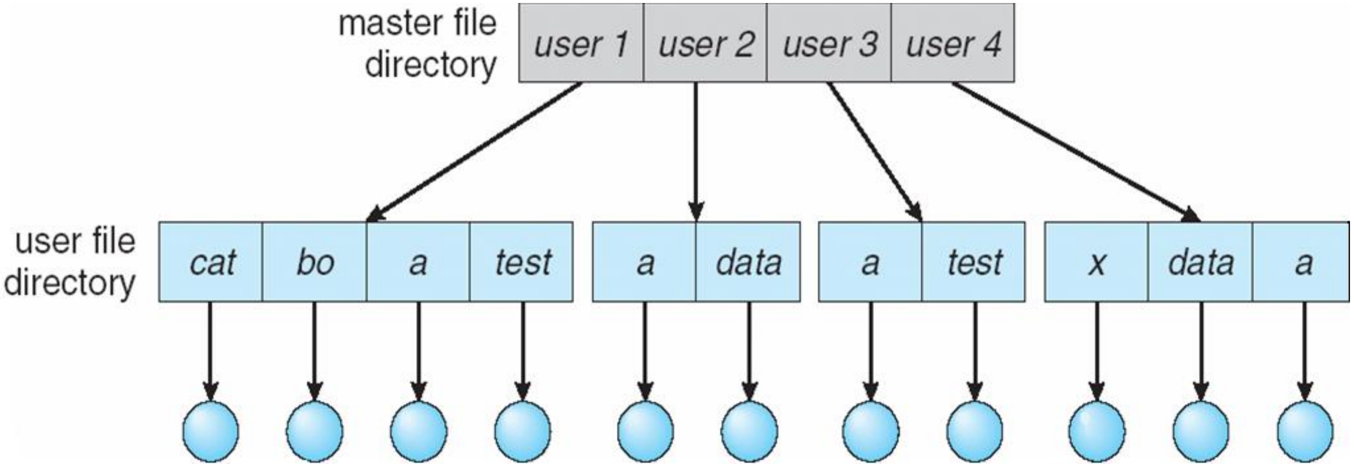
- A single directory for all users – all files are contained in the same directory



  - Naming problem – files in the same directory must have unique name, if the number of files is very large…
  - Grouping problem – hard to remember a lot of unique names

Two-Level Directory

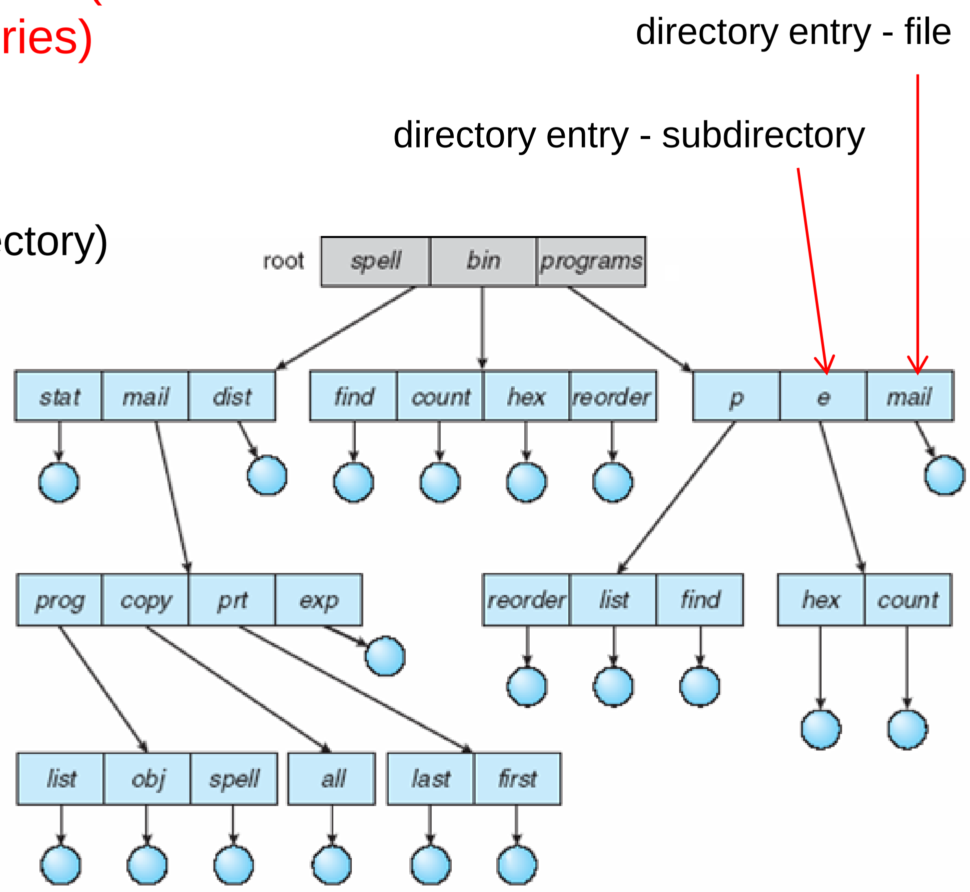
- Separate directory for each user



  - Path name – defined by a user name and a file name, so can have the same file name for different user
  - Efficient searching
  - No grouping capability

