File Systems: Interface

Last Update: May 10, 2023

Objectives and Outline

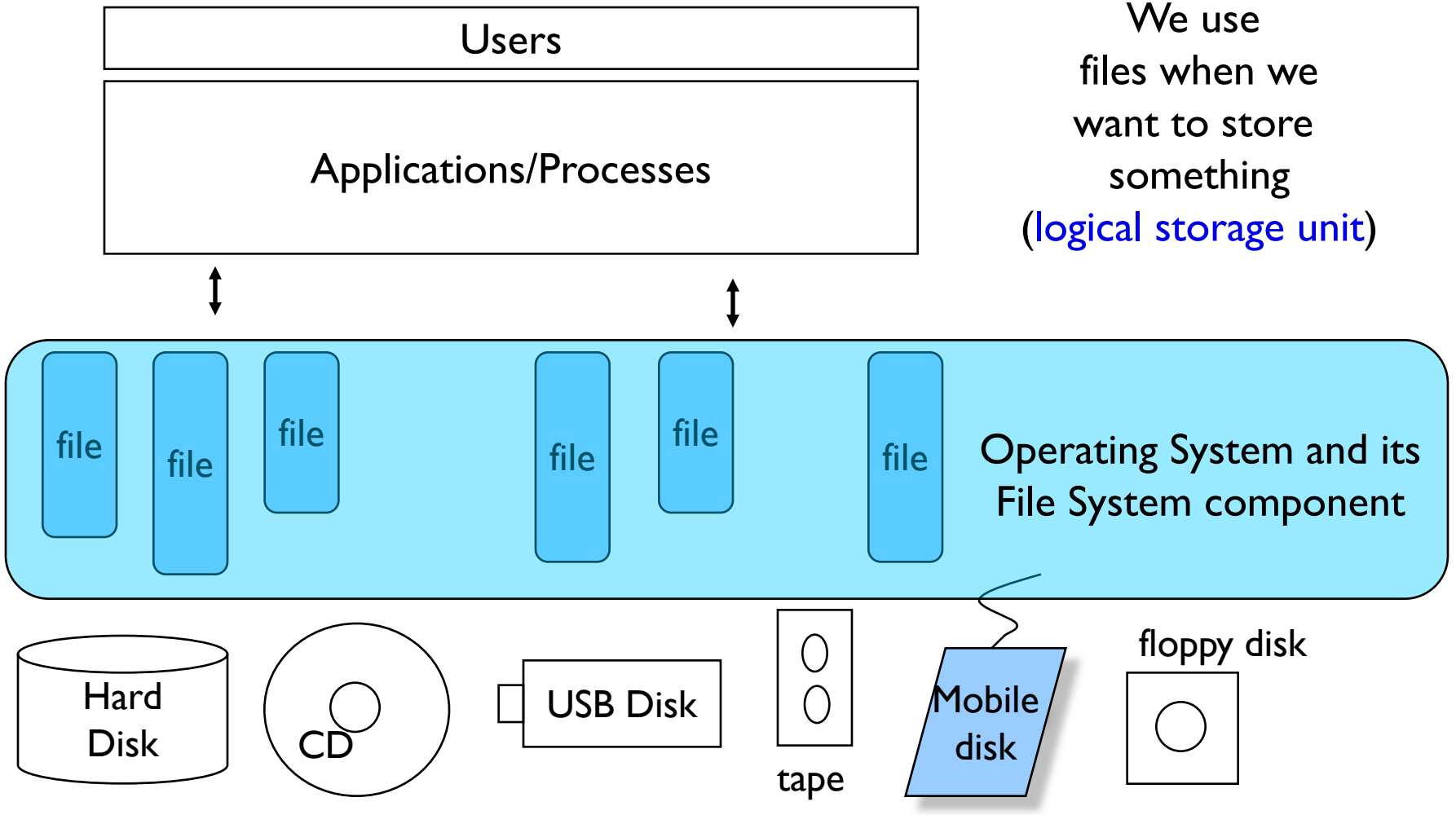
Objectives

- 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.

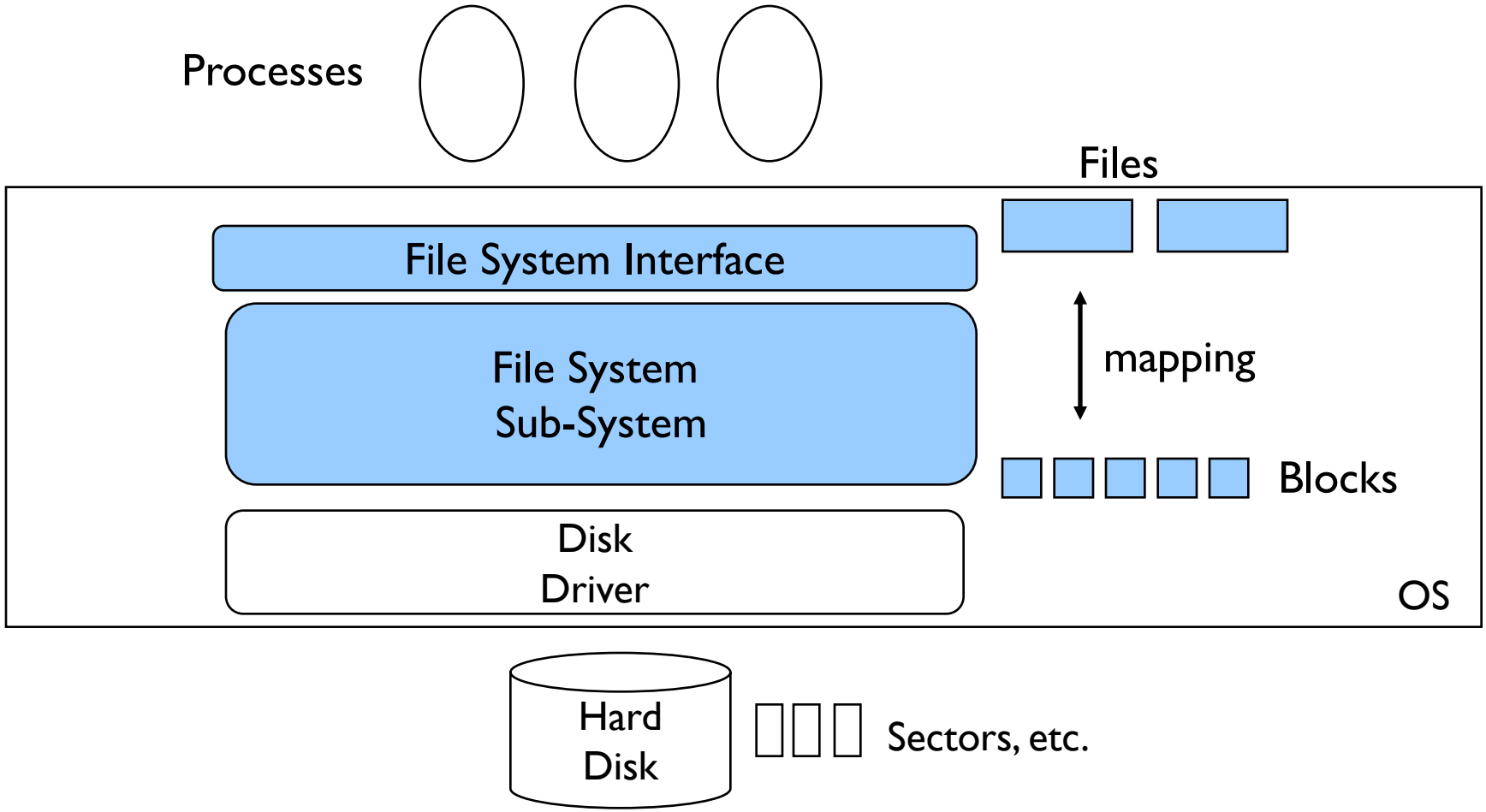
Outline

- File Concept
- Access Methods
- Directory Structure
- File-System Mounting
- File Sharing
- Protection
- VFS
- DFS
- NFS

File Concept

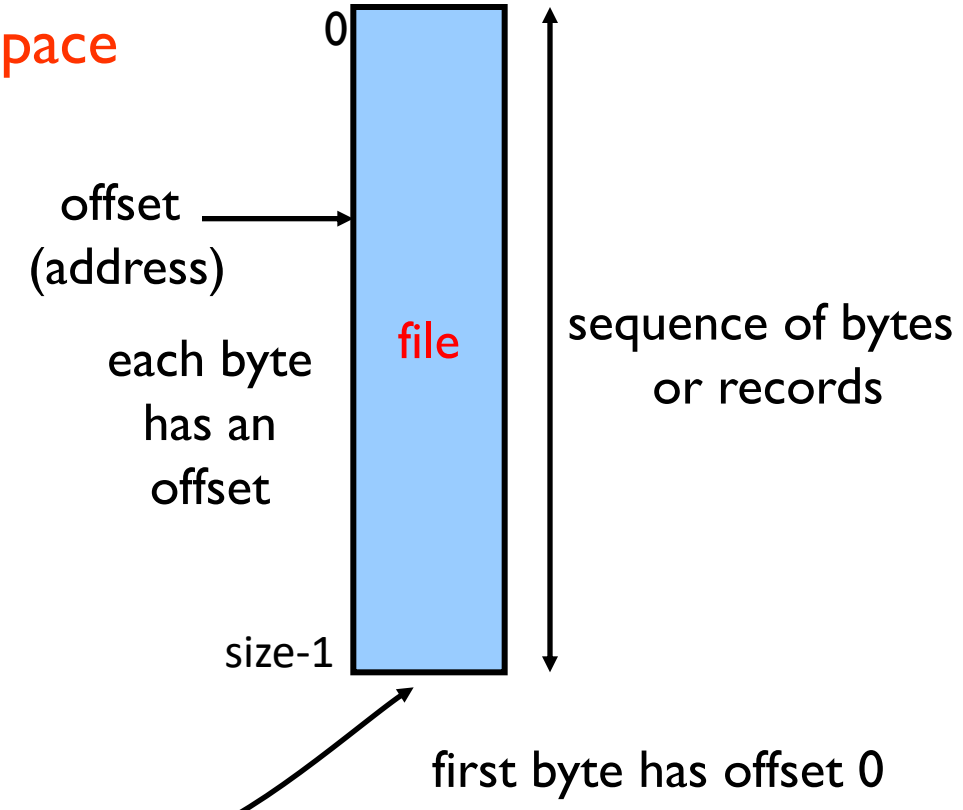


File Concept



File Concept

- Contiguous logical address space
  (a logical storage unit)
- Content:
  - Data
    - numeric
    - character
    - binary
  - Program



File Content Structure

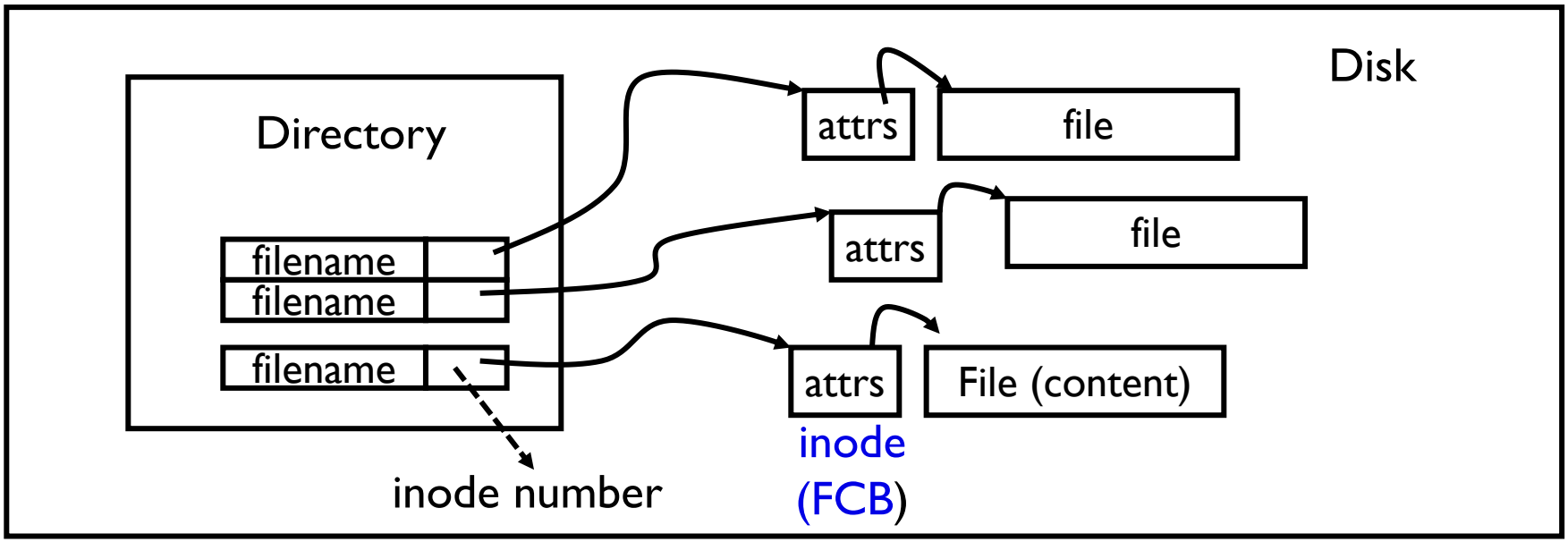
- No structure: sequence of bytes (Linux, etc.)
- Simple record structure
  - Lines; Fixed length records; Variable length records
- Complex Structures
  - Formatted document (application specific)
  - Relocatable executable file (OS recognizes this)
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
  - Application Programs (usually)

File Attributes

- Name: only information kept in human-readable form.
- Identifier: unique number identifying the file within file system.
  - In Linux: inode number
- Type: needed for systems that support different types.
  - Regular files, special files, ...
- Location: information about device and location of file data in device
- Size: current file size (in bytes)
- Protection: access control information: who can do reading, writing, executing, etc.
- Timestamps. Creation, modification date/time.
- User identification: ID of User, who created the file.

