Chapter 12: 파일 시스템 인터페이스

File Concept

- **Contiguous logical address space**
- Types:
  - Data
    - numeric
    - character
    - binary
  - Program
- Contents (many types) is defined by file's creator
  - 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**
- ***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

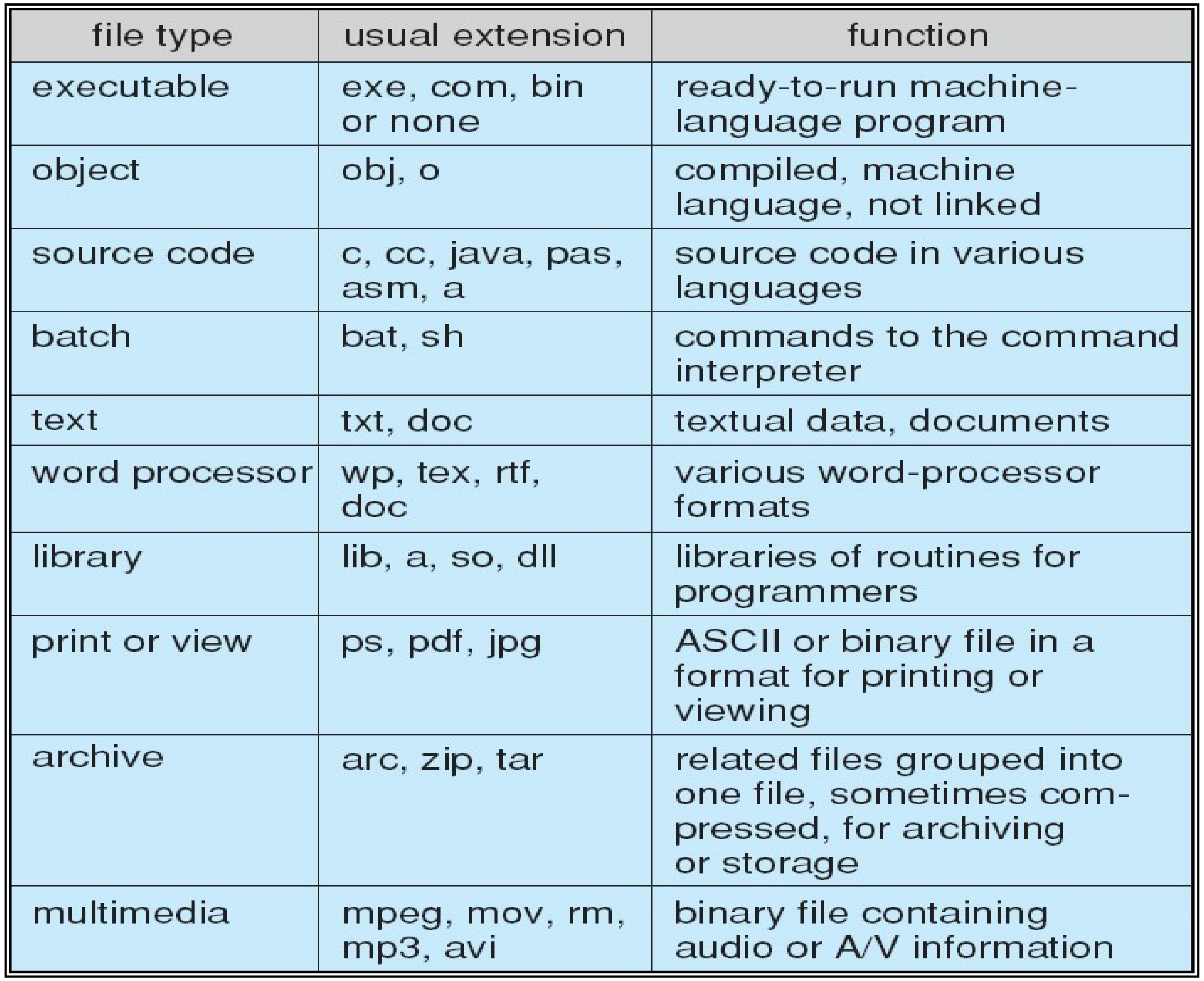
Open Files

- **Open-file table**: Keeps information (inode data) about all the open files
- **File pointer** : A pointer to last read/write location, per process that has the file open
- **File-open count**: A counter of the number of times a file is open – to allow removal of data from open-file table when last processes closes it
- **Disk location of the file**: Many file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
- **Access rights**: Per-process access mode information

Open File Locking

- Provided by some operating systems and file systems
  - Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock
- Mediates access to a file
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do

File Types – Name, Extension

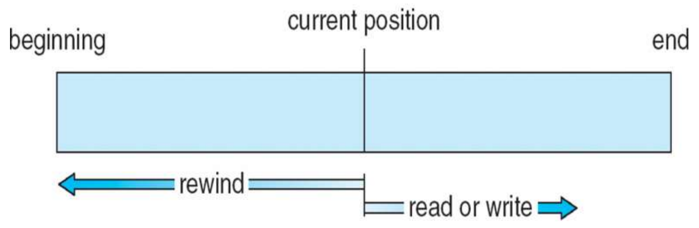


file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual 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

Access Methods

Sequential Access

- General structure



- Operations:
  - **read_next ()** – reads the next portion of the file and automatically advances a file pointer.
  - **write_next () –** append to the end of the file and advances to the end of the newly written material (the new end of file).
  - **reset** – back to the beginning of the file.

Access Methods

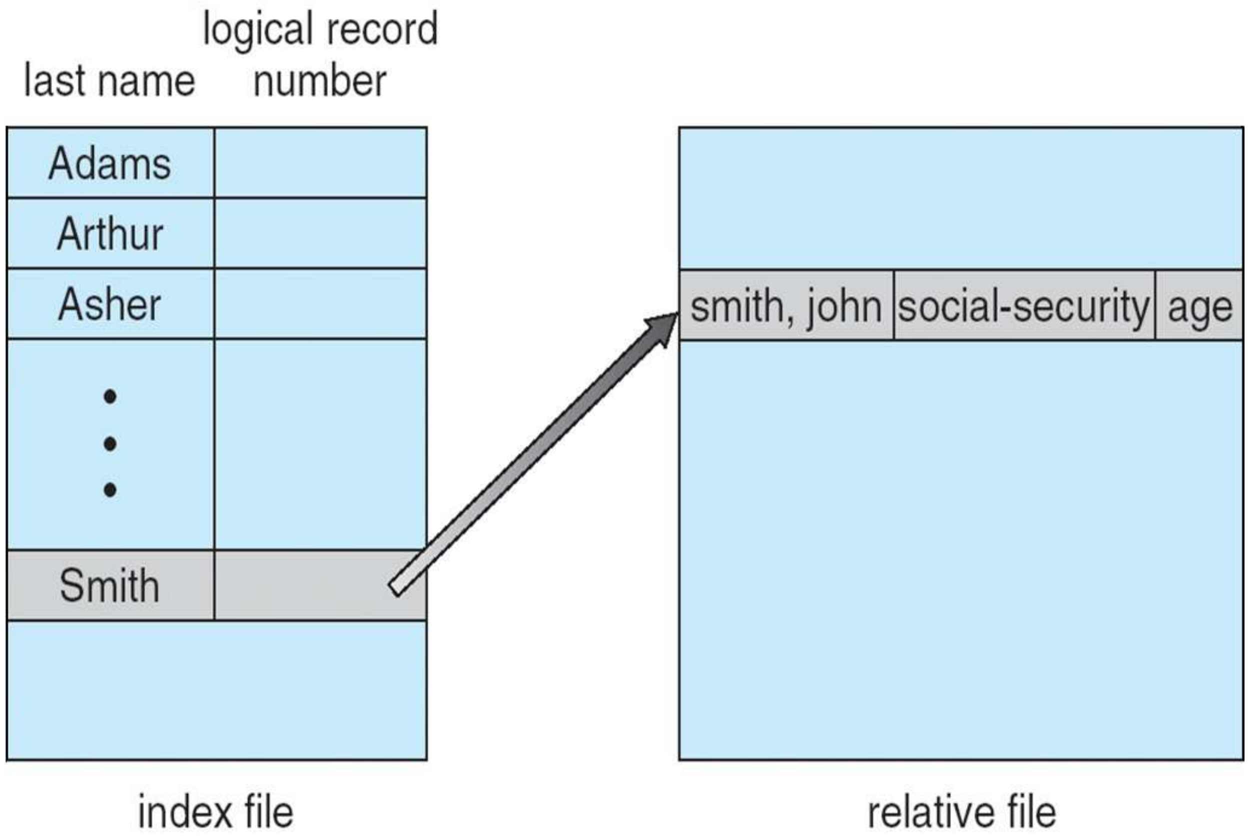
Direct Access

- File is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- File is viewed as a numbered sequence of blocks or records. For example, can read block 14, then read block 53, and then write block 7.
- Operations:
  - **read(n)** – reads relative block number n.
  - **write(n)** – writes relative block number n.
- Relative block numbers allow OS to decide where file should be placed