Tree-Structured Directories

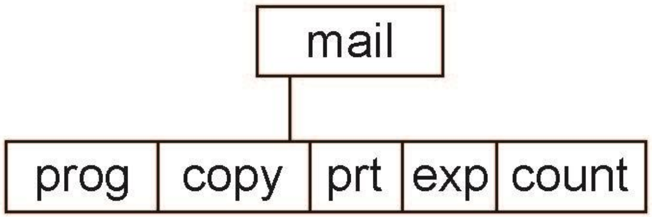
- Most common directory structure (allow each user to define own subdirectories)
  - Efficient searching
  - Grouping Capability
  - Current directory (working directory)



Tree-Structured Directories (Cont)

- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file **`rm <file-name>`**
- Creating a new subdirectory is done in current directory
  - Example: if in current directory /mail

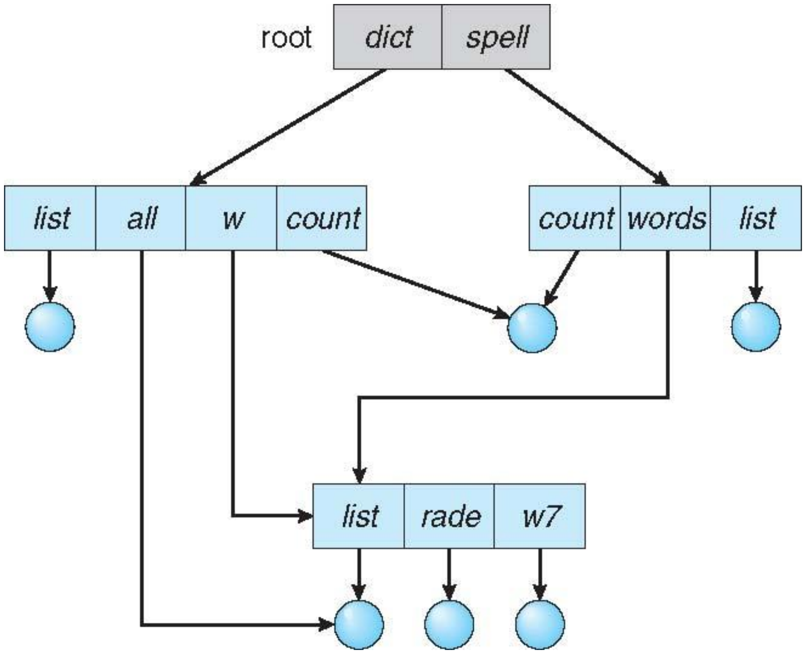
    `mkdir count`



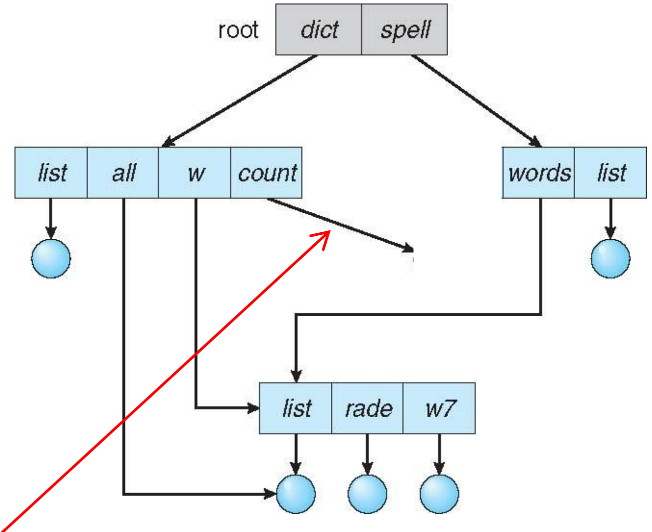
Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”

Acyclic-Graph Directories

- Have shared subdirectories and files
  - a graph with no cycles
  - the shared subdirectory and file has one copy



Acyclic-Graph Directories (Cont.)