Files and Directories

- A file system consists of two parts, stored on disk in the partition controlled by the file system:
  - files (store content).
  - directory structure: maps filenames to file control blocks(files)



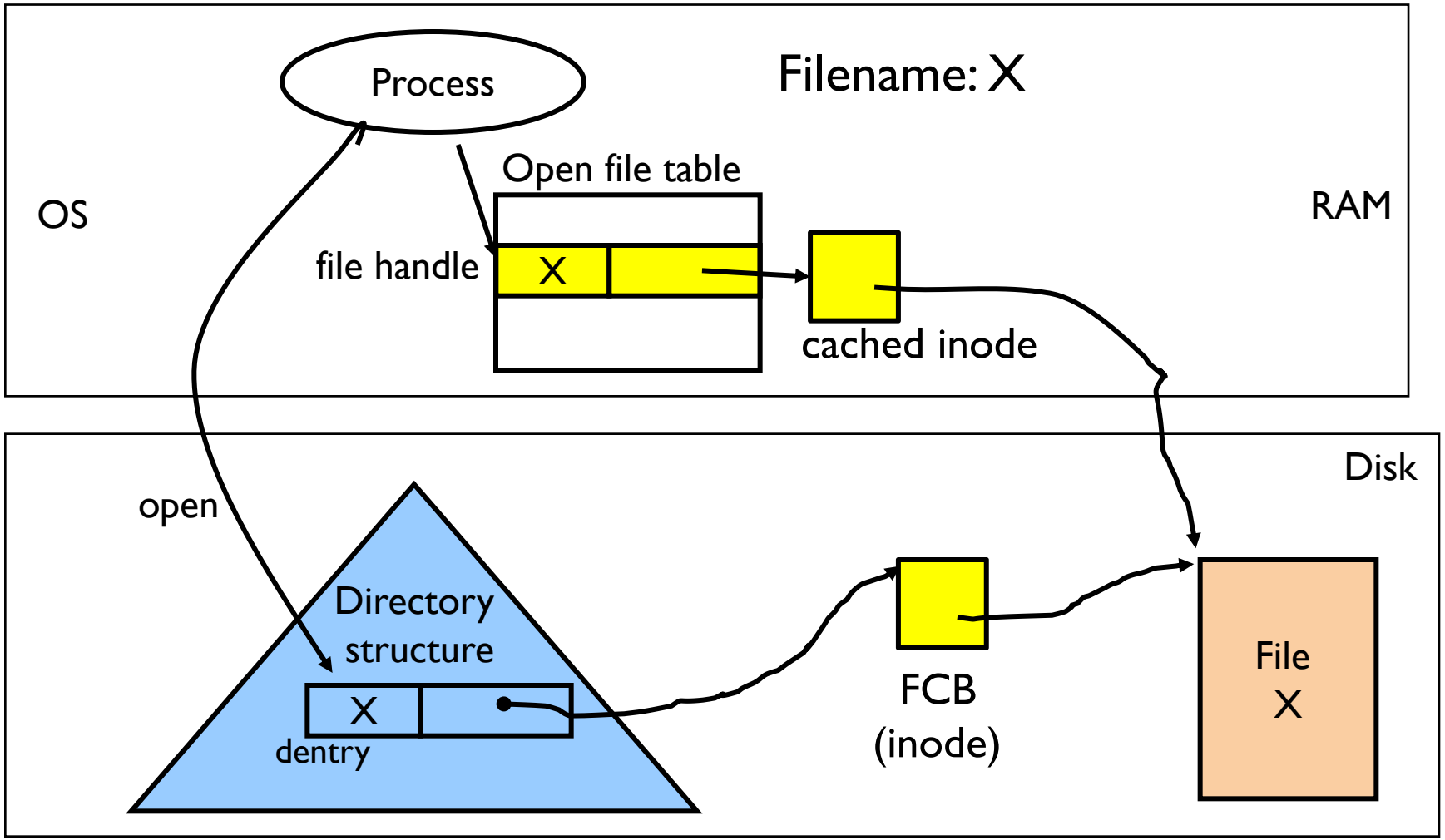
File Operations

- File is an abstraction.
- Common operations that are supported by OS.
  - create – create a file (allocate an inode – file control block).
  - open (F) – search the directory structure on disk for the entry of file with name F, and cache it in memory, and return a file handle.
  - close (F) – write changed file metadata back to disk. Remove open file entry for file.
  - write – write some amount of data to file. Uses file handle.
  - read – read some amount of data from file. Uses file handle.

File Operations

- Common operations that are supported by OS.
  - reposition within file (update file position pointer; *next* read or write from that position).
  - delete - deallocate file space, dentry set to free.
  - truncate (set the size to 0 or some value)

Opening a file

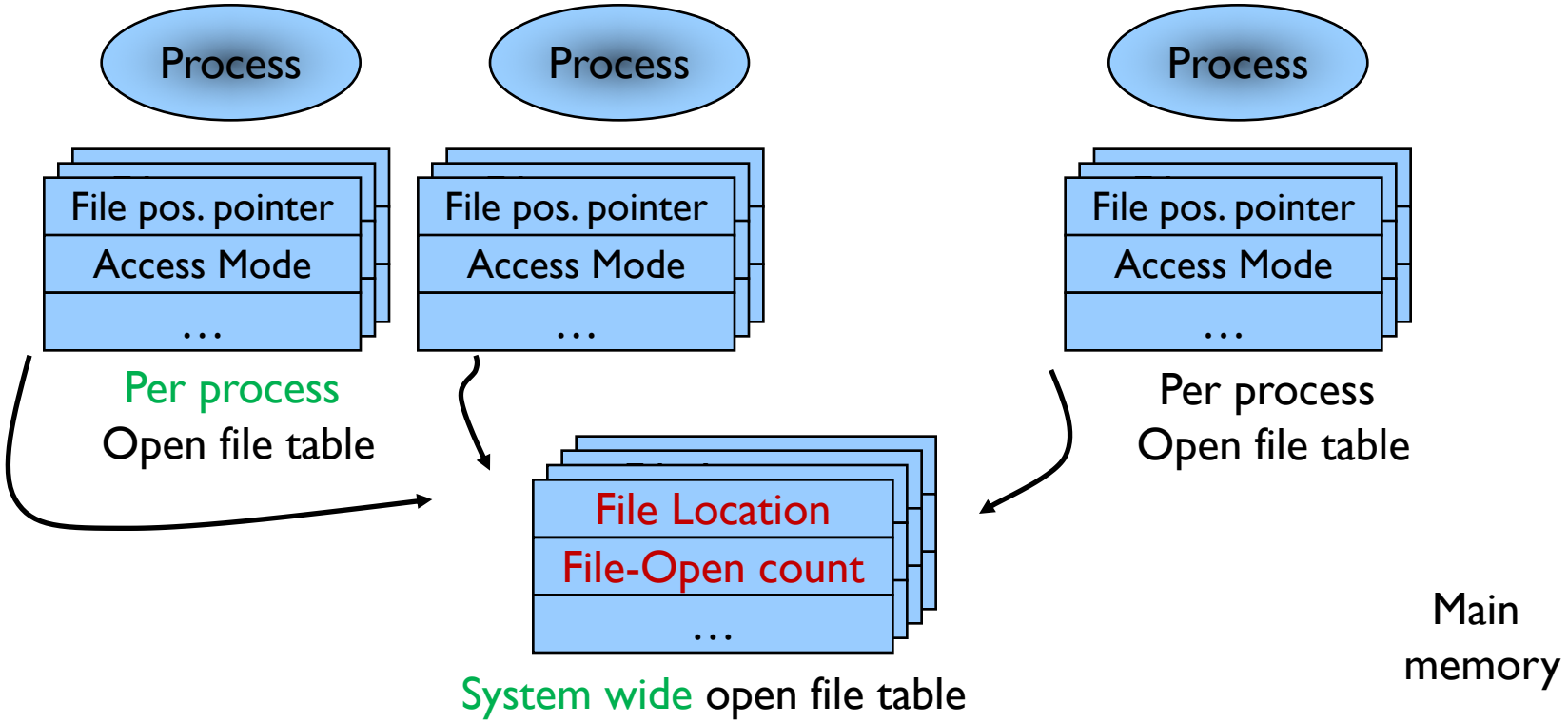


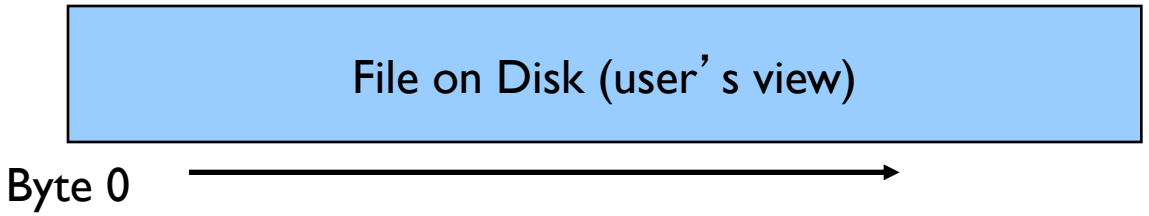
Subsequent operations will be able to directly access file metadata and data

Open Files

- Several pieces of data are needed to manage open files:
  - File position pointer (fp): pointer to last read/write location, per process that has the file open (i.e., for each opening of the file).
  - Access mode: per-process access mode information, such as file opened for "reading", for "writing", for "r/w".
  - File-open count: number of times a file is opened – to allow removal of data from open-file table when last process closes it
  - Disk location of the file: information about where file data is on the disk.