Other Access Methods

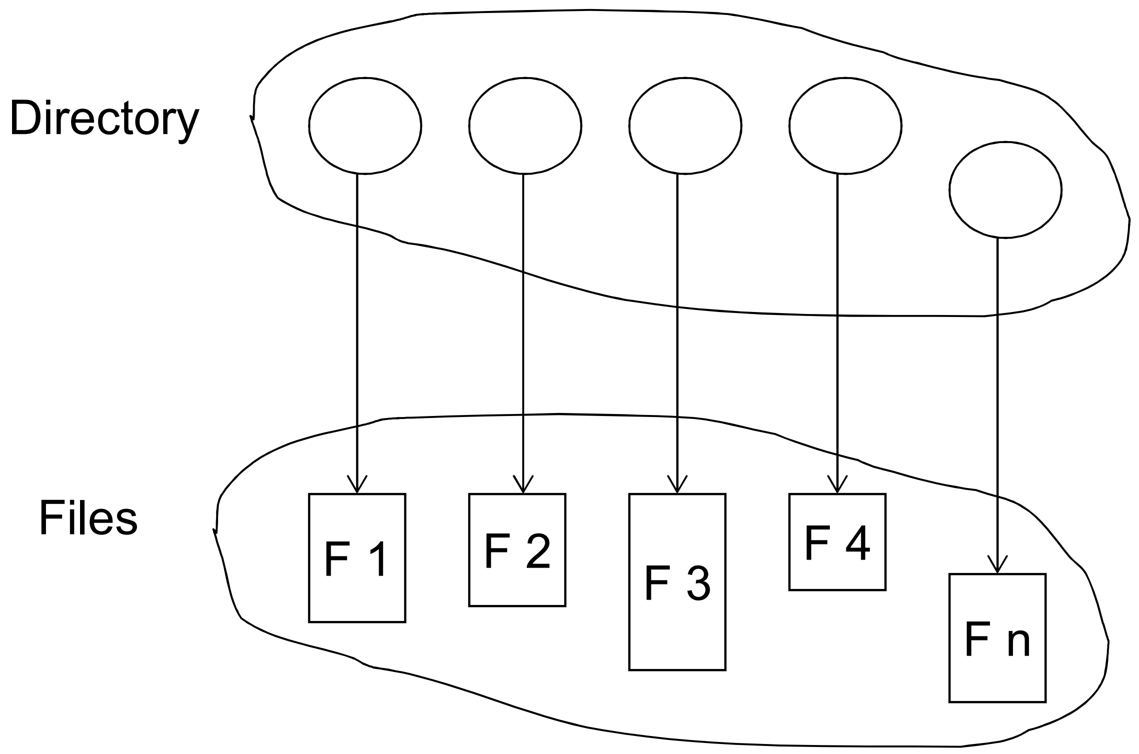
- Can be built on top of **base methods**
- General involve creation of **an index** for the file
- Keep index in memory for fast determination of location of data to be operated on (consider UPC code plus record of data about that item)
- If too large, index (in memory) of the index (on disk)
- IBM indexed sequential-access method (ISAM)
  - Small master index, points to disk blocks of secondary index
  - File kept sorted on a defined key
  - All done by the OS
- VMS operating system provides index and relative files as another example (see next slide)

Example of Index and Relative Files



Directory Structure

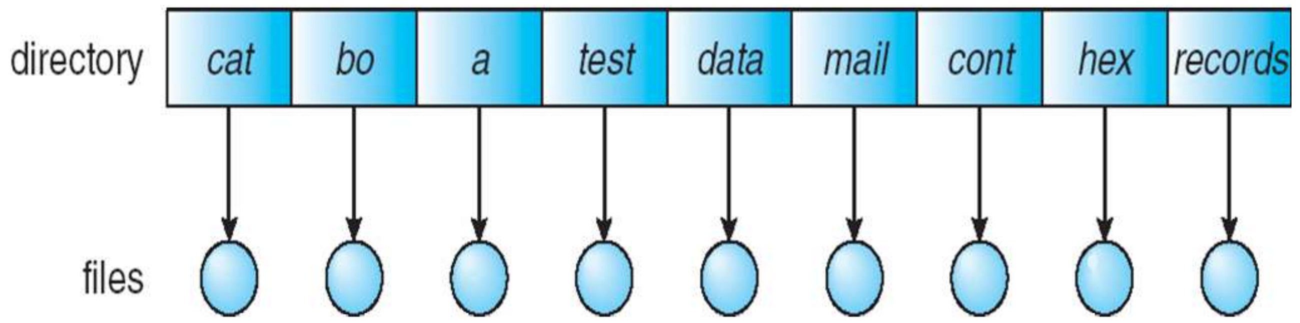
- A collection of nodes containing information about all files



Both the directory structure and the files reside on disk

Single-Level Directory

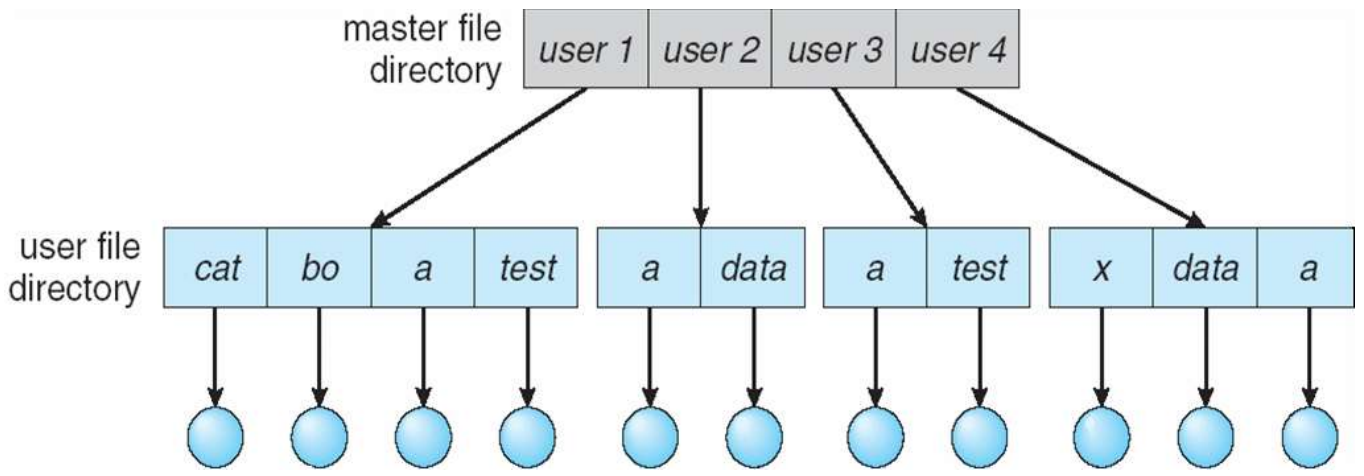
- A single directory for all users



- Naming problem
- Grouping problem

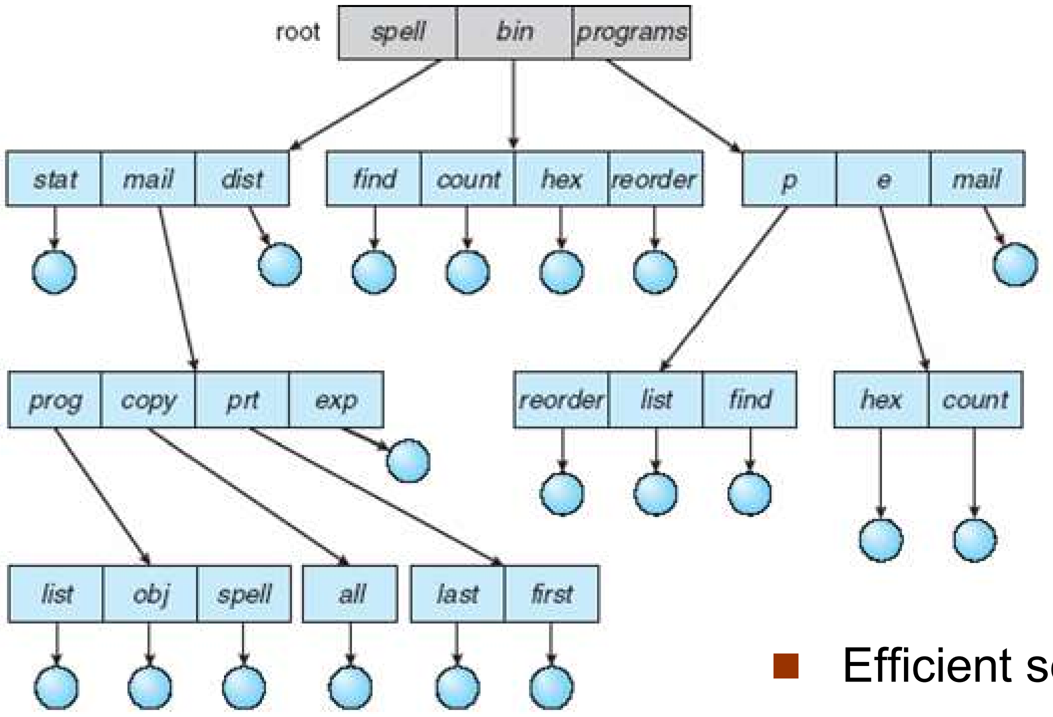
Two-Level Directory

- Separate directory for each user



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories



- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - `cd /spell/mail/prog`
  - `type list`