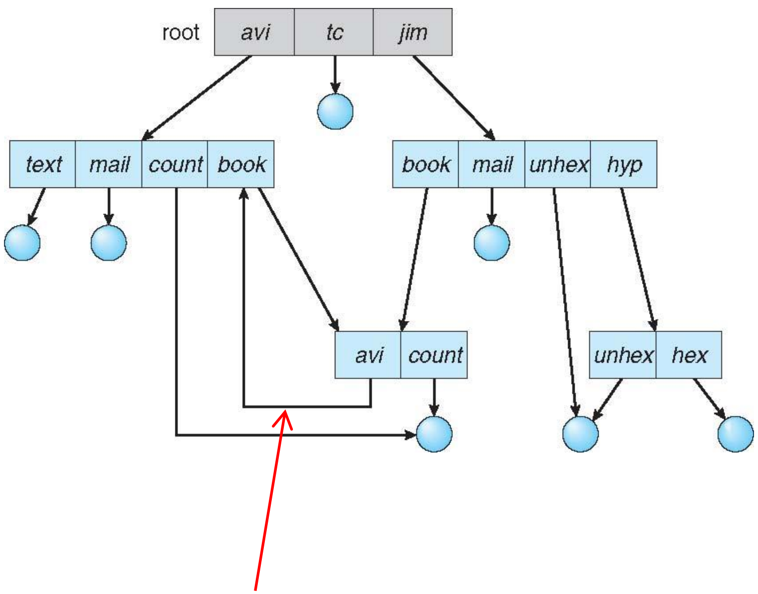


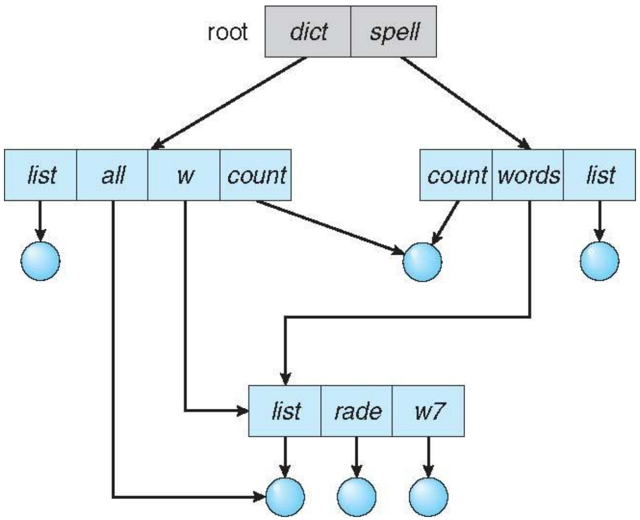
- Two different names (aliasing)
- New directory entry type
  - Symbolic Link (soft link) – a pointer to an existing file; follow pointer to locate the file to resolve the symbolic link.
  - Duplicate file entries (hard link) - all information about the shared file/directory stored in both sharing directories.
- Problem involves deletion

  When can the space allocated to a shared file be deallocated and reused.
  - Dangling pointers (i.e., the original file has been deleted)
    - Remain Symbolic link – Unix, Windows
    - Remove all duplicate file entries – Unix, Windows

General Graph Directory

- A serious problem of acyclic-graph structure is ensuring that there is no cycles
- If cycles are allowed to exist, resulting in a simple graph structure.





- How to avoid infinite loop?
  - Limit the number of directories that will be accessed during a search.

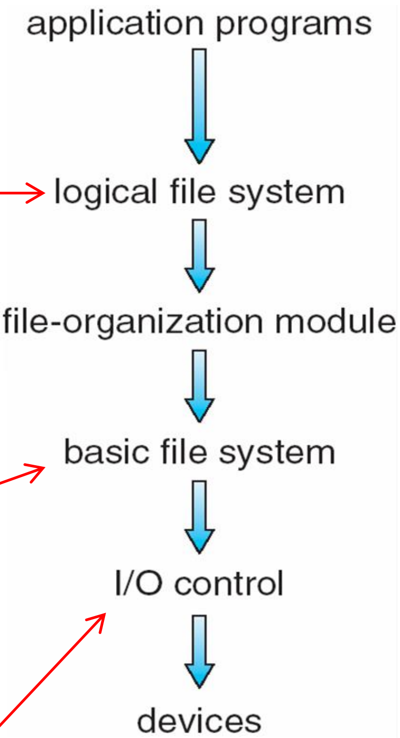
File-system implementation

Concerned with issues about file storage and access on the most common secondary-storage medium, the disk

File-system implementation

The file system is generally composed of many different levels.

Layered File System



Manage the directory structure
Translates file name into file number, file handle, location by maintaining file control blocks (e.g., inode)

Knows about files and their logical blocks as well as physical blocks

- Translates logical block # to physical block #
- Manages free space, disk allocation