Open file information





File Locking

- Operating systems provide file locking to prevent simultaneous access when needed.
- Mediates access to a file
- Locking can be SHARED or EXCLUSIVE.
  - If shared, multiple processes can access concurrently.
  - If exclusive, one process at a time can access the file.
- Mandatory or advisory:
  - Mandatory – lock and unlock supported. If a file is locked, no other process can access the file.
  - Advisory – lock and unlock supported. But, it is up to the processes to use them. Responsibility of locking belongs to processes. (Linux provides advisory locking)

File Content Types

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 content interpreted by the application

Access Methods

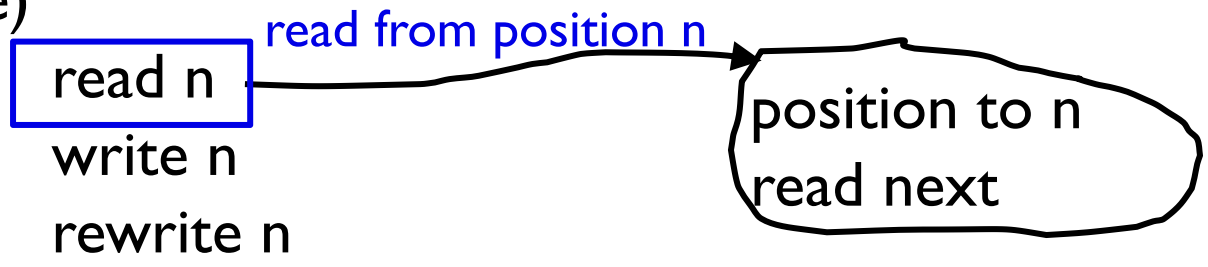
Sequential Access

read next — internally file position pointer maintained

write next — reading or writing a byte or block

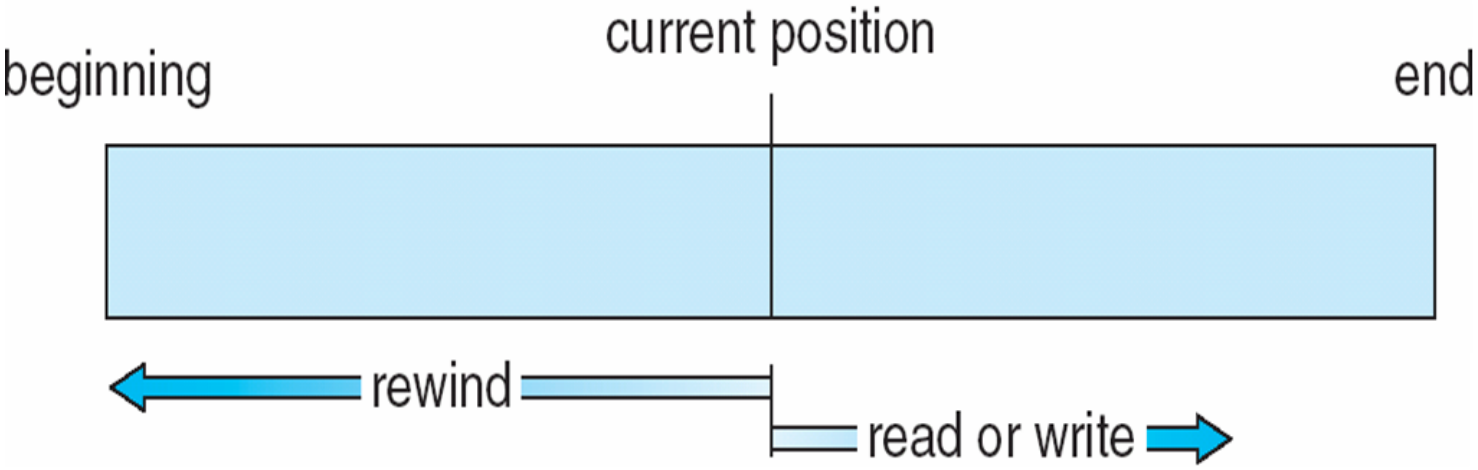
reset

Direct Access (relative access – relative to the beginning of the file)



n: relative byte (block) number (byte number) – offset.
First byte of the file has offset 0.

Sequential-access File

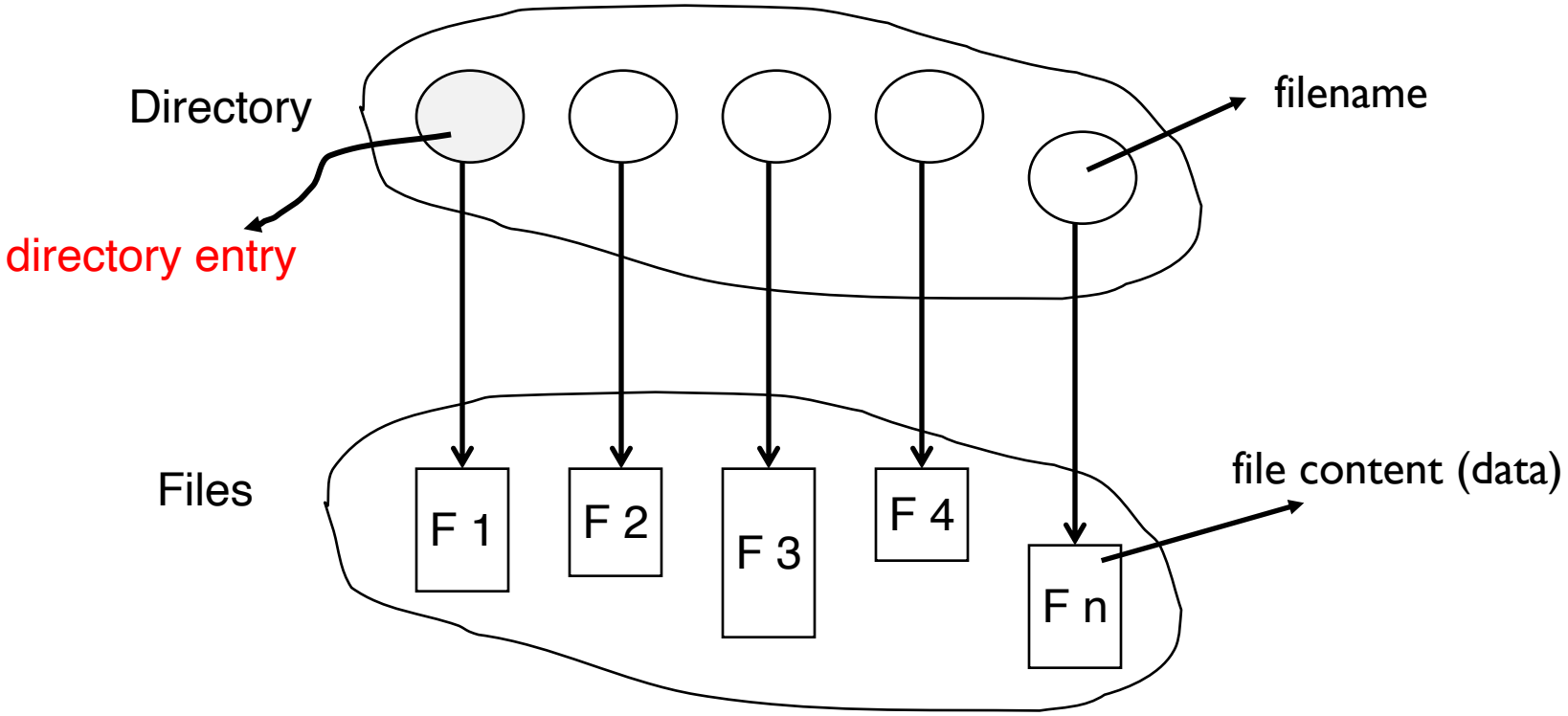


Simulation of Sequential Access on Direct-access File

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

Directory Structure

- A collection of entries containing information about files. Directory structure helps organizing files.



Both the directory structure and the files reside on disk

Operations on a Directory

- Operations that need to be performed on a directory.
  - search for a file: given a filename, find out the corresponding directory entry (dentry).
  - create a file (add a *new dentry* into the directory)
  - delete a file (*remove* the related *dentry* from directory)
  - rename a file (*change* the *filename* in the *respective dentry*)
  - list a directory: list the *names of all files in that directory*. For each file, more information may be printed out.
  - traverse the file system: starting from the root directory, go though *all directory entries*, including the *sub-directories and their entries*, recursively. That means traverse the whole directory structure.

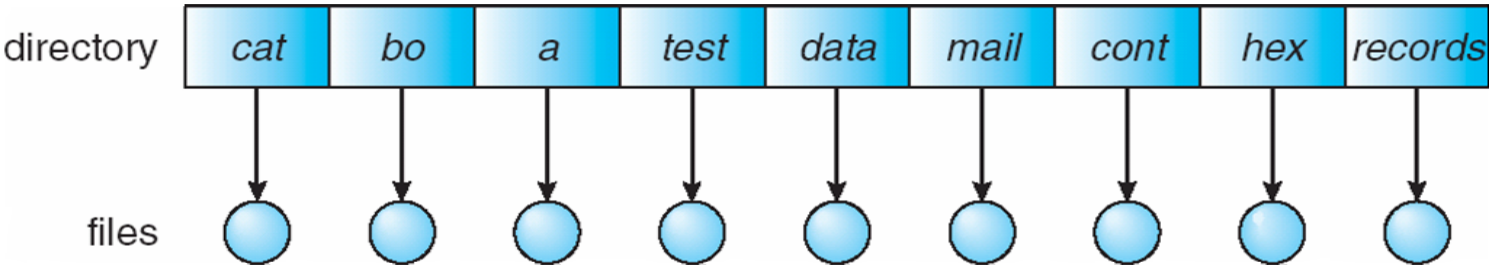
Organizing the Directory (Logically) of a File System

Goals:

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

Single-Level Directory

- A single directory for all users

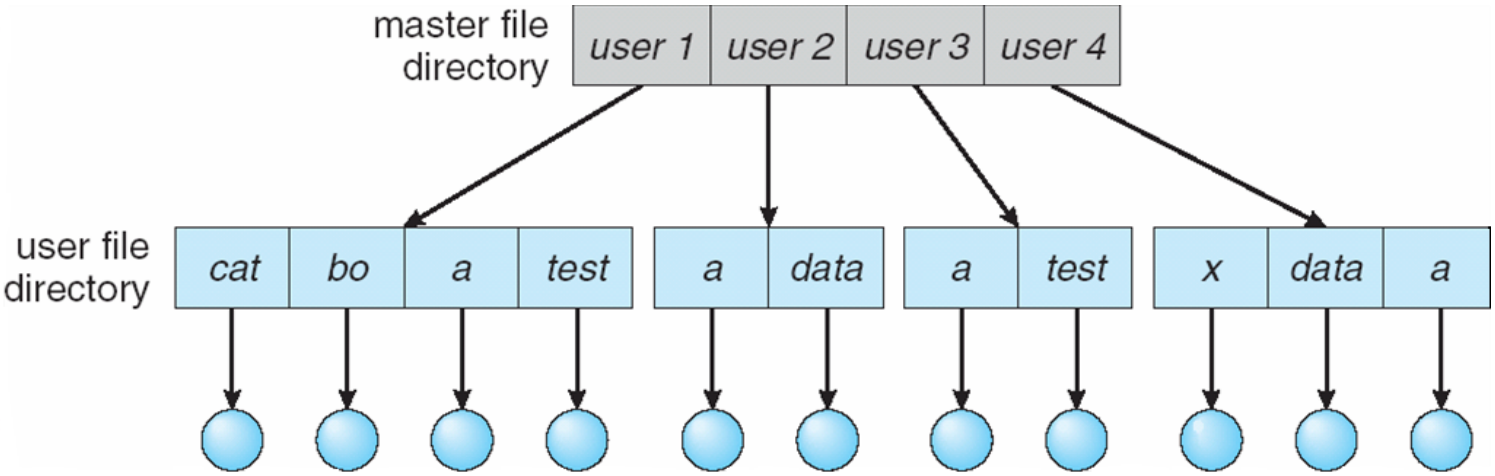


Naming problem

Grouping problem

Two-Level Directory

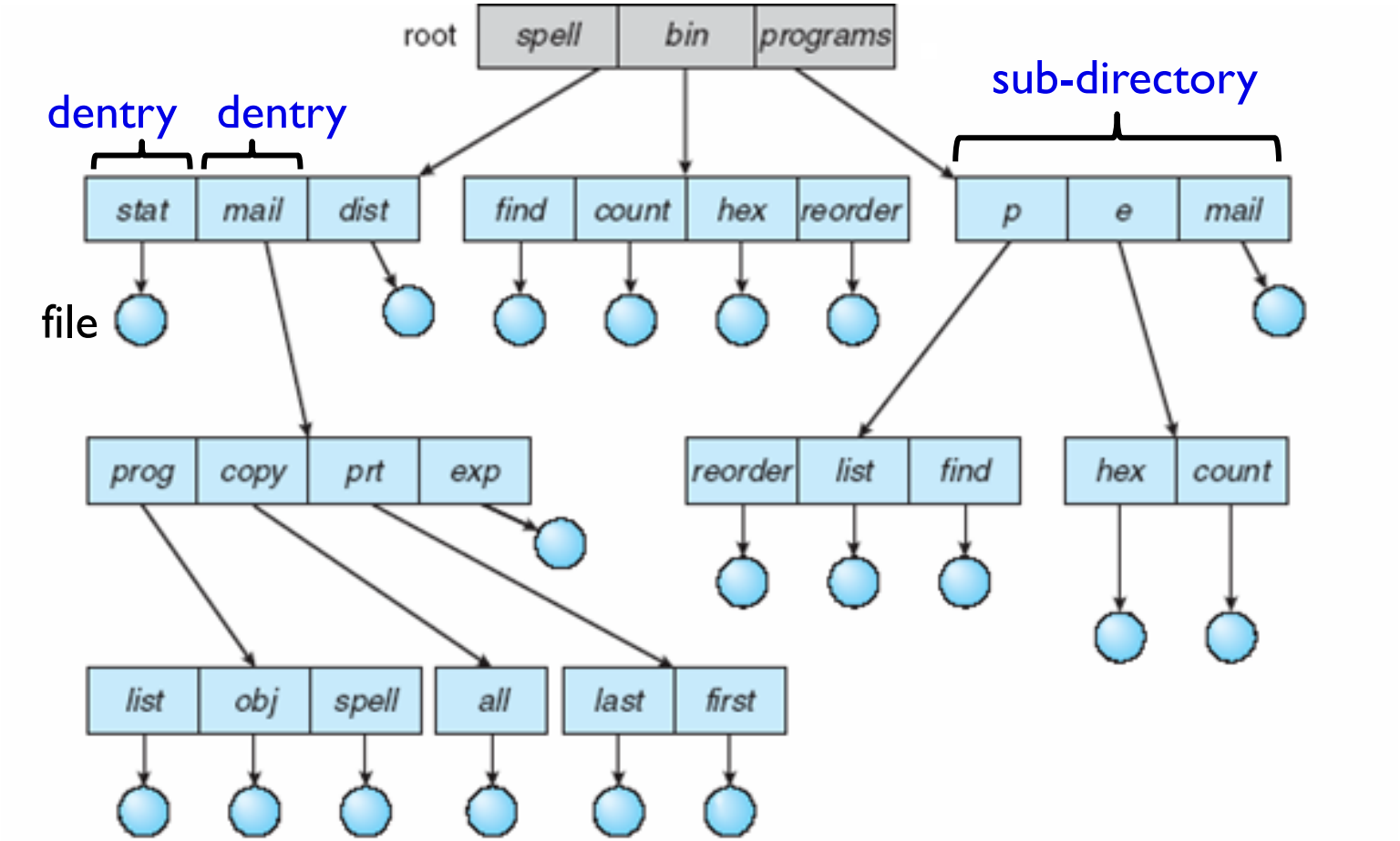
- Separate directory for each user



- Pathname used for a file
- Can have the same filename for different users
- Efficient searching
- No grouping capability

Tree-Structured Directories

Directory structure is a tree. Directories (sub-directories) are nodes of the tree.



Commonly used organization method for directory information of a file system.

Tree-Structured Directories

- Efficient searching
  - A pathname indicates where a file is. Parse the pathname and follow those subdirectories indicated in the pathname

```
/usr/home/ali/projects/cs342/file.txt
```

- Grouping capability
- Current directory (*working directory*) concept
  - cd /spell/mail/prog

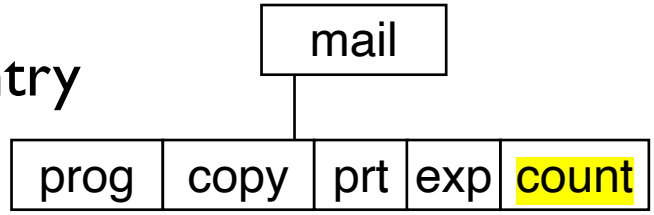
Tree-Structured Directories

- Absolute or relative pathname to refer to a file
- Creating a new file is done in current directory
- Deleting a file is done in current directory

  rm <file-name>
- Creating a new subdirectory is done in current directory

  mkdir <dir-name>

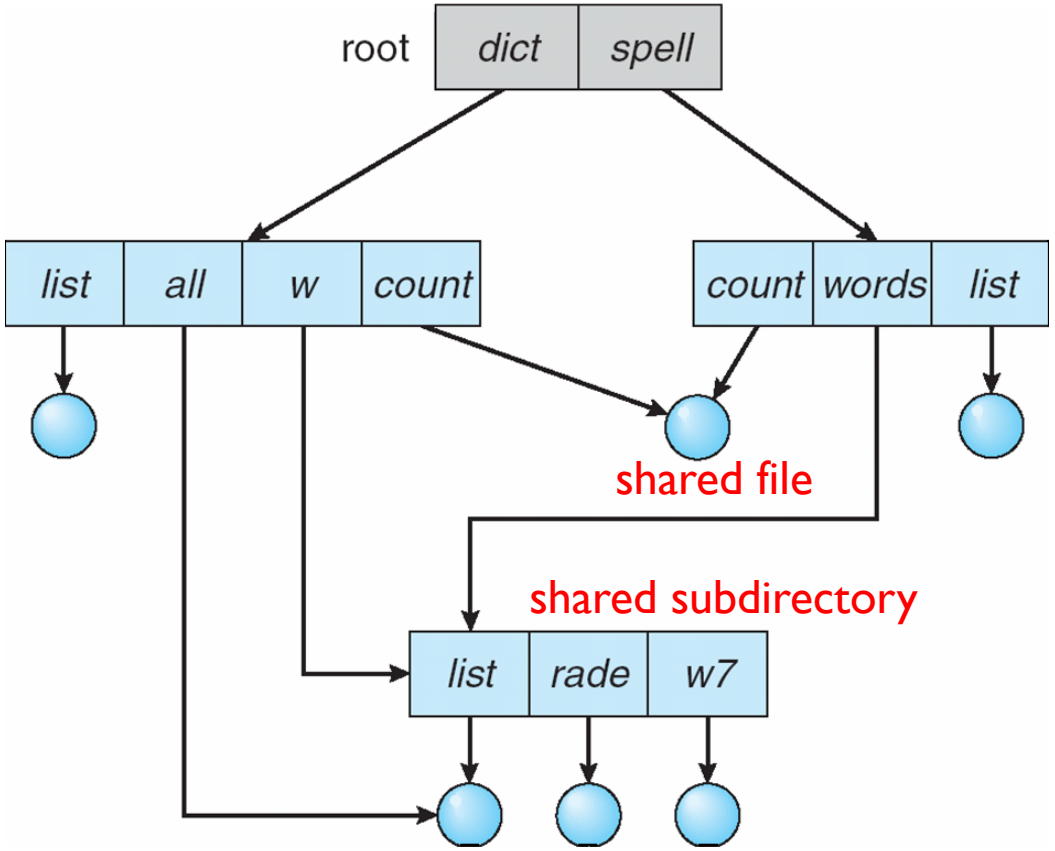
Example: if the current directory is /mail,
mkdir count creates another directory entry



Deleting "mail" ⇒ deleting the entire subtree rooted at "mail"

Acyclic-Graph Directories

- Have shared subdirectories and files



Referring to a file with different names or from different directories (file sharing)

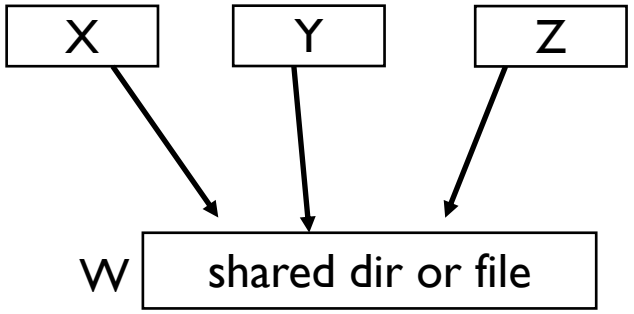
- Two methods to share a file or sub-directory in another sub-directory (or other sub-directories).
- 1) The original directory entry (dentry) for the file is copied into another sub-directory. Hence multiple sub-directories will have identical dentries for the file. In Linux this is a hard link. The inode (FCB) of the file keeps a reference count.

```
ln <sourcefile> <targetfile>
```

- 2) Other sub-directory just has a soft link (symbolic link) to the file. A soft link is another type of directory entry, that is actually referring to a special file (of type symbolic_link). That special file (that we can not see) contains the pathname of the file to which we are linking.

```
ln -s <sourcefile> <targetfile>
```

Referring to a file with different names or from different directories

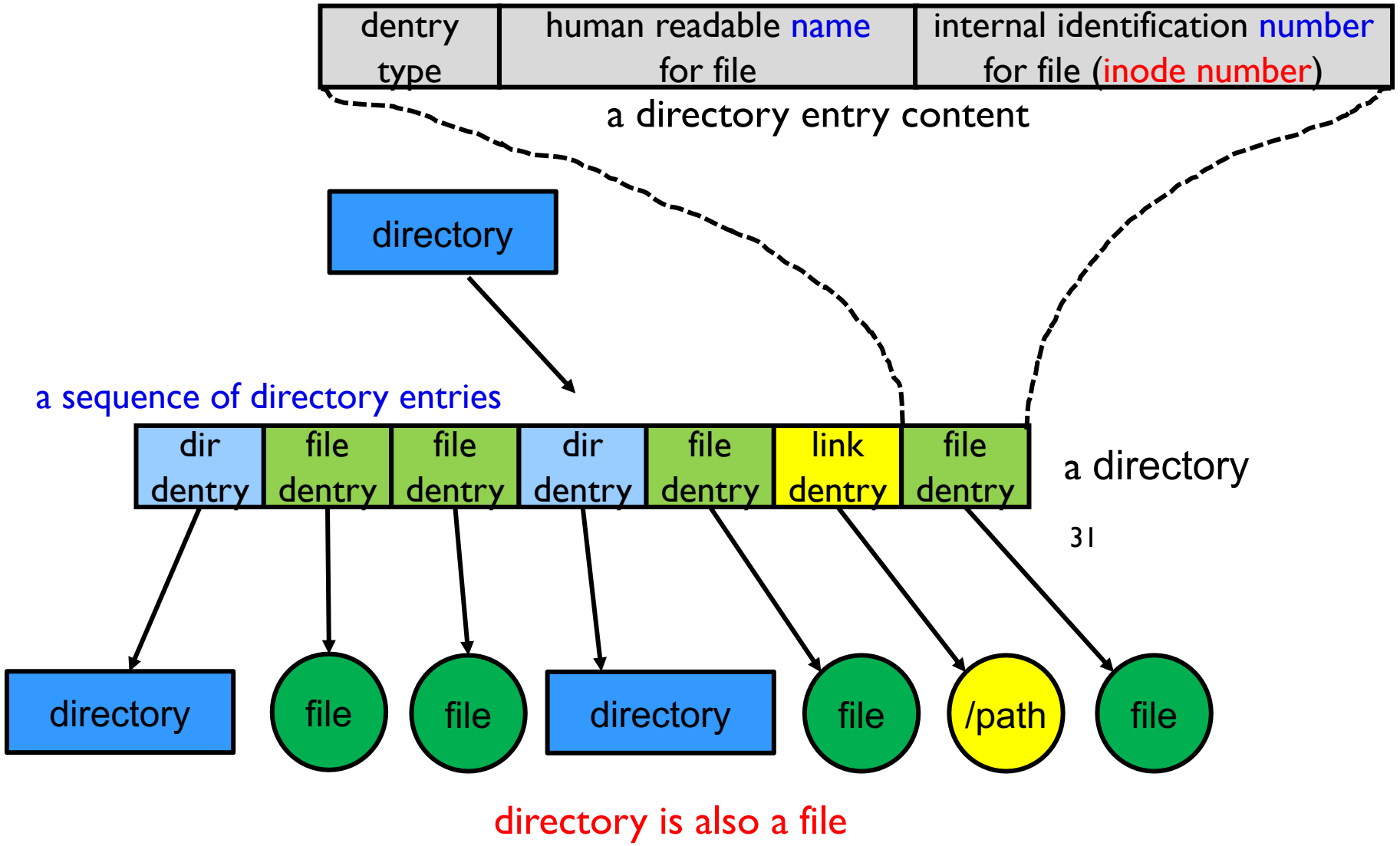


- 1) Two or more different names for a file (in the same or other directory)
  - Identical directory entries (hard link)
  - Uses reference count
  - When deleted (unlinked), count decremented. When reaches zero, file actually removed,
- 2) new directory entry type: link (i.e., soft link or symbolic link)
  - The entry is for a special file which includes the (path)name of the file to which we are linking.
  - Resolve the link: system looks into the special file for filename and follows.