Tree-Structured Directories (Cont)

- **Absolute or relative path name**
- Creating a new file is done in current directory
- Delete a file

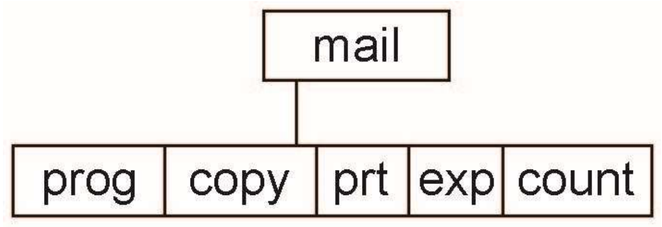
```
rm <file-name>
```

- Creating a new subdirectory is done in current directory

```
mkdir <dir-name>
```

Example: if in current directory `/mail`

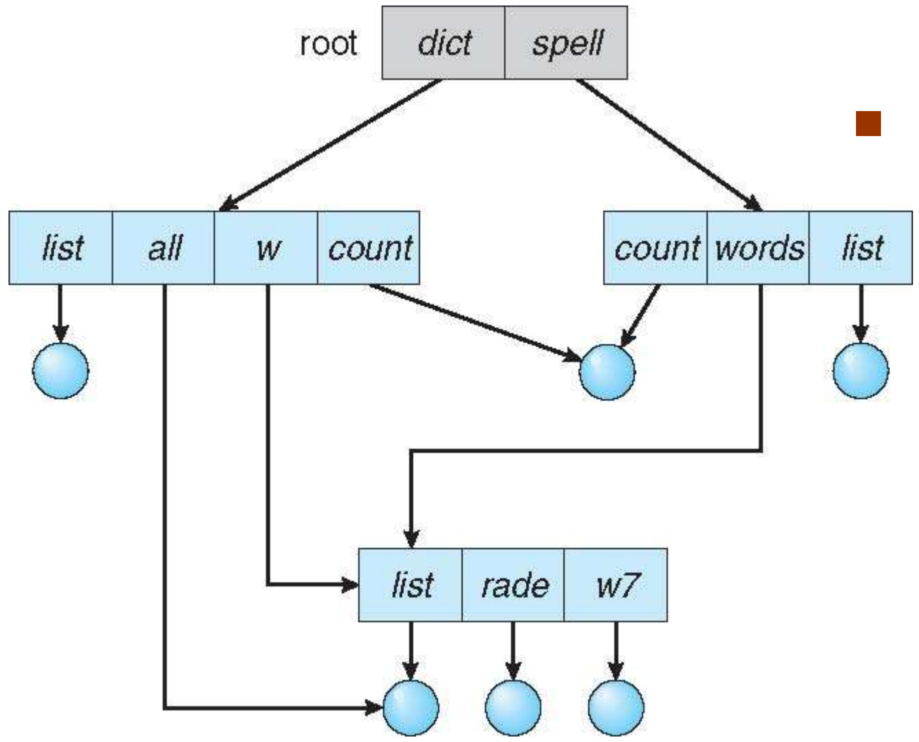
```
mkdir count
```



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

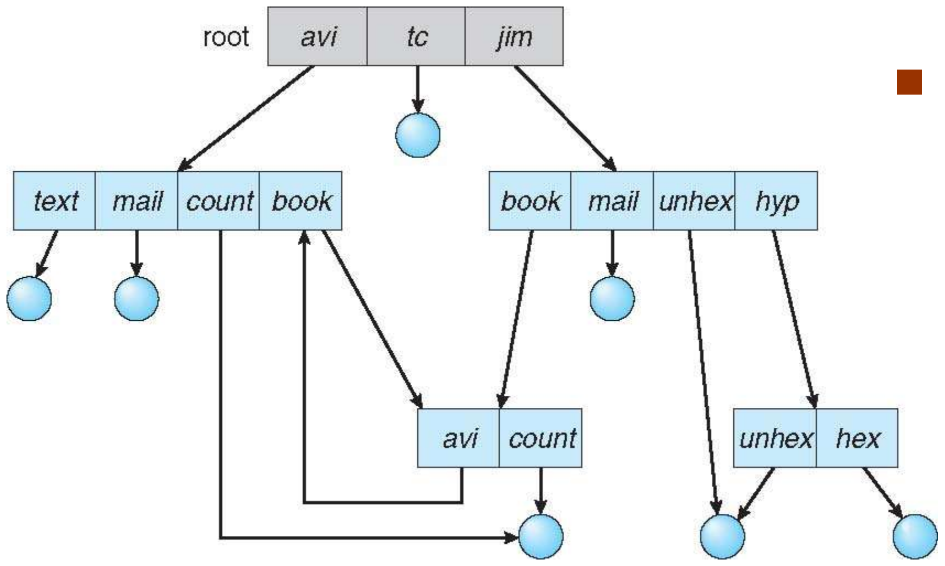
Acyclic-Graph Directories

- Have shared subdirectories and files



- New directory entry type
  - **Link** – another name (pointer) to an existing file
    - A link may be implemented as an **absolute** or a **relative** path name.
    - When a reference to a file is made, the directory is searched. If the directory entry is marked as a link, then the name of the real file is included in the link information.
  - **Resolve the link** – follow pointer to locate the file

General Graph Directory



- If ***dict /count*** is deleted ⇒ dangling pointer

  Solutions:
  - Backpointers -- so we can delete all pointers.
    - Variable size records a problem
  - Backpointers using a daisy-chain organization
  - Entry-hold-count solution

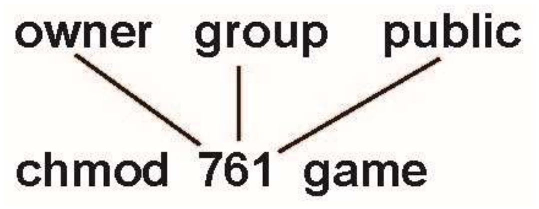
- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - **Garbage collection**
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

			RWX
a) **owner access**	7	⇒	1 1 1
			RWX
b) **group access**	6	⇒	1 1 0
			RWX
c) **public access**	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



```
chmod 761 game
```

Attach a group to a file

```
chgrp     G     game
```

A Sample UNIX Directory Listing

```
-rw-rw-r--    1 pbg   staff     31200  Sep 3 08:30    intro.ps
drwx------    5 pbg   staff       512  Jul 8 09.33    private/
drwxrwxr-x    2 pbg   staff       512  Jul 8 09:35    doc/
drwxrwx---    2 pbg   student     512  Aug 3 14:13    student-proj/
-rw-r--r--    1 pbg   staff      9423  Feb 24 2003    program.c
-rwxr-xr-x    1 pbg   staff     20471  Feb 24 2003    program
drwx--x--x    4 pbg   faculty     512  Jul 31 10:31   lib/
drwx------    3 pbg   staff      1024  Aug 29 06:52   mail/
drwxrwxrwx    3 pbg   staff       512  Jul 8 09:35    test/
```

Chapter 14: 파일 시스템 구현

File System Implementation

- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery

Disk Space Allocation Methods

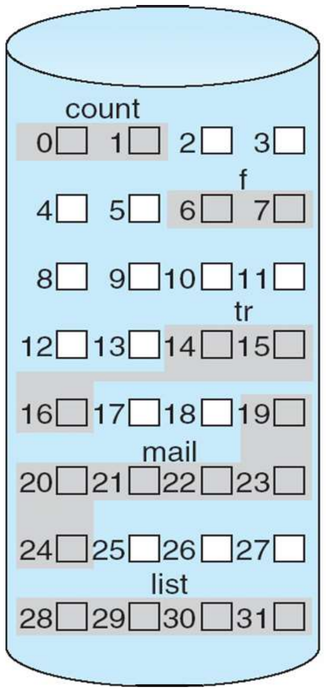
- A disk is a direct-access device and thus gives us flexibility in the implementation of files.
- The main issue is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.
- Three major methods of allocating disk space are in wide
  - Contiguous
  - Linked
  - Indexed

Contiguous Allocation

- Mapping from logical to physical

$$\begin{tikzcd} & Q \\ LA/512 \arrow[ur, no head] \arrow[dr, no head] & \\ & R \end{tikzcd}$$

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

Contiguous Allocation

Each file occupies a set of contiguous blocks

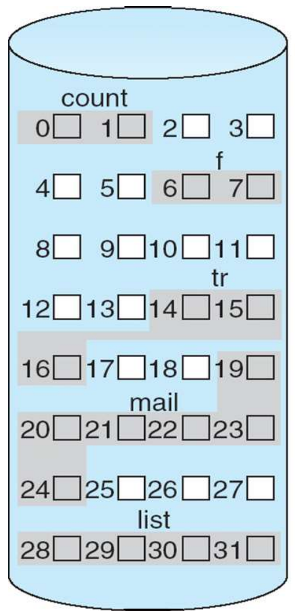
- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include:
  - Finding space for file,
  - Knowing the max file size,
  - External fragmentation,
    - Need for compaction
      - Off-line (downtime) or
      - On-line

Contiguous Allocation (Cont.)