Given command like “retrieve block 123” translates to device driver

Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller

File-System Implementation

Several on-disk and in-memory structures are used to implement a file system

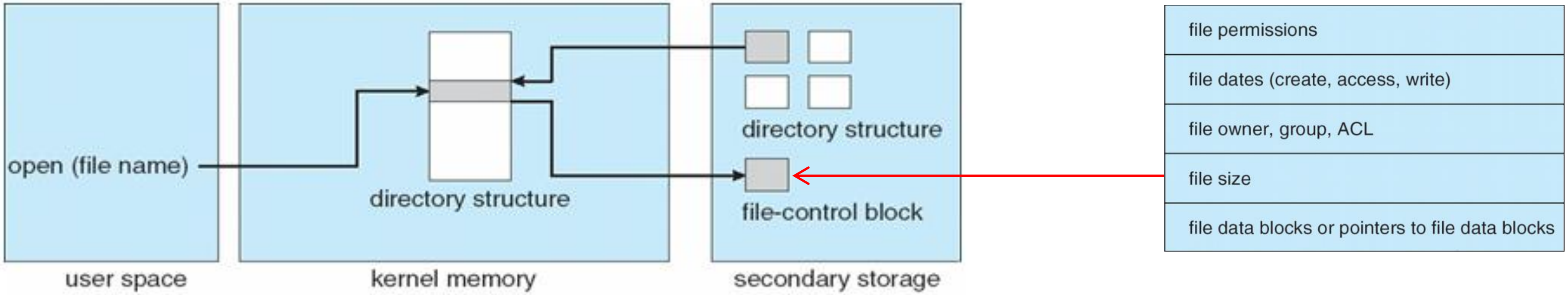
- On-disk - a file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files
  - **Boot control block**
    - contains info needed by system to boot OS from that volume
    - Needed if volume contains OS, usually first block of volume
    - Empty if the disk does not contain OS
  - Per-volume **Volume control block** **(**known as **superblock, master file table)**
    - contains volume details such total # of blocks, # of free blocks, block size, free block pointers or array
  - Per-file **File Control Block** **(FCB)** contains many details about the file (called inode in Linux)

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

File-System Implementation

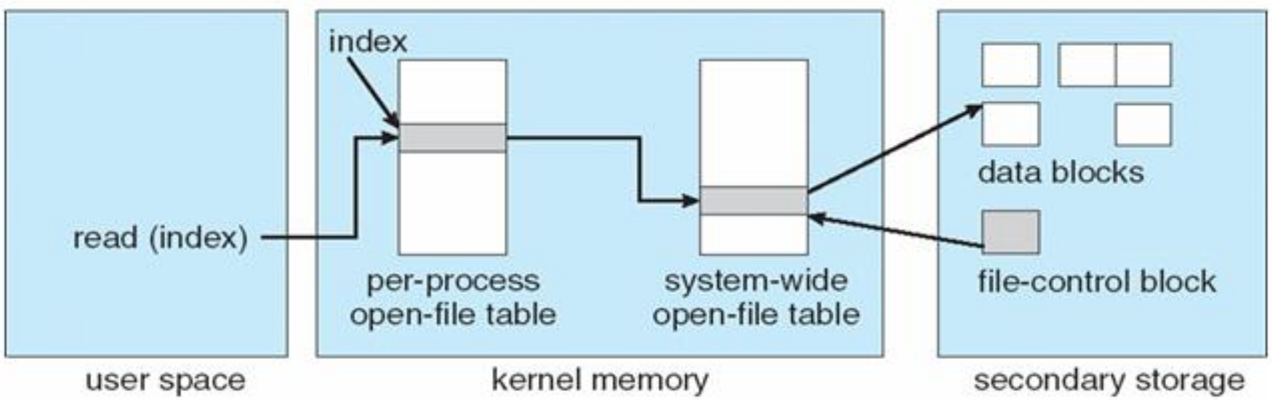
- In-Memory File System Structures

The open() system call first searches the system-wide open-file table to see if the file is already in use by another process.



(a)

- If the file is not already open, the directory structure is searched for the given file name. Once the file is found, the FCB is copied into a system-wide open-file table in memory. This table not only stores the FCB but also tracks the number of processes that have the file open.



(b)

- Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields (including a pointer to the current location in the file for the next read() or write() operation) and the access mode in which the file is open.

File-System Implementation

- In-Memory File System Structures
  - If the file is already open, a per-process open-file table entry is created pointing to the existing system-wide open-file table. This algorithm can save substantial overhead.

Allocation Methods - Contiguous

An allocation method refers to how disk blocks are allocated for files (note different from process allocation in memory)

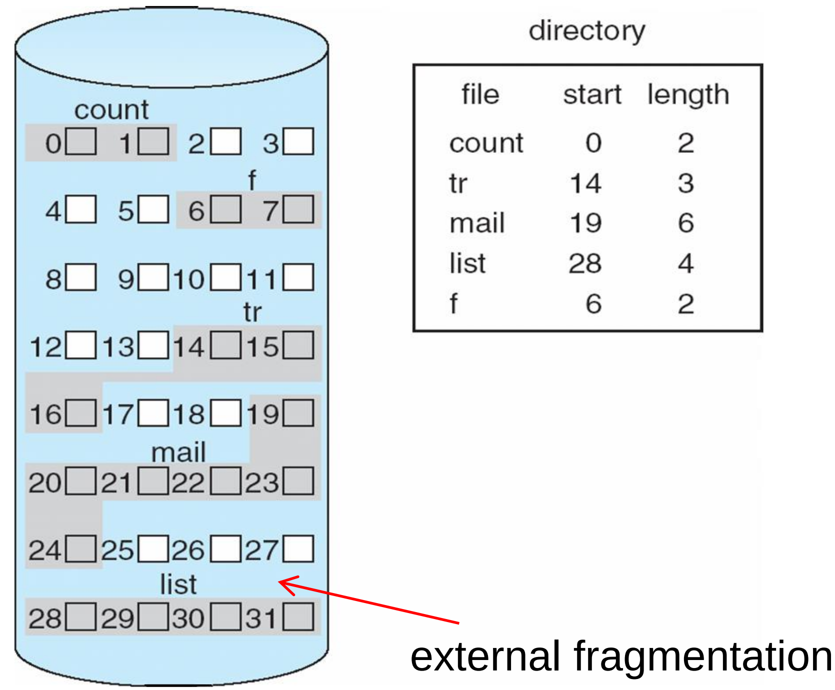
- **Contiguous allocation** – each file occupies set of contiguous blocks (Note different from contiguous allocation of memory to process. Here is contiguous blocks and there, the memory is not contiguous)
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include finding space for file, knowing file size, external fragmentation, need for **compaction off-line** (**downtime**) or **on-line**