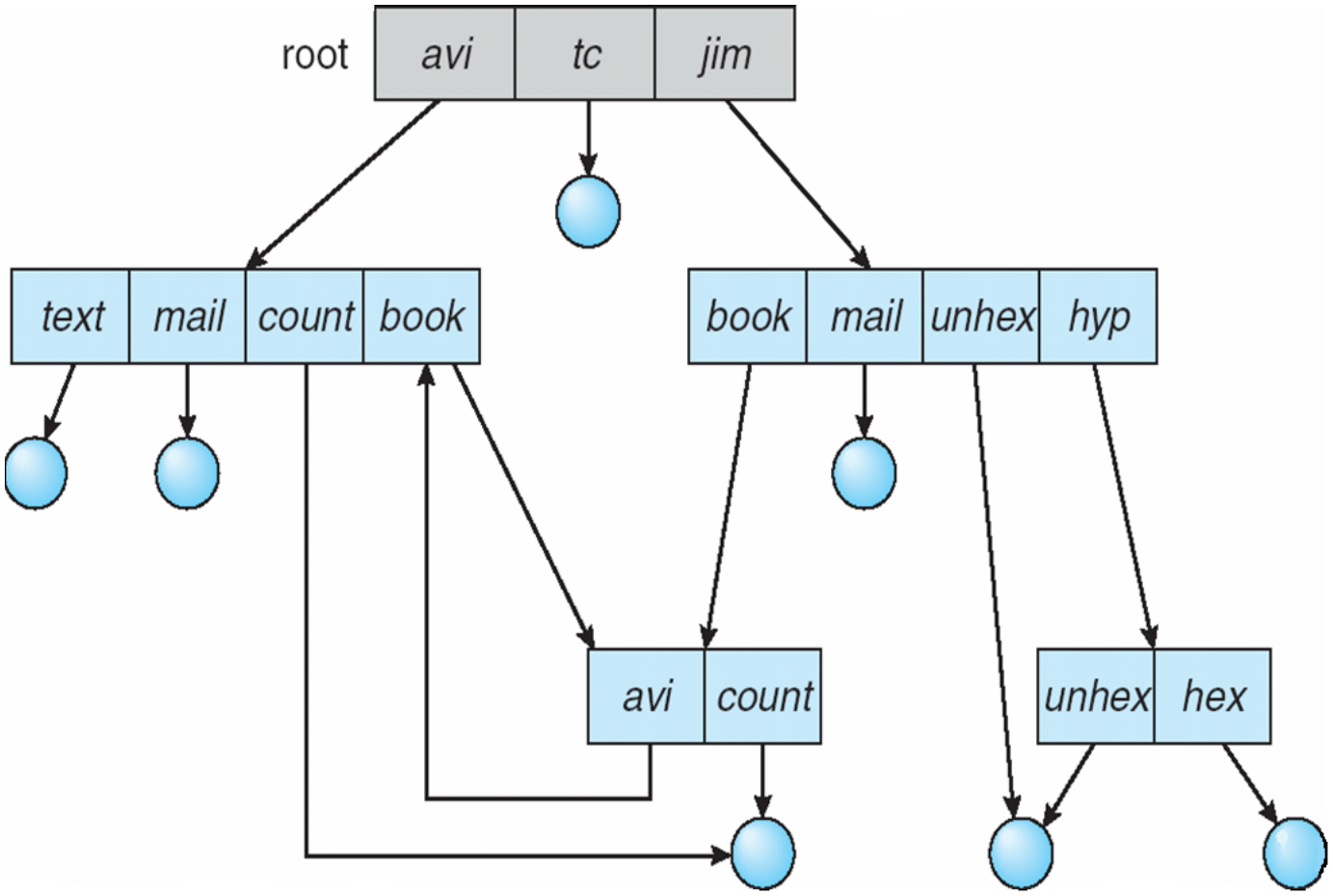
Directory entries

- Hence we have 3 types of files and therefore 3 types of directory entries. (we can have more)
  - regular file (all files we work with, no matter what the content type is).
    `dentry-type = REGULAR`
  - directory file (stores directory information – stores a sequence of directory entries). *We can not directly see the content of such a file.*
    `dentry-type = DIRECTORY`
  - special file (for example, stores the pathname of a file which we linked to in a soft manner). *We can not directly see the contents of such a file.*
    `dentry-type = LINK`

Directory entries



General Graph Directory



General Graph Directory

- How do we guarantee no cycles?
  - Allow only links to files not subdirectories
  - Every time a new link is added use a cycle detection algorithm to check if it causes a cycle (closes the loop)
    - Costly (especially for disk).

File Sharing and Protection

- Sharing of files on multi-user systems is desirable.
  - Multiple people having account on the computer.
- Sharing requires also a protection scheme.
- We want controlled access to the file.
- The owner should be able to control access.

File Sharing and Protection

- Each user that has an account in the computer has a username and a unique user ID (UID)
- The administrator can create groups. A group may have a set of usernames (users) associated with it. Each group has a unique group ID (GID).
- Example: group os_team: ali, veli, selcuk, ….

File attributes for a file

…
UID (user ID)
GID (group ID)
User (owner) permissions
Group permissions
Other people permissions
……

File Sharing – Multiple Users Protection

- Protection is based on the use of UIDs and GIDs.
- Each file has associated protection bits (permissions) for UID and GID.
  - User ID: read, write, execute?
  - Group ID: read, write, execute?
- UIDs identify users, allowing permissions and protections to be per-user.
- GIDs allow users to be in groups, permitting group access rights.

Protection

- File owner/creator should be able to control:
  - *what* can be done (read, write, execute….)
  - by *whom* (owner, others, group member…)
- Types of access for a file (what can be done)
  - Read (a file)
  - Write (a file)
  - Execute (a file)
  - Append (a file)
  - Delete (a file)
  - List (a file)
  - Attribute change

For directories:

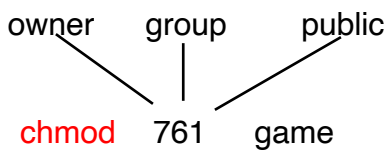
- Reading means listing
- Writing means being able to create a file
- Executing means being able to change into the directory

Basic Protection

- Mode of access: read, write, execute
- Three classes of users

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

- Ask the admin 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.



Attach a group to a file

chgrp G game

ACL

- ACL: access control list
- Associate with each file (or directory) a list of usernames (user ids) and their access rights.
- ACL may be long (a lot of users can be listed).
- If both ACL and basic protection is used, more specific access rights comes first.
- Commands like setfacl, getfacl can be used to set ACL rights for a file and learn about them.
  - Example: `setfacl -m u:lisa:r file`
    (give read access to Lisa).

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/

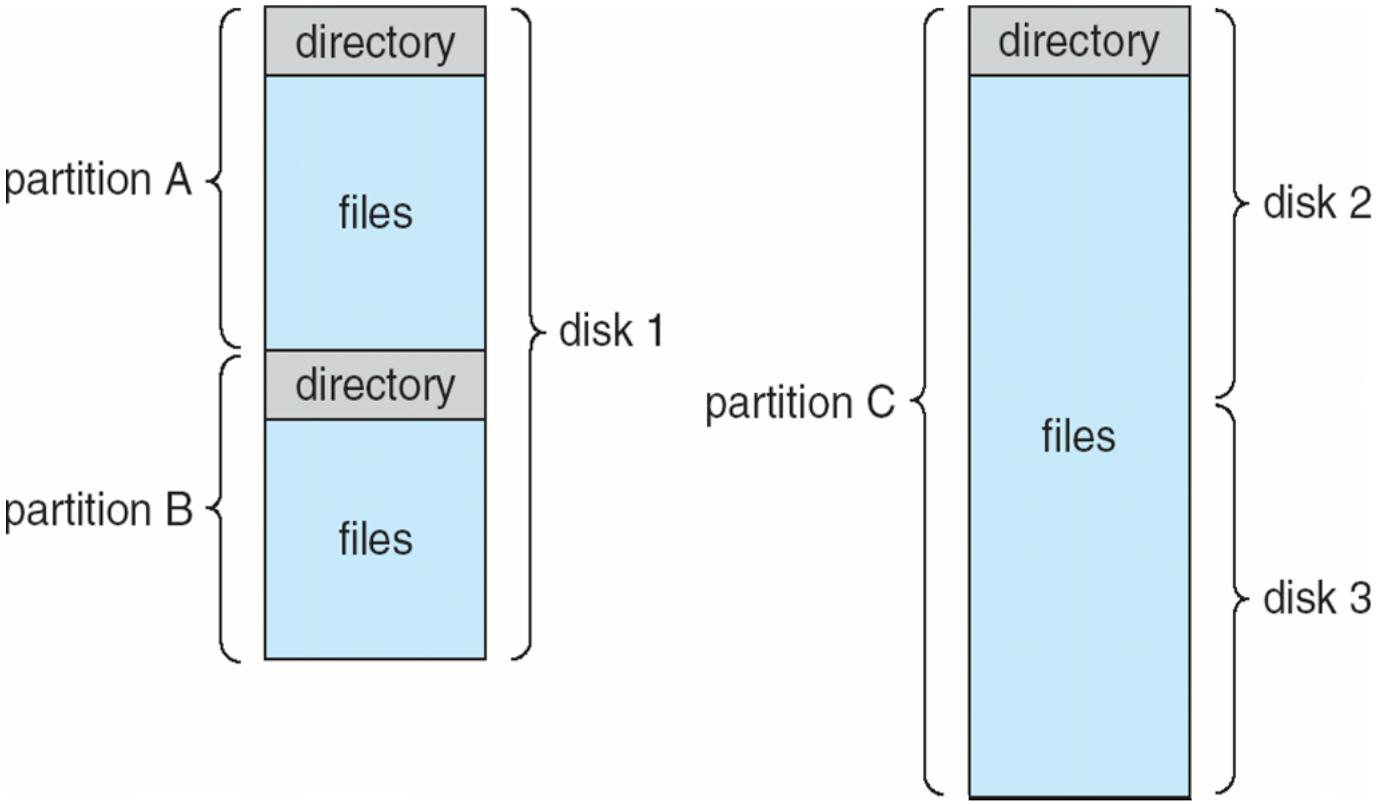
File Sharing – Consistency Semantics

- Consistency semantics specify what should be expected from the system when multiple users access a file concurrently.
  - Unix file system (UFS) implements:
    - Writes to an open file visible immediately to other users who opened the same file.
    - A single file image accessed by all users
  - AFS has session semantics
    - Writes only visible to sessions starting after the file is closed (to processes who opened the file after the file is closed).
    - Multiple file images accessed by users.

Disk Structure

- Disk can be subdivided into partitions
  - Partitions also known as minidisks, slices
  - Disks or partitions can be RAID protected against failure
- Disk or partition can be used
  - raw – without a file system, or
  - formatted with a file system.
- *Entity (partition or disk or multiple disks) containing a file system is known as a volume.*
- Each volume has a Directory, that keeps track of and organizes files.