- Mapping from logical to physical (assume block size if 512)

$$\begin{tikzcd} & Q \\ LA/512 \arrow[ur, dash] \arrow[dr, dash] & \\ & R \end{tikzcd}$$

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



directory

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

Extent-Based Systems

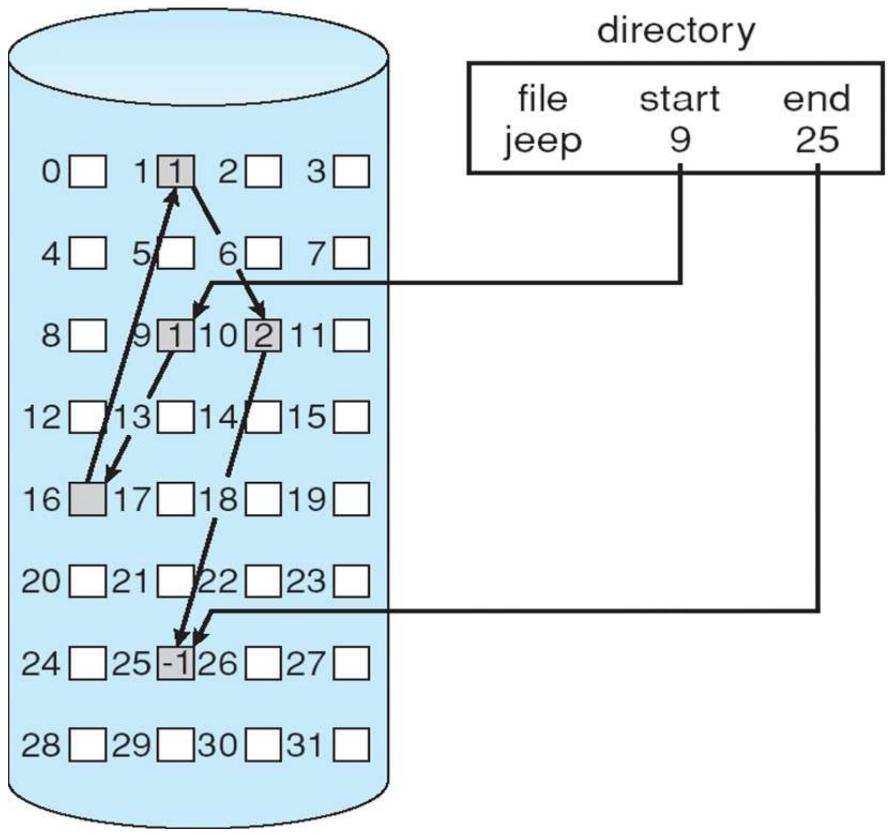
- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

Linked Allocation

- **Linked allocation** – each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - No compaction, external fragmentation
  - Free space management system called when new block needed
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks

Allocation Methods – Linked (Cont.)

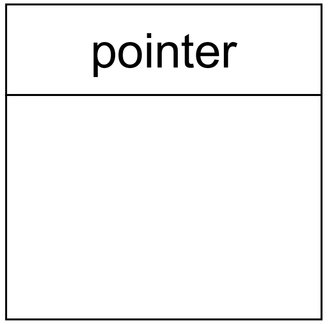
- FAT (File Allocation Table) variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple



Linked Allocation

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

block =



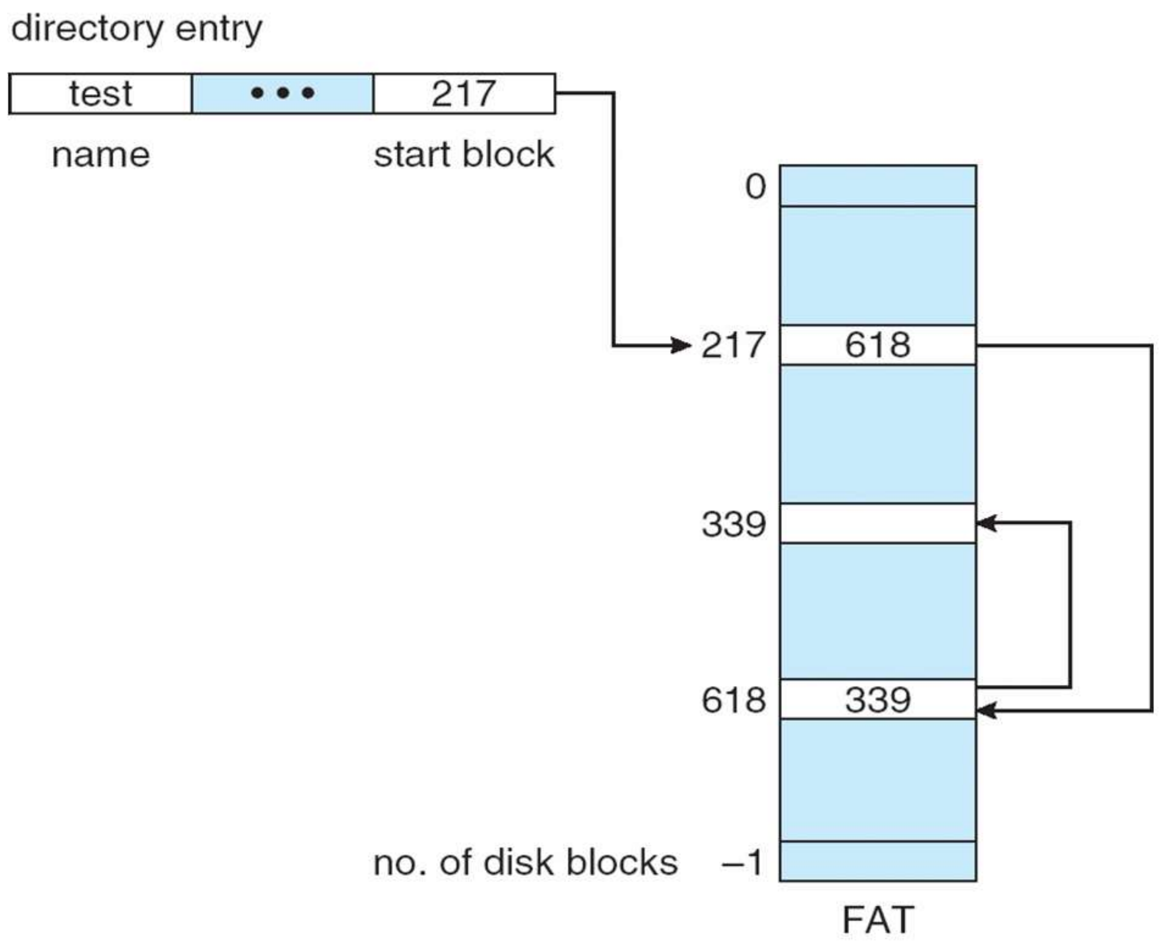
- Mapping

$$\begin{tikzcd}
 & Q \\
LA/511 \arrow[ur, dash] \arrow[dr, dash] & \\
 & R
\end{tikzcd}$$

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

Displacement into block = R + 1

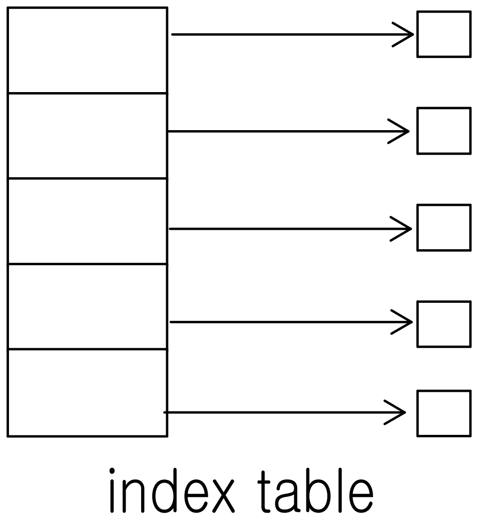
File Allocation Table(FAT)