Contiguous Allocation (Cont.)

- Mapping from logical to physical

LA/512 → Q, R

Block to be accessed = Q + starting address
Displacement into block = R



directory

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

Q: quotition
R: remainder
Divisor 512 is block size
Dividend LA is address in the file to access

Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - Free space management system called when new block needed
  - Locating a block can take many I/Os and disk seeks
- FAT (File Allocation Table) variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

Linked Allocation (Cont.)

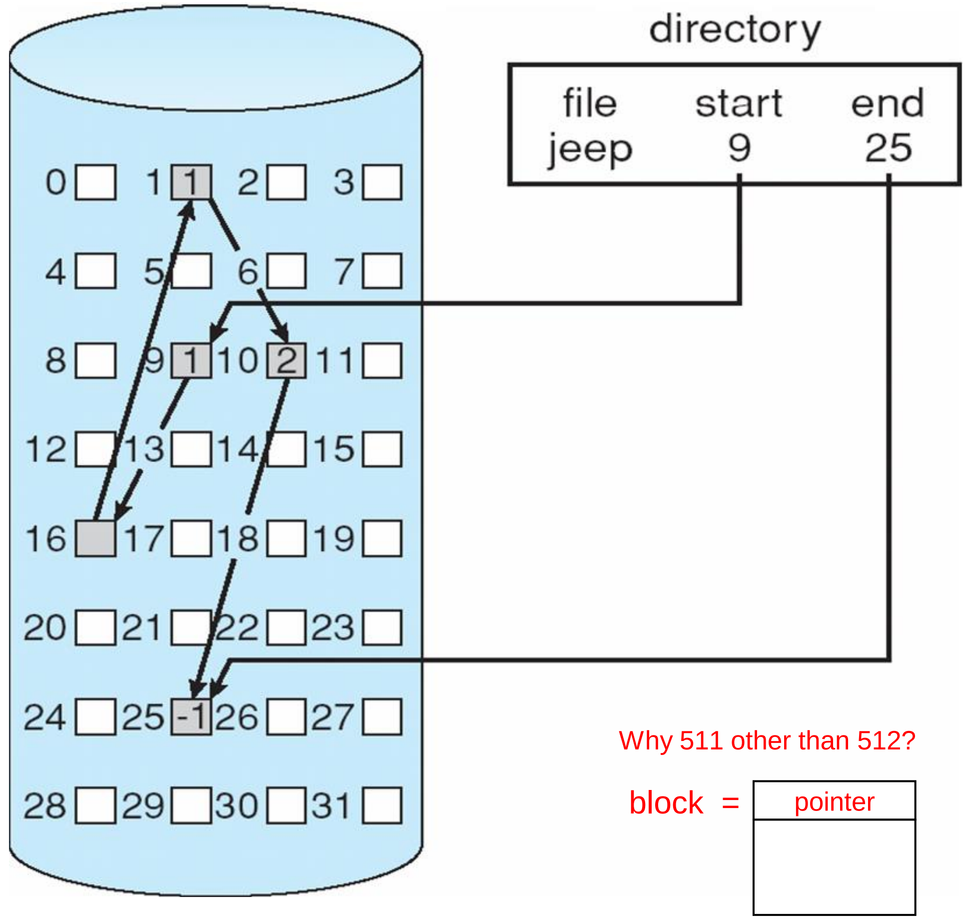
Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk

- Mapping from logic to physical

LA/511 → Q, R

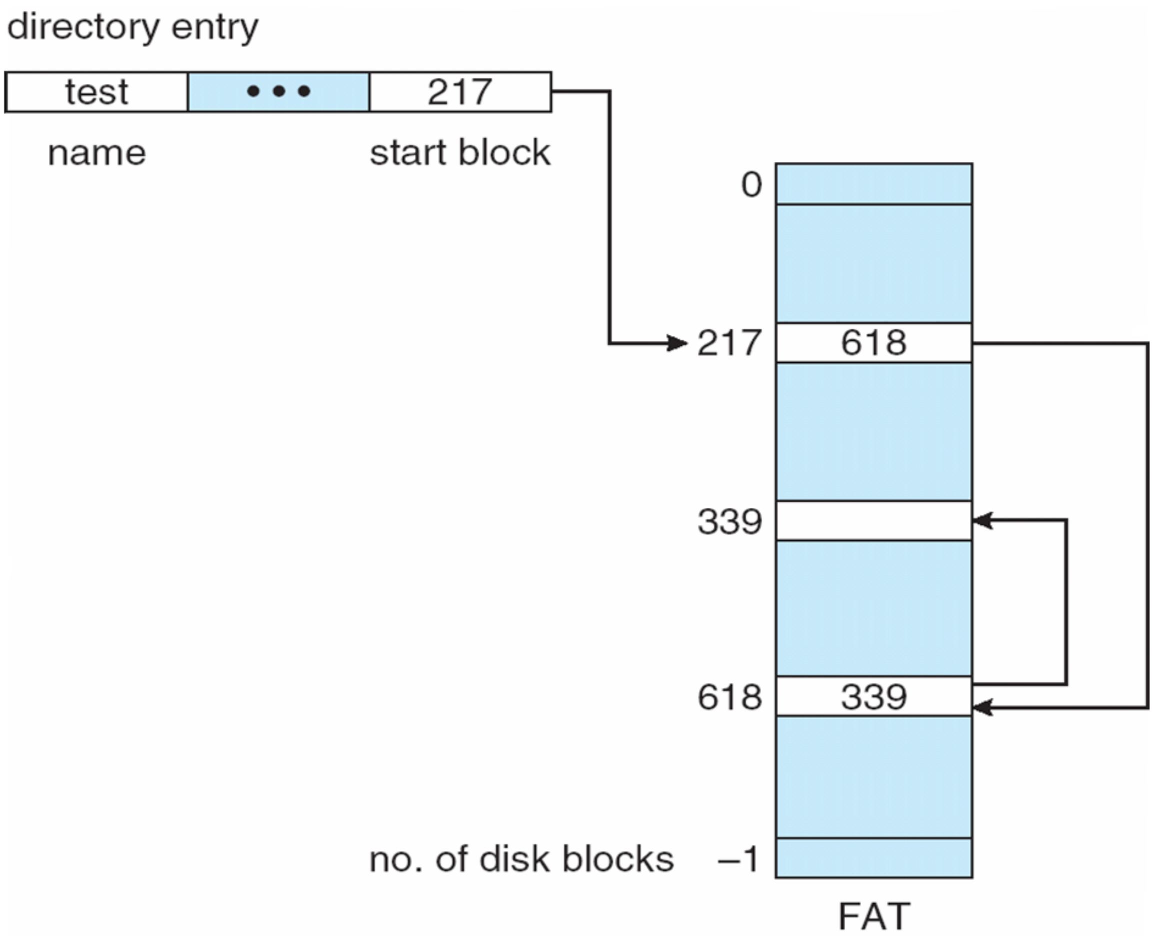
Block to be accessed is the Qth block in the linked chain of blocks representing the file.

Displacement into block = R + 1



Q: quotition
R: remainder
Divisor 511 is block size
Dividend LA is address in the file to access

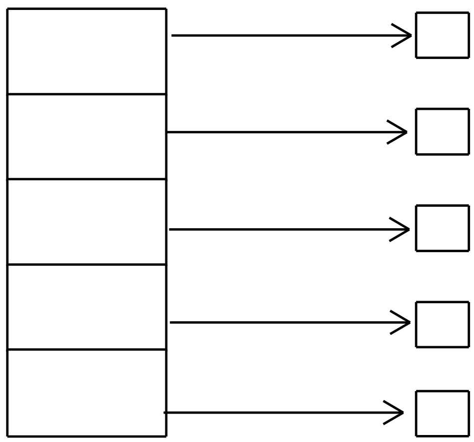
Linked Allocation - File-Allocation Table (Cont.)