A Typical File-system Organization



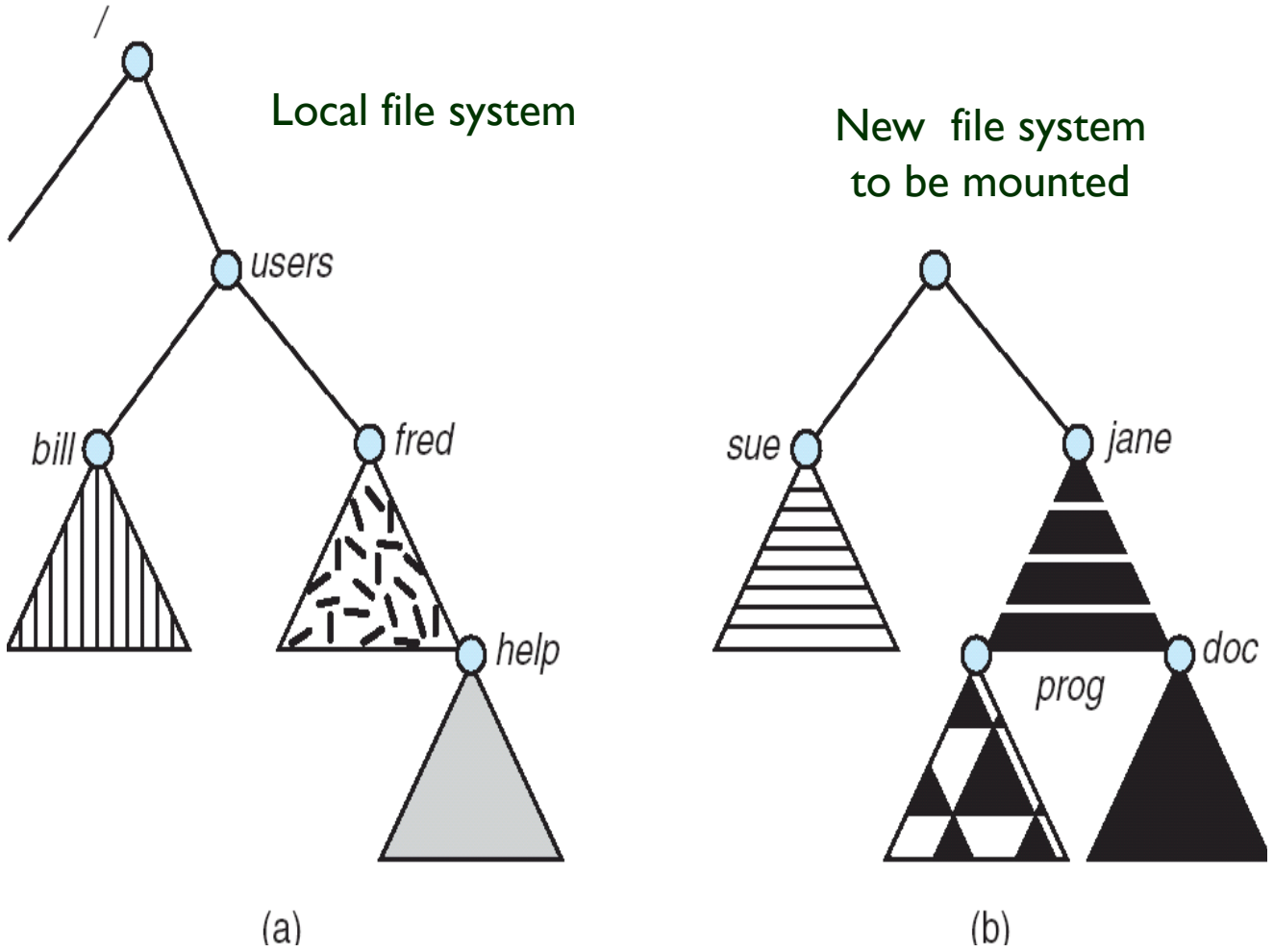
Many File Systems

- There may be many general-purpose file systems and also many special-purpose file systems all within the same operating system of a computer (Linux, Solaris, ...)
- General purpose file systems: FAT32, exFAT, NTFS, ext4 (Linux FS), ext3, ZFS, UFS, APFS, HFS, XFS, MinixFS, ...
- Special-purpose file systems:
  - procfs – a virtual file system that presents information about current processes in the system (retrieved from kernel).
  - tmpfs: temporary file system created in RAM.

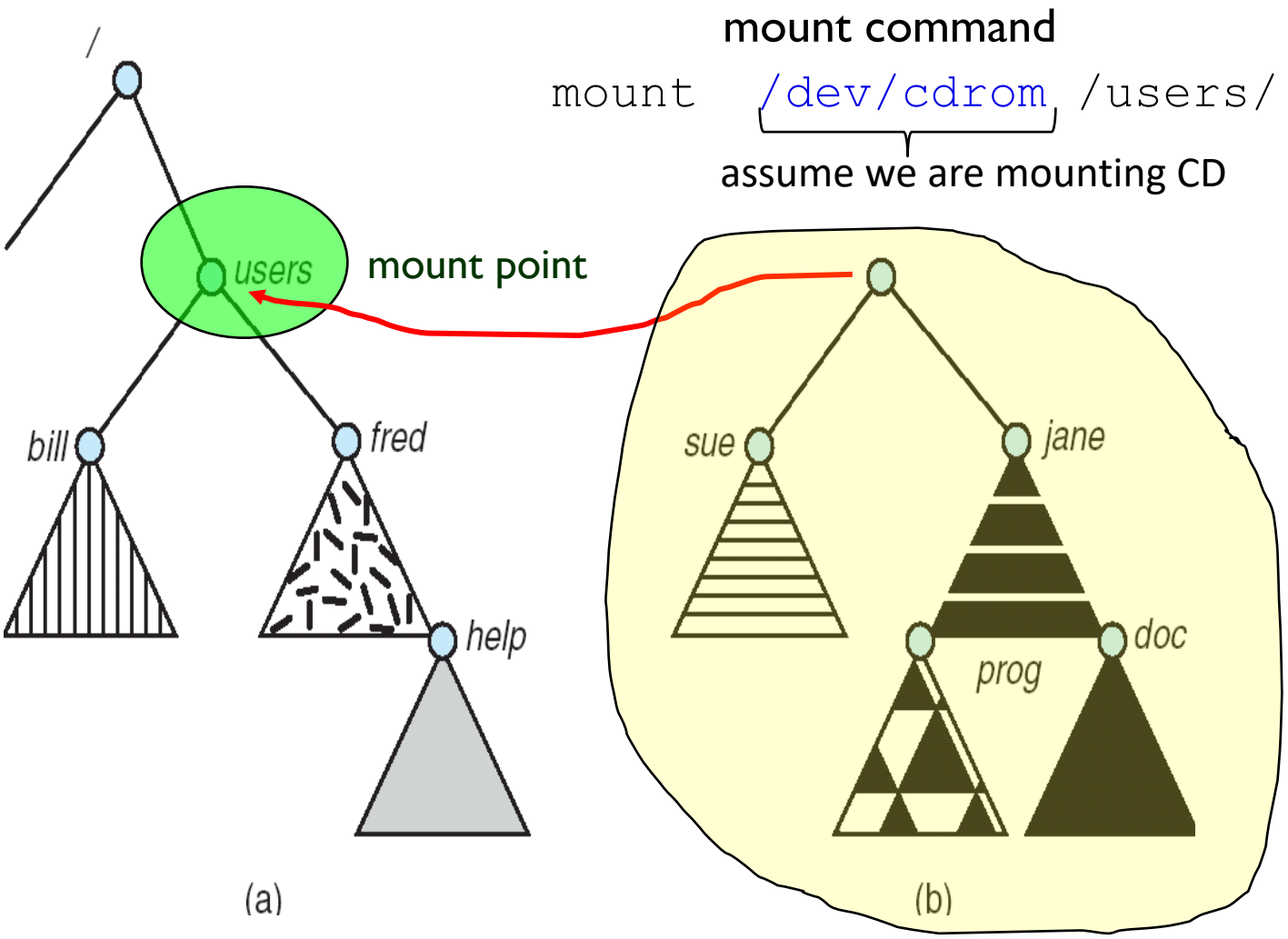
File System Mounting

- A file system must be mounted before it can be accessed
- Mounting a *new file system* means attaching (connecting) the new file system into a location in the *local directory tree* (local file system. i.e., default file system).
- mount point: the place (i.e., usually an empty directory) in the local directory tree where the new file system is attached.
- The root of the mounted file system appears in the *mount point of local file system*.
- We can also mount a sub-directory (sub-tree) in a filesystem.

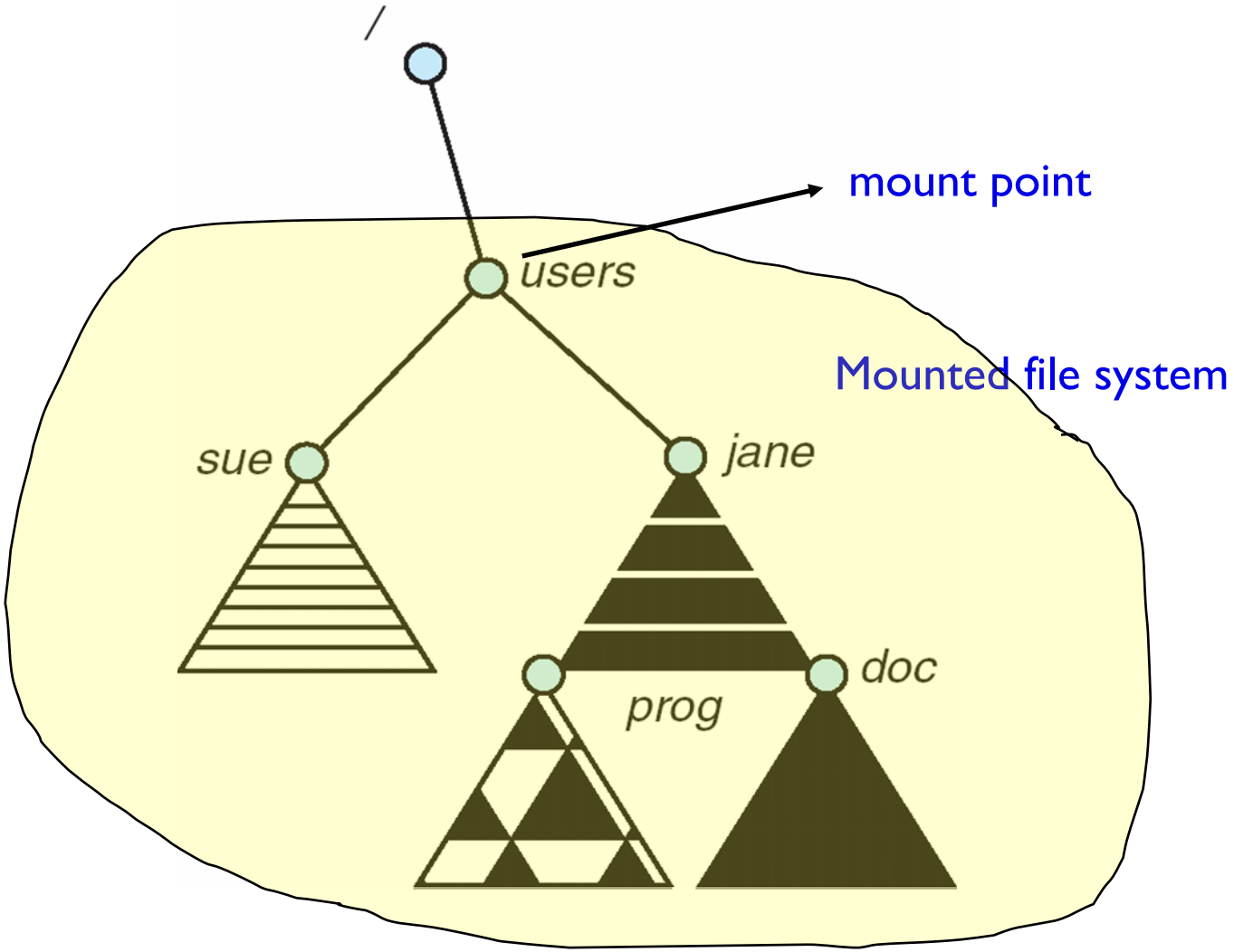
(a) Existing. (b) Unmounted Partition



Mount Point

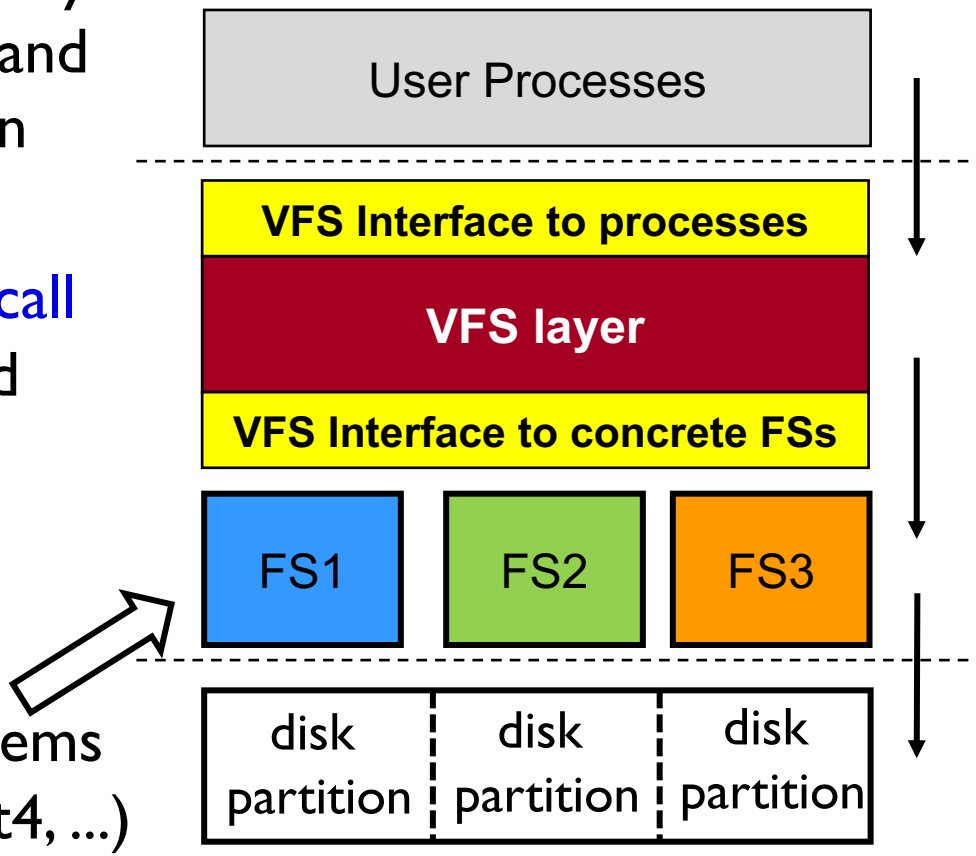

mount command
mount /dev/cdrom /users/
assume we are mounting CD
/
users
mount point
bill
fred
help
sue
jane
prog
doc
(a)
(b)

Local file system appearance after mounting



Virtual File System (VFS)

- Virtual File System (VFS) provides an object-oriented way of implementing file systems and integrating them easily into an OS.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- It is a glue.