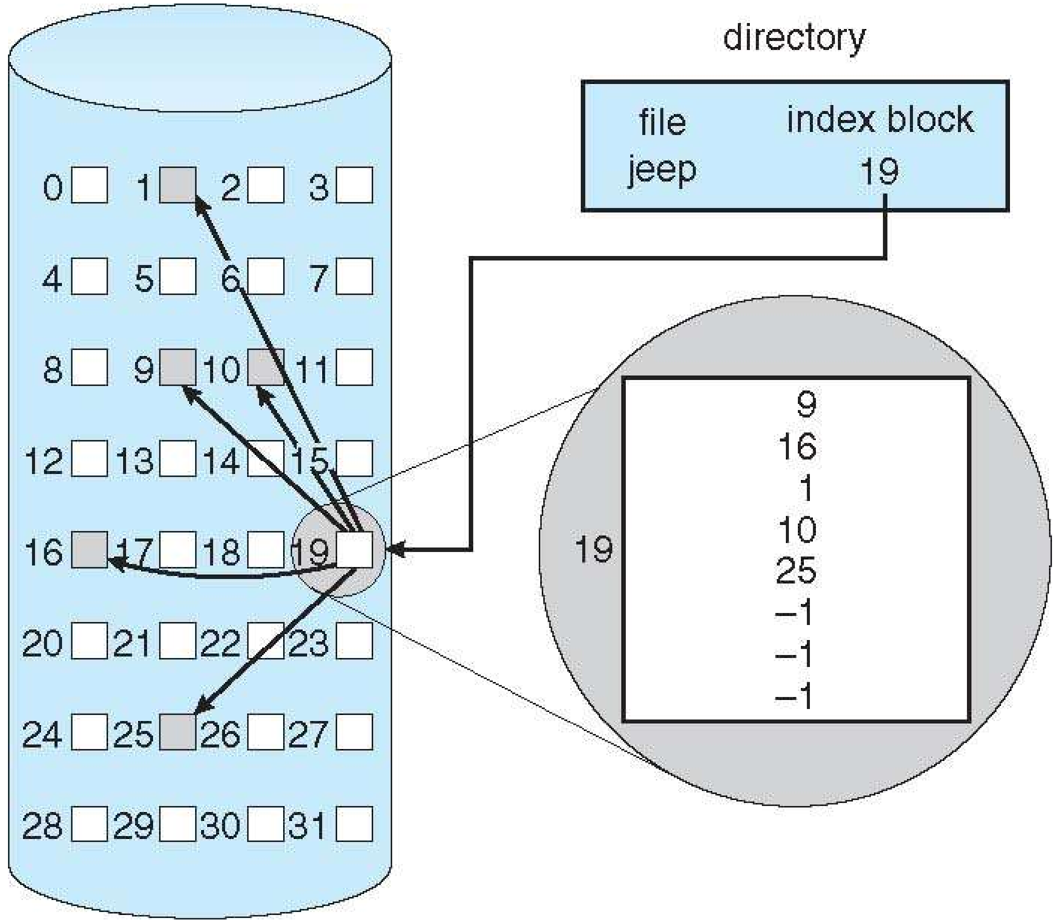
Indexed Allocation

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

- Logical view



Example of Indexed Allocation



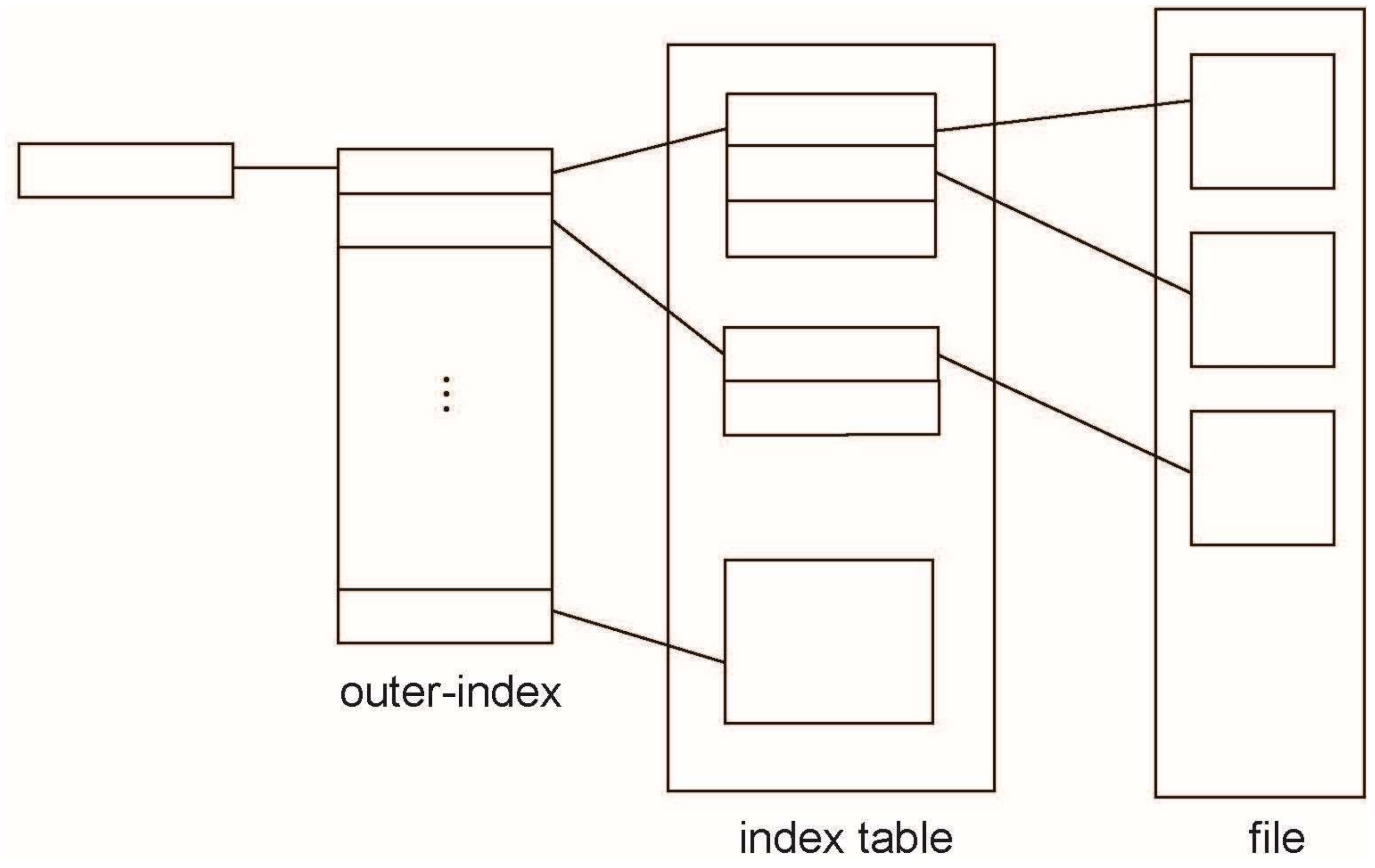
Indexed Allocation (Cont.)

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table

$$\begin{tikzcd}
 & Q \\
LA/512 \arrow[ur, no head] \arrow[dr, no head] & \\
 & R
\end{tikzcd}$$

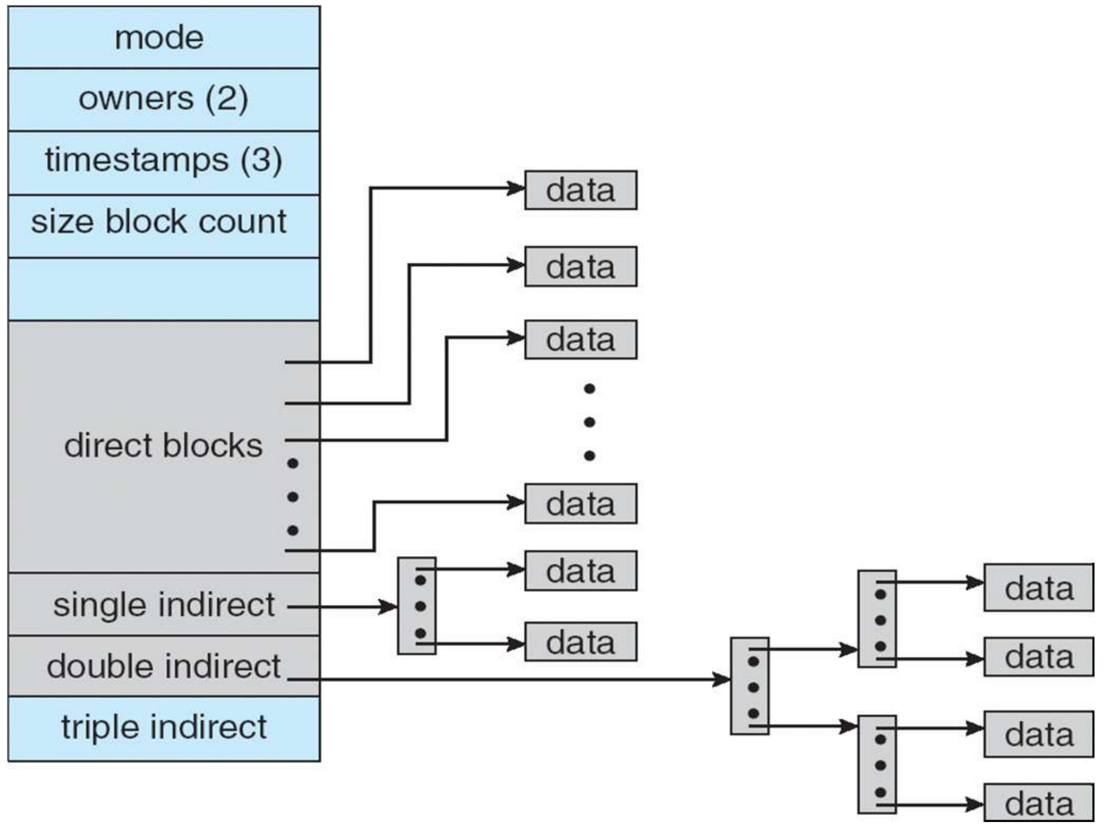
Q = displacement into index table
R = displacement into block