Allocation Methods - Indexed

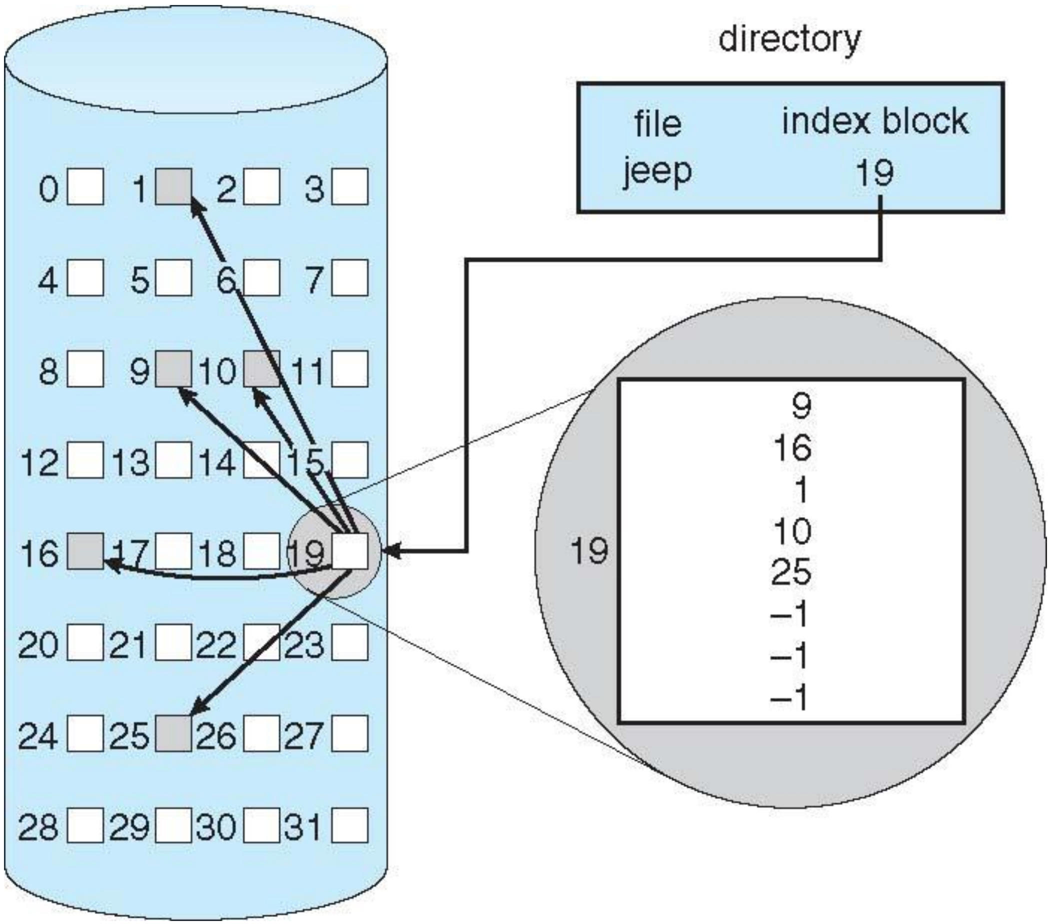
- Each file has its own **index block**(s) of pointers to its data blocks

- Logical view



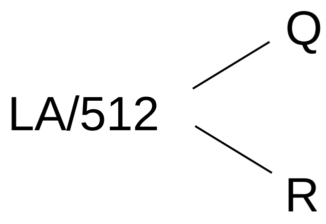
index table

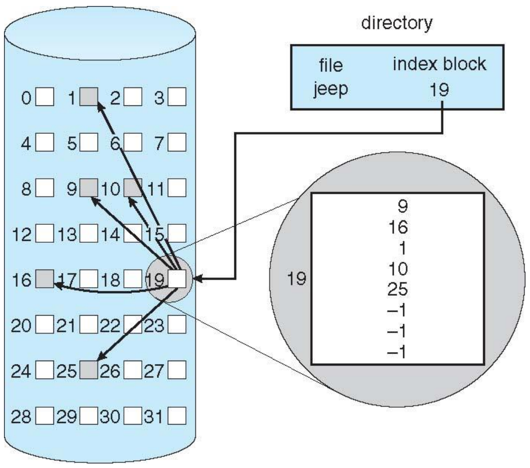
Example of Indexed Allocation (Cont.)



Indexed Allocation (Cont.)

- Need index table, Random access
- Dynamic access without external fragmentation, but have overhead of index block
- An index block is normally 1 block



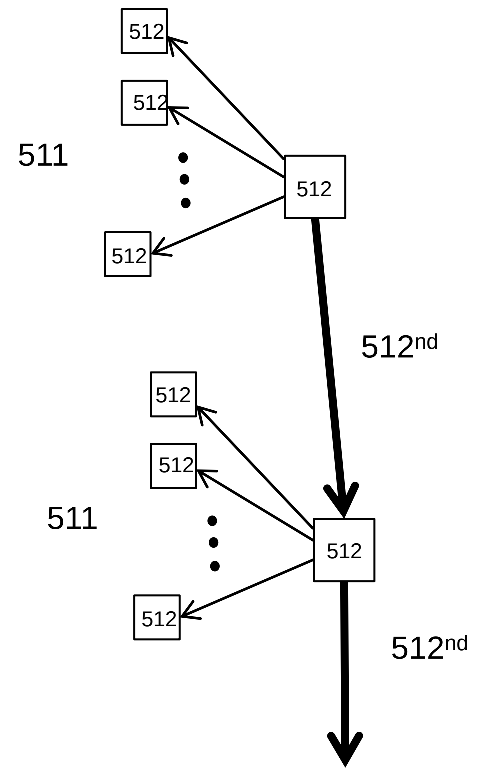


Q = displacement into index table
R = displacement into block

Indexed Allocation – Mapping (Cont.)