Concrete file systems (NTFS, FAT32, ext4, ...)

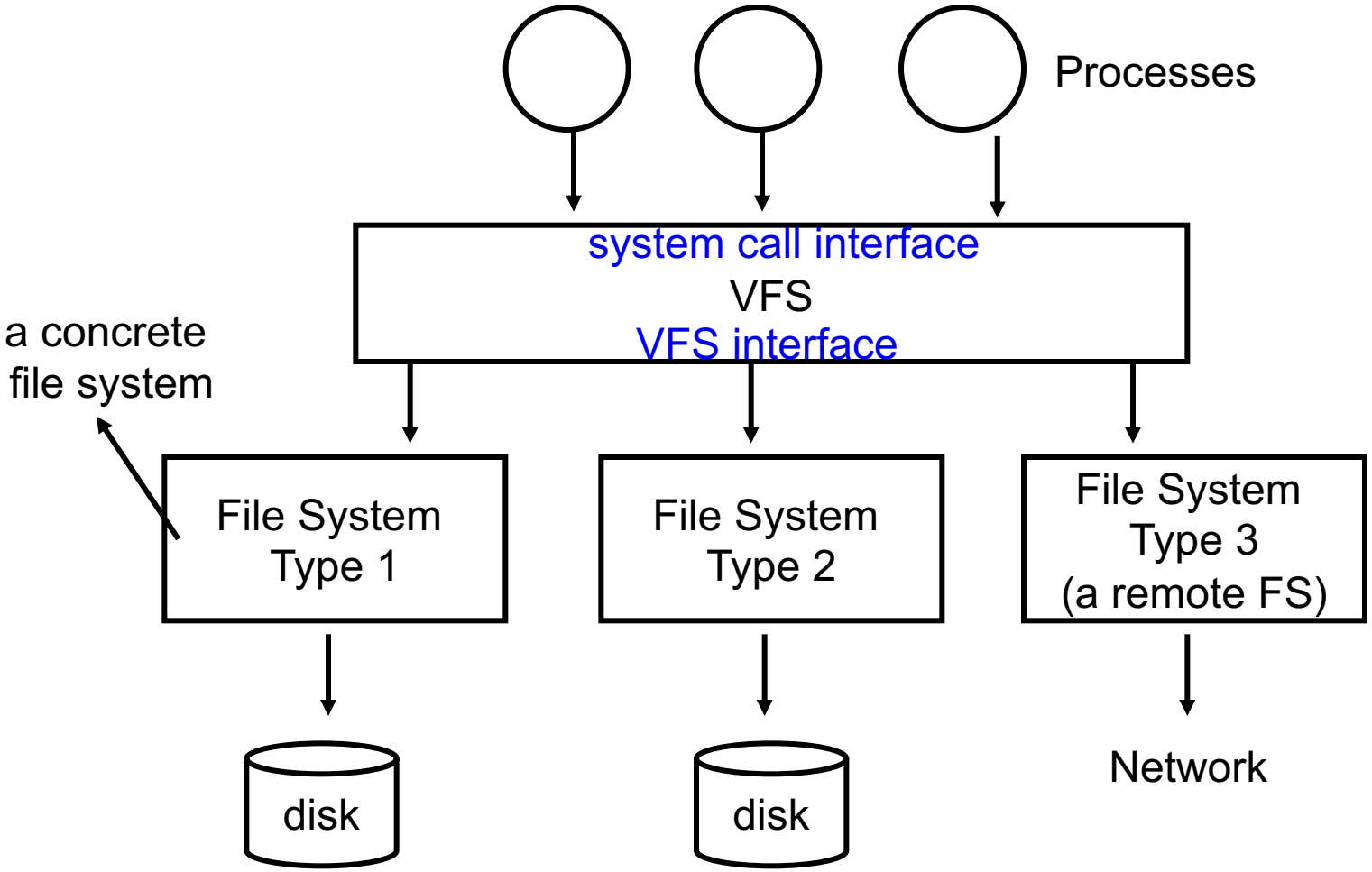
VFS

- VFS defines a common file model that is applicable for all files systems. In the model there are four generic objects:
  - 1) superblock objects, 2) inode (or vnode) objects, 3) file objects, 4) dentry objects.
  - inode keeps info a about a file (attributes, data location, etc.)
- For each object, there are operations defined by VFS.
  - Each disk-based FS should implement these operations.
- VFS separates file-system generic operations from implementation details.
- VFS implements some common operations as well (without triggering the underlying file system operations).

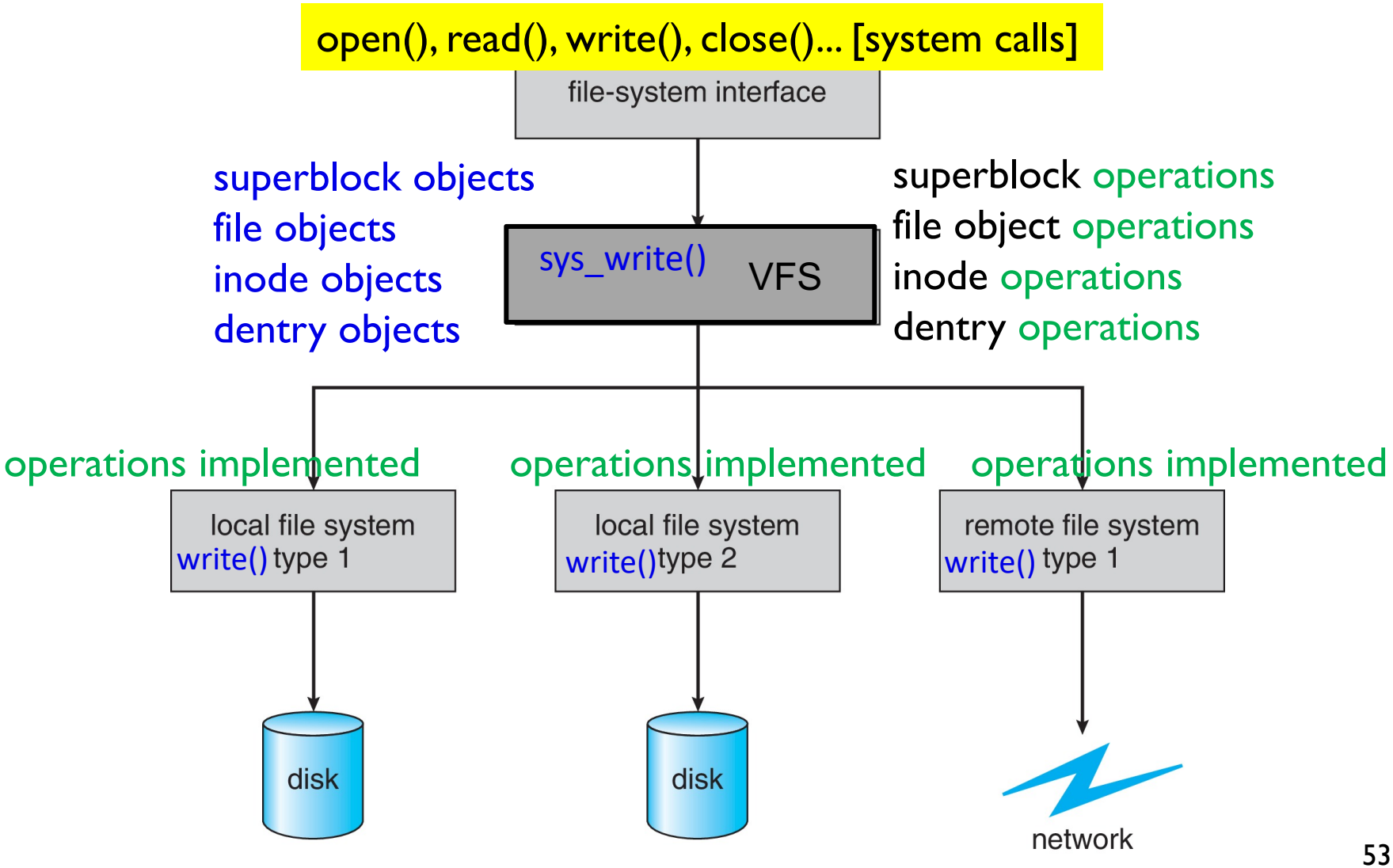
VFS

- Therefore, VFS has also an interface to file systems (concrete file systems, i.e., storage-based file systems)
  - This is called VFS interface to local (e.g., NTFS) or remote file systems (e.g., NFS)
  - This interface defines the functions that a FS must implement.
- A concrete file system (FS) should provide the implementation of these functions defined in VFS interface (functions that VFS expects and that VFS can call).
- VFS dispatches an operation (like open, read) to the appropriate file system implementation routine.

VFS


Processes
system call interface
VFS
VFS interface
a concrete
file system
File System
Type 1
File System
Type 2
File System
Type 3
(a remote FS)
disk
disk
Network

VFS


open(), read(), write(), close()... [system calls]
file-system interface
superblock objects
file objects
inode objects
dentry objects
sys_write()
VFS
superblock operations
file object operations
inode operations
dentry operations
operations implemented
operations implemented
operations implemented
local file system
write() type 1
local file system
write()type 2
remote file system
write() type 1
disk
disk
network

VFS

- For example, in Linux VFS, we have four object types:
  - inode object, file object, superblock object, dentry object
- VFS defines set of operations on these objects that must be implemented by a concrete filesystem.
- Every object has a pointer to a function table
- Function table has a set of function pointers (for operations that can be performed on that object).
  - These pointers are set to point to the respective implementations of the operations in the concrete file system.

File object in Linux VFS (file object represents an opened file)

defined in /include/linux/fs.h

```
struct file { // represents an open file
        union {
            struct list_head    fu_list;        /* list of file objects */
            struct rcu_head     fu_rcuhead;     /* RCU list after freeing */
        } f_u;
        struct path             f_path;         /* contains the dentry */
        struct file_operations  *f_op;          /* file operations table */
        spinlock_t              f_lock;         /* per-file struct lock */
        atomic_t                f_count;        /* file object's usage count */
        unsigned int            f_flags;        /* flags specified on open */
        mode_t                  f_mode;         /* file access mode */
        loff_t                  f_pos;          /* file offset (file pointer) */
        struct fown_struct      f_owner;        /* owner data for signals */
        const struct cred       *f_cred;        /* file credentials */
        struct file_ra_state    f_ra;           /* read-ahead state */
        u64                     f_version;      /* version number */
        void                    *f_security;    /* security module */
        void                    *private_data;  /* tty driver hook */
    .....
    }
```

File object operations structure in Linux

defined in /include/linux/fs.h

each field is a pointer to a function of the specified prototype.