Indexed Allocation – Mapping (Cont.)



Combined Scheme: UNIX UFS

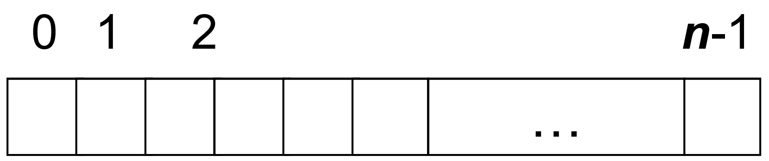
4K bytes per block, 32-bit addresses



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

Free-Space Management – Bit map

- File system maintains **free-space list** to track available blocks/clusters
  - (Using term “block” for simplicity)
- **Bit vector** or **bit map** (***n*** blocks)



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

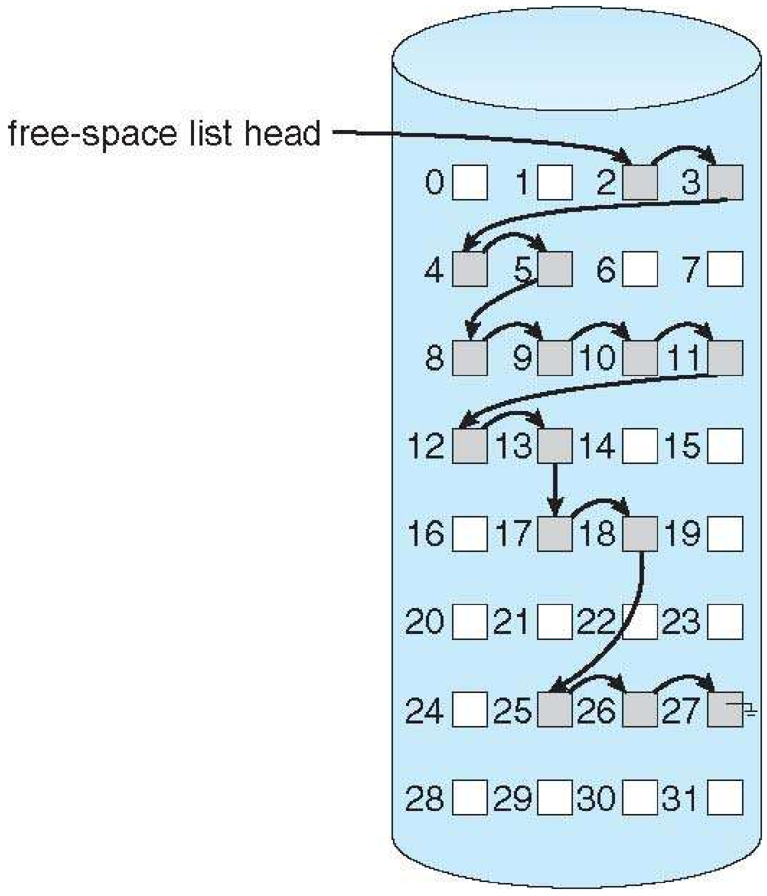
Block number calculation

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

CPUs have instructions to return offset within word of first “1” bit

Linked Free Space List on Disk

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
  - No need to traverse the entire list (if # free blocks recorded)



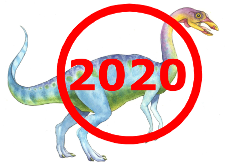
Chapter 15: 파일 시스템 내부구조(Advanced)

File System Internals

- File-system Structure
- Disk Structure
- File-System Mounting
- Partitions and Mounting
- Virtual File Systems
- Remote File Systems
- Consistency Semantics
- NSF
- Internal File Structure
- File Sharing

파일 시스템(File System, 파일체계) 이란?

-> 컴퓨터에서 파일이나 자료를 쉽게 발견 및 접근할 수 있도록, 보관 또는 조직하는 체제를 가리키는 말.



File-System Structure

- File structure
  - Logical storage unit, Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user **interface to storage**, **mapping logical to physical**
  - Provides **efficient and convenient access to disk** by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and **random access**
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block** – storage structure consisting of **information about a file**
- **Device driver** controls the physical device
- File system **organized into layers**
- 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. Linux has more than 40 types, with **extended file system** ext2 and ext3 leading + distributed file systems.
  - New ones - ZFS, GoogleFS, Oracle ASM, FUSE

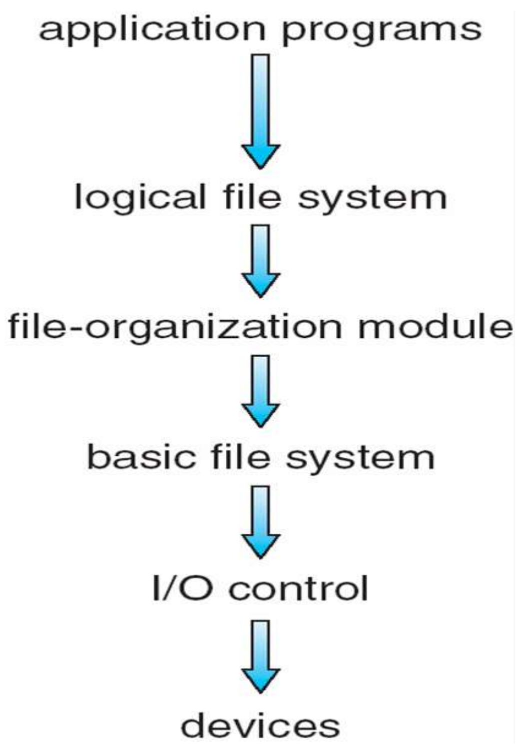
File System Layers

- **Device drivers** manage I/O devices at the I/O control layer
  - Given commands like "read drive1, cylinder 72, track 2, sector 10, into memory location 1060" outputs low-level hardware specific commands to hardware controller
- **Basic file system** given command like "retrieve block 123" translates to device driver
- **File organization module** understands files,

  logical address, and physical blocks
- **Logical file system** manages metadata information
  - Translates file name into file number,