- To allow large file, we can use
  - linked scheme – link together several index blocks

LA / (512 x 511) → Q_1, R_1

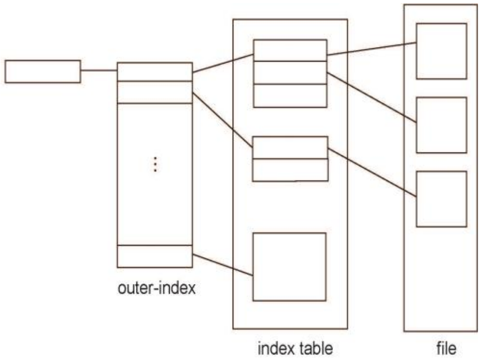


Q_1 = block of index table
R_1 is used as follows:

R_1 / 512 → Q_2, R_2

Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)



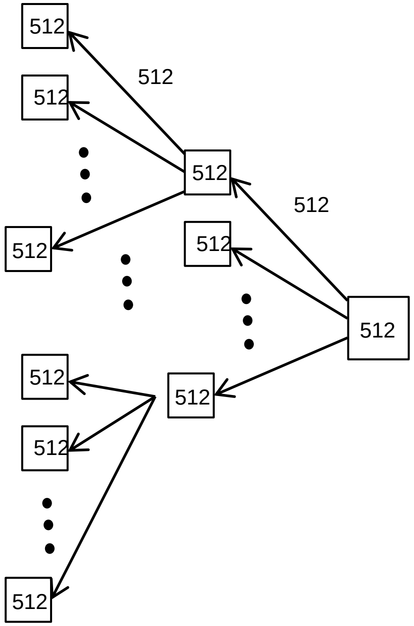
- To allow large file, we can also use
  - Two-level index –

LA / (512 x 512) → Q_1, R_1

Q_1 = displacement into outer-index
R_1 is used as follows:

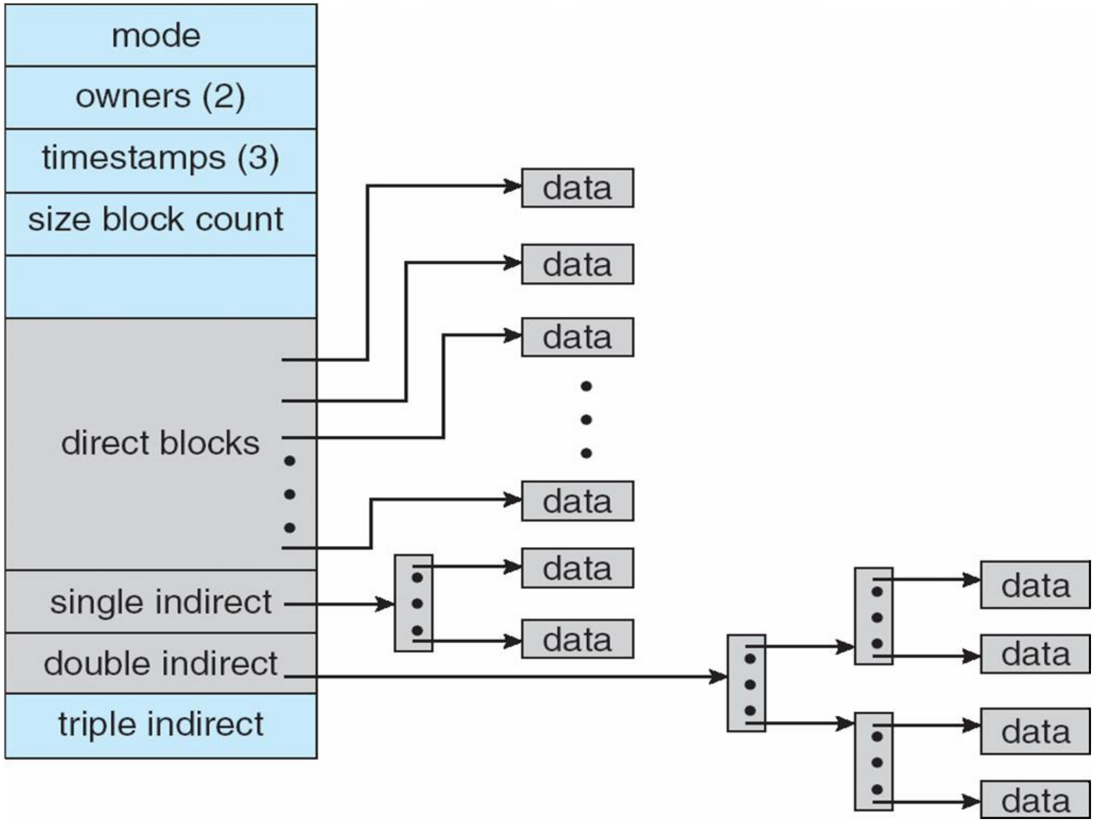
R_1 / 512 → Q_2, R_2

Q_2 = displacement into block of index table
R_2 displacement into block of file:



Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses



More index blocks than can be addressed with 32-bit file pointer

Next Week

Lecture 10 – I/O Systems

Tutorial 9