    file handle, location by maintaining file control

    blocks (**inodes** in UNIX)
  - Directory management
  - Protection



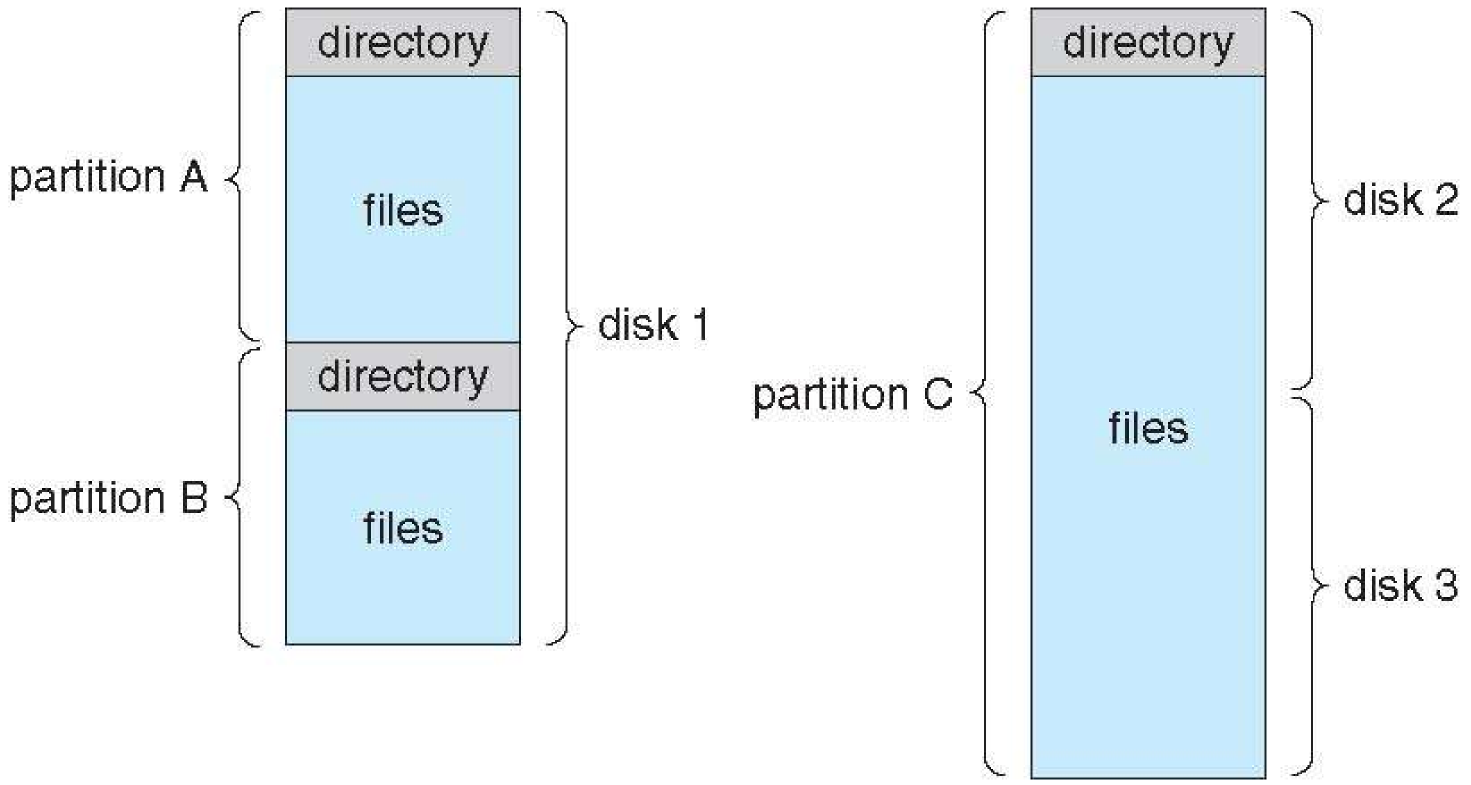
Disk Structure

- Disk can be subdivided into **partitions**
- Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
- An entity containing a file system is known as a **volume**
- Each volume containing file system also tracks that file system’s info in **device directory** or **volume table of contents**
- A device directory (or simply **directory**) records information; e.g., name, location, size, etc, about all the files that are kept on the volume.

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
  - Or a **boot management program** for multi-os booting
- **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 consistency checked
  - Is all metadata correct?

A Typical File-system Organization

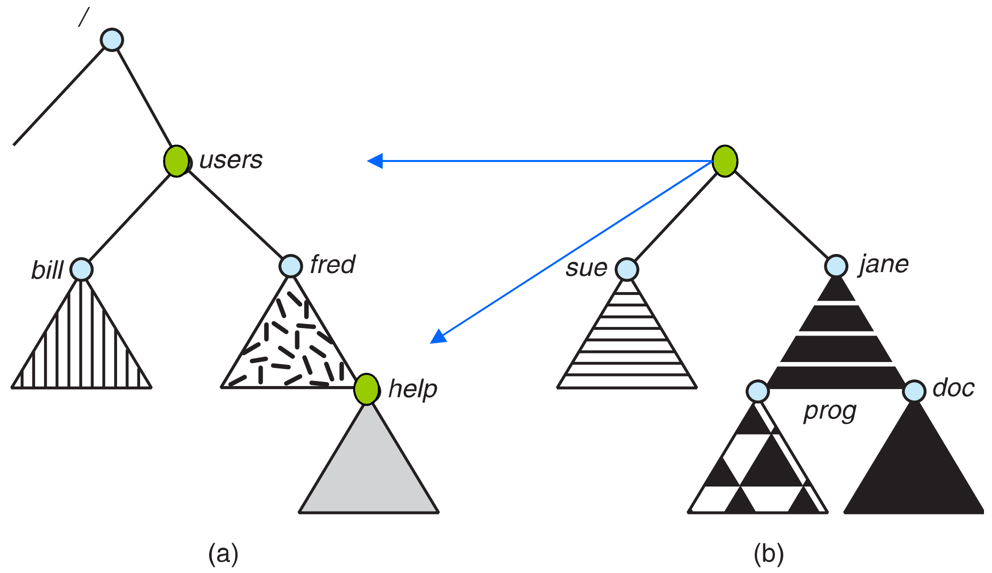


File System Mounting

- **A file system must be mounted before it can be accessed** by the various processes executing in the system
- Mount procedure. The operating system is given the name of the device and the **mount point** -- a location within the file structure where the file system is to be attached.
- Typically, **a mount point is an empty directory**.
  - For instance, on a UNIX system, a file system containing a user's home directories might be mounted as:
    - /home
  - To access the directory structure within that file system, we precede the directory names with /home, as in
    - /home/jane.

File System Mounting Example(1)

- Consider the file system depicted below



- The triangles represent **subtrees of directories**.
- Figure (a) shows an existing file system. Figure (b) shows an **unmounted volume residing on /device/dsk**.
- At this point, only the files on the existing file system can be accessed.

File-System Implementation

- We have system calls at the API level, but how do we implement their functions?
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block** (**superblock, master file table**) contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and **i-node(inode)** numbers, master file table

File-System Implementation (Cont.)

- Per-file **File Control Block** (**FCB**) contains many **details about the file**
  - inode number, permissions, size, dates
  - NTFS stores into in master file table using **relational DB structures**

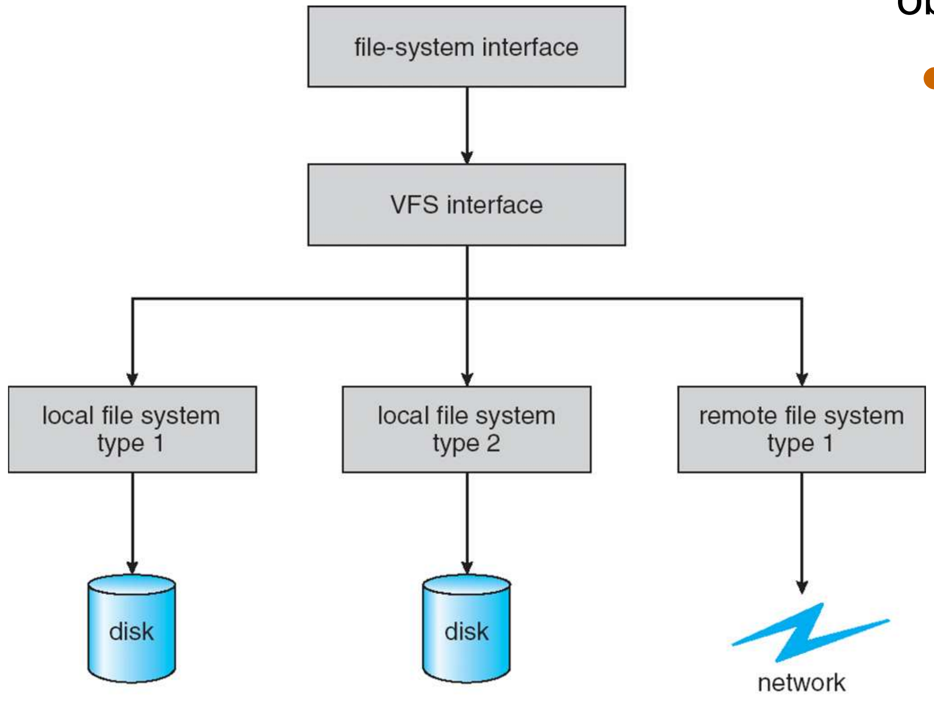
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

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**
  - 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

Virtual File Systems (Cont.)

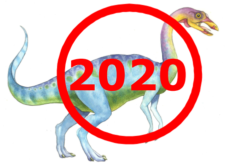
- The API is to the VFS interface, rather than any specific type of file system



- VFS defines set of operations on the objects that must be implemented
  - Every object has a pointer to a function table
    - • **`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 Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done **through a protection scheme**
- On distributed systems, files may be shared across a network
- **Network File System (NFS) is a common distributed file-sharing method**
- If multi-user system
  - **User IDs** identify users, allowing permissions and protections to be per-user
    Group IDs allow users to be in groups, permitting group access rights
  - Owner of a file / directory
  - Group of a file / directory



File Sharing – Remote File Systems

- Uses networking to allow file system access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using **distributed file systems**
  - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - Client and user-on-client identification is insecure or complicated
  - **NFS** is standard UNIX client-server file sharing protocol
  - **CIFS** is standard Windows protocol
  - Standard operating system file calls are translated into remote calls
- Distributed Information Systems (**distributed naming services**) such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing

Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol) and Ethernet
- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner

NFS - Interconnected workstations

- A remote directory is mounted over a local file system directory
  - The mounted directory looks like an integral sub-tree of the local file system, replacing the sub-tree descending from the local directory
- Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided
  - Files in the remote directory can then be accessed in a transparent manner
- Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

NFS (Cont.)

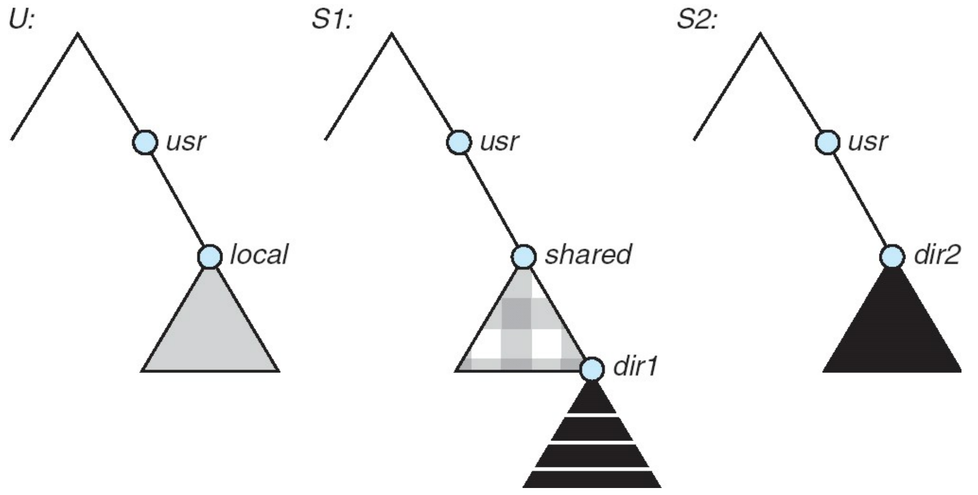
- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services

NFS Mount

- For a remote directory to be accessible in a transparent manner from a particular machine (say M1), a client of M1 must first carry out a mount operation.
- The semantics of the operation involve mounting a remote directory over a directory of a local file system.
- Once the mount operation is completed, the mounted directory looks like an integral sub-tree of the local file system, replacing the sub-tree descending from the local directory.
- The local directory becomes the name of the root of the newly mounted directory.
- Specification of the remote directory as an argument for the mount operation is not done transparently; the location (or host name) of the remote directory has to be provided.
- However, from then on, users on machine M1 can access files in the remote directory in a totally transparent manner.

NFS Mount Example

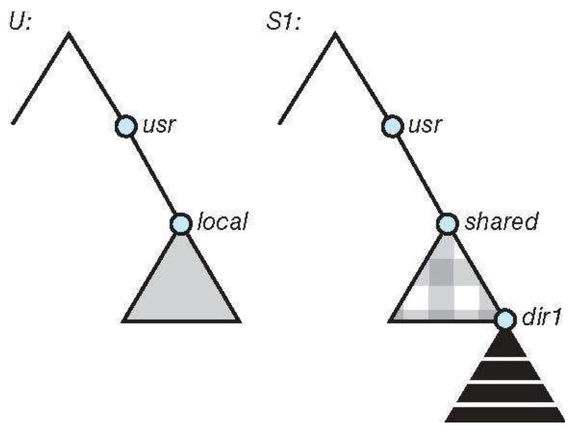
- Consider the file system depicted in the figure below, where the triangles represent subtrees of directories that are of interest.



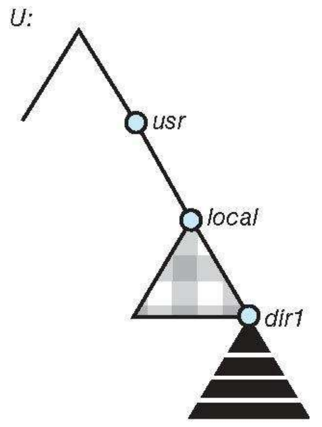
- The figure shows three independent file systems of machines named U, S1, and S2.
- At this point, on each machine, only the local files can be accessed.

NFS Mount Example

- Figure (b) below shows the effects of mounting S1:/usr/shared over U:/usr/local. This figure (b) depicts the view users U have of their file system

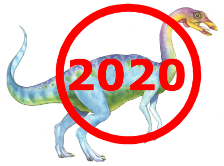


(a)



(b)

- After the mount is complete, the users can access any file within the dir1 directory using the prefix /usr/local/dir1.
- The original directory /usr/local on that machine is no longer visible.



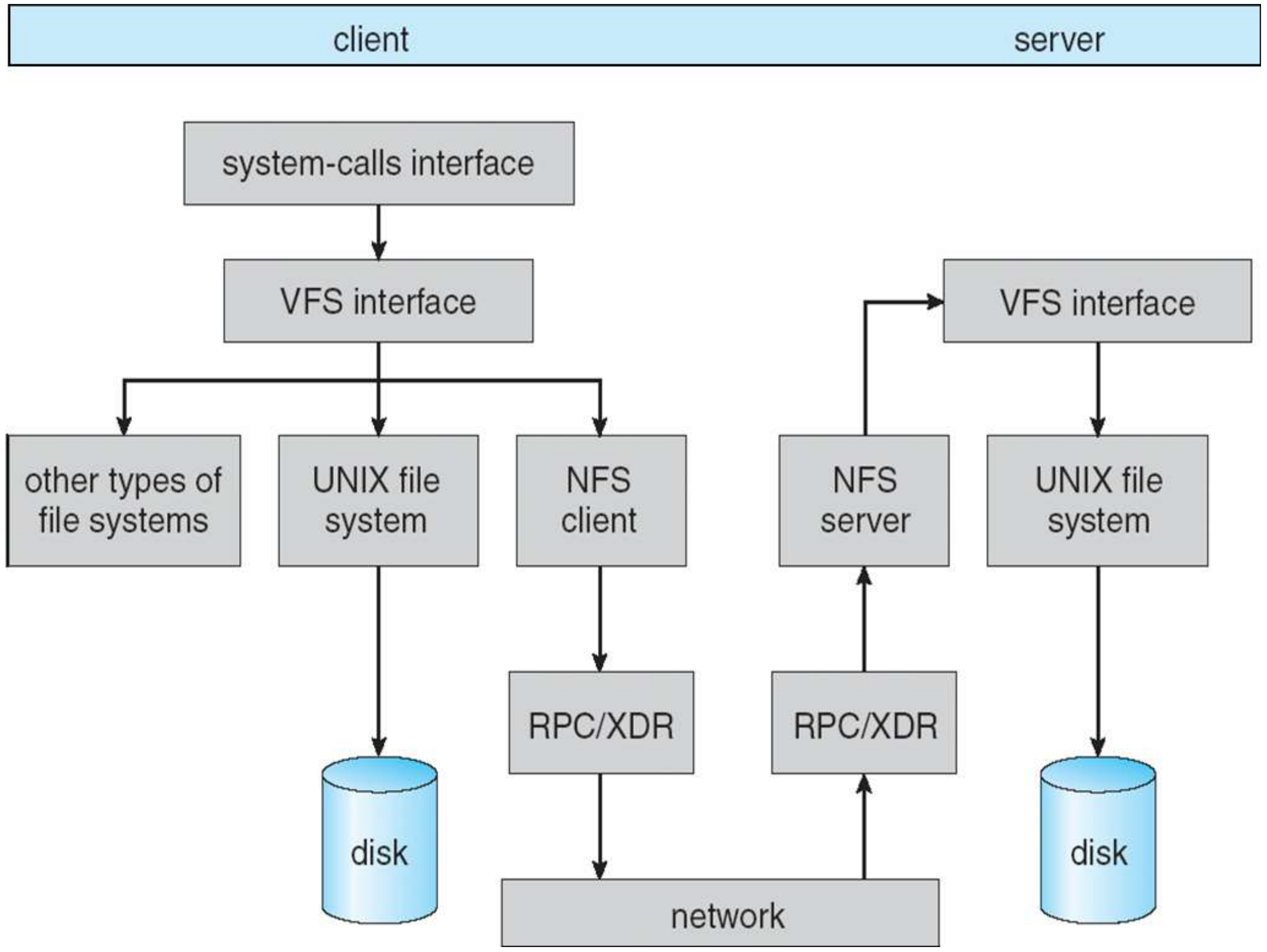
NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments (NFS V4 is just coming available – very different, stateful)
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does not provide concurrency-control mechanisms

Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the **open, read, write**, and **close** calls, and **file descriptors**)
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
    - The VFS activates file-system-specific operations to handle local requests according to their file-system types
    - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture
    - Implements the NFS protocol

Schematic View of NFS Architecture



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