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                             unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                              unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int,
                      unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
    .....
    }
```

a concrete file system implements some or all of these functions.

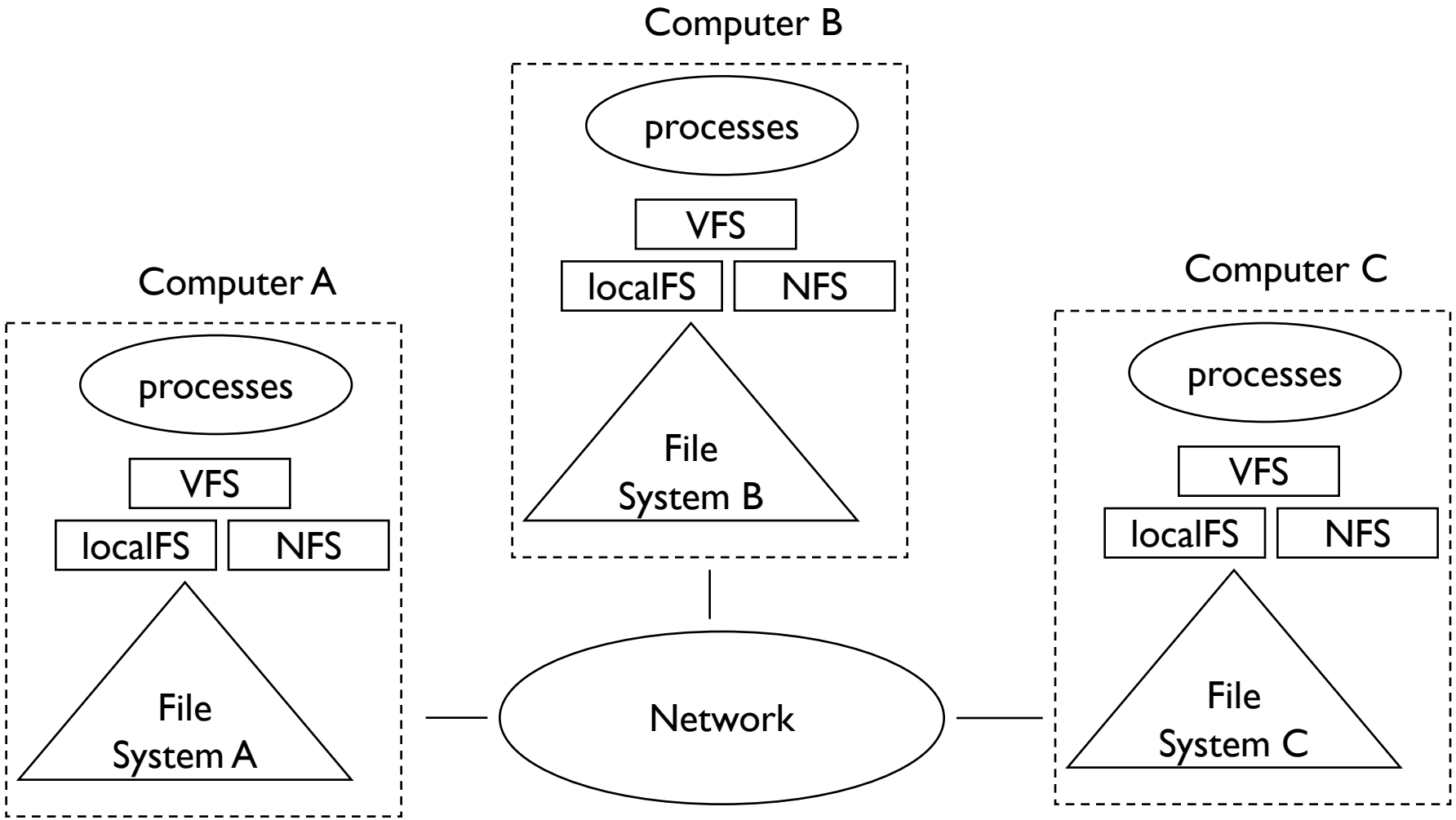
File Sharing – Distributed File Systems (DFS)

- A distributed file system (DFS) allows sharing files among a set of computers connected via a network.
- A computer (user or program) can access remote files seamlessly as if the files were local.
  - open, read, write, close.
- DFS has client-server architecture.
  - A client machine can mount a remote file system in a server machine.
  - Server can serve multiple clients.
  - Client can mount multiple servers.
  - Client and user identification can be complicated. Therefore distributed information (name) systems can be used.

DFS examples

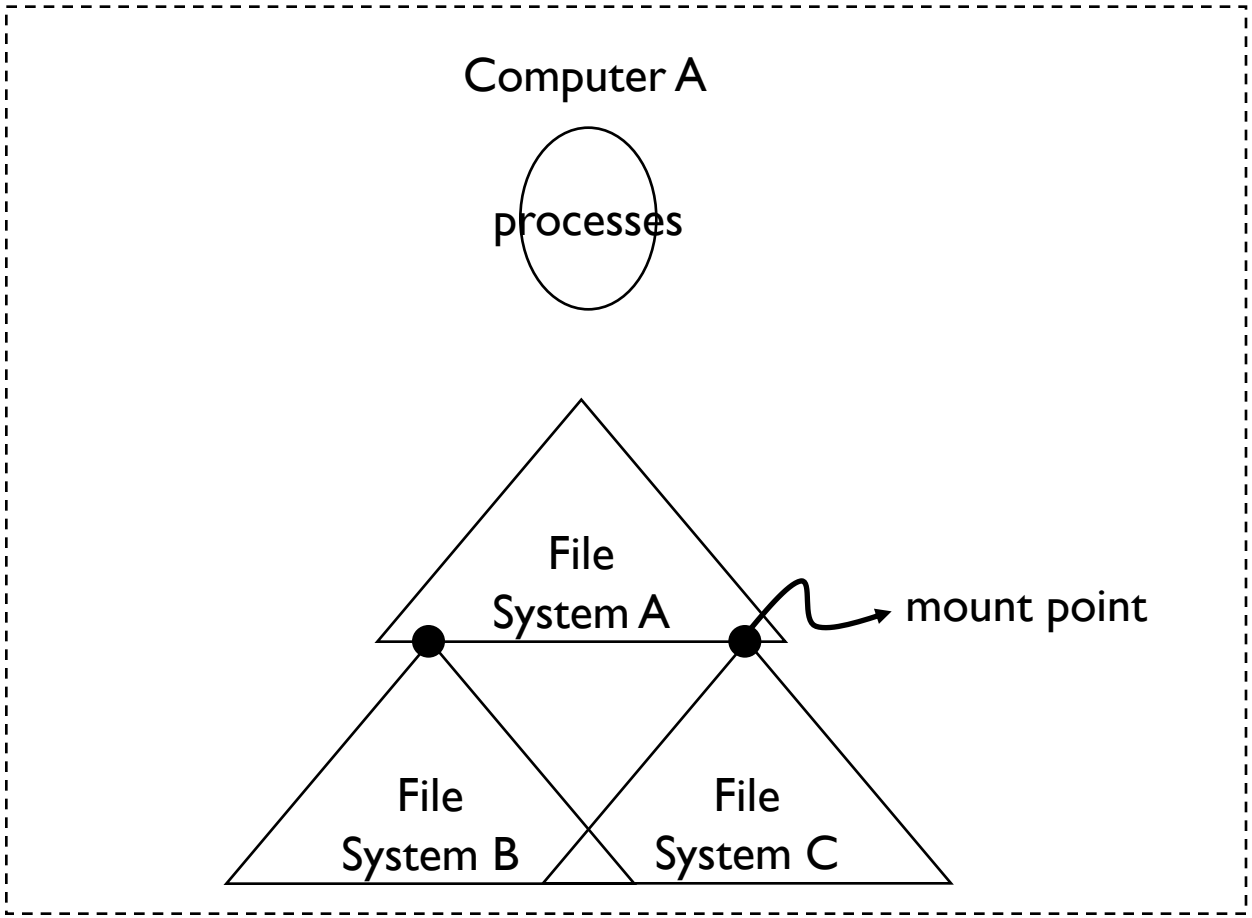
- NFS is standard UNIX client-server file sharing protocol
- CIFS is standard Windows protocol
- Uses RPC: standard operating system file calls are translated into remote procedure calls (RPC)
- Name server examples (to have a common name, id space throughout the network and machines involved). No ambiguity in user names and/or user IDs in this way.
  - LDAP: lightweight directory access protocol.
  - NIS.
  - Active Directory.
  - DNS.

Distributed File System



Distributed File System

File System view at computer A after remote file systems are mounted



File Sharing – Failure Modes

- Distributed file systems add new failure modes, due to network failure, server failure, etc.
- How to recover from failures?
- Recovery from failure can involve state information (to be maintained at server) about status of each remote request.
  - State information: what files are opened by a client; what is the file position pointer, etc.
- Some DFSs do not keep state information (stateless) (NFS v3).

Additional (optional) Study Material

LDAP

- LDAP provides user authentication and other information services for an organization.
- Is a secure distributed naming mechanism.
- Enables a person or an application (process) to locate data about individuals and other resources such as files and devices in a network.
- Can be used in public Internet or in a corporate Intranet.
- Provides a central place for usernames, passwords, resources (like printer information), in an organization.
- An application level network protocol.
- Uses TCP or UDP over IP.
- There is an LDAP server (or even servers).

The Network File System (NFS)

- An implementation and a specification of a software system for accessing remote file systems, directories and files across LANs (or WANs).
- The implementation originally part of SunOS operating system, but now it is industry standard; very commonly used.
- Can use unreliable datagram protocol (UDP/IP) or TCP/IP, over Ethernet or other network to transport RPC calls related messages.

NFS

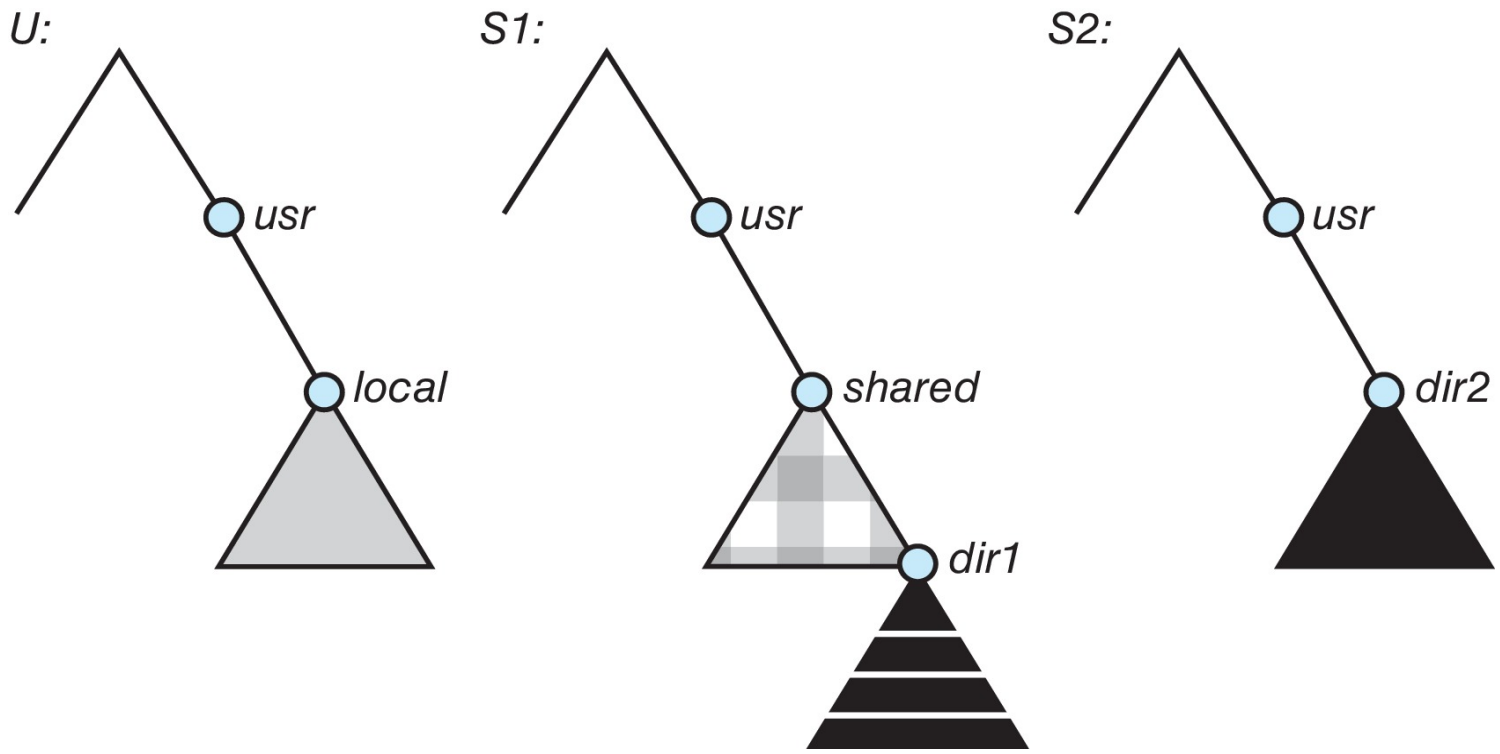
- Interconnected computers are viewed as independent machines having independent file systems.
- Allows sharing of these file systems in a transparent manner
- A remote directory is mounted over a local file system directory:
  - the mounted directory looks like an integral subtree of the local file system.
- The mount operation requires the name of the remote machine and the remote directory specification.
- Assuming client has necessary access-rights, potentially any remote file system (or directory within a file system) can be mounted on top of any local directory.

NFS

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures.
- NFS specification is independent of these.
- 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 has two parts:
  - a mount mechanism and its services.
  - actual remote-file-access services (called NFS protocol)

NFS

- Three independent file systems in 3 different machines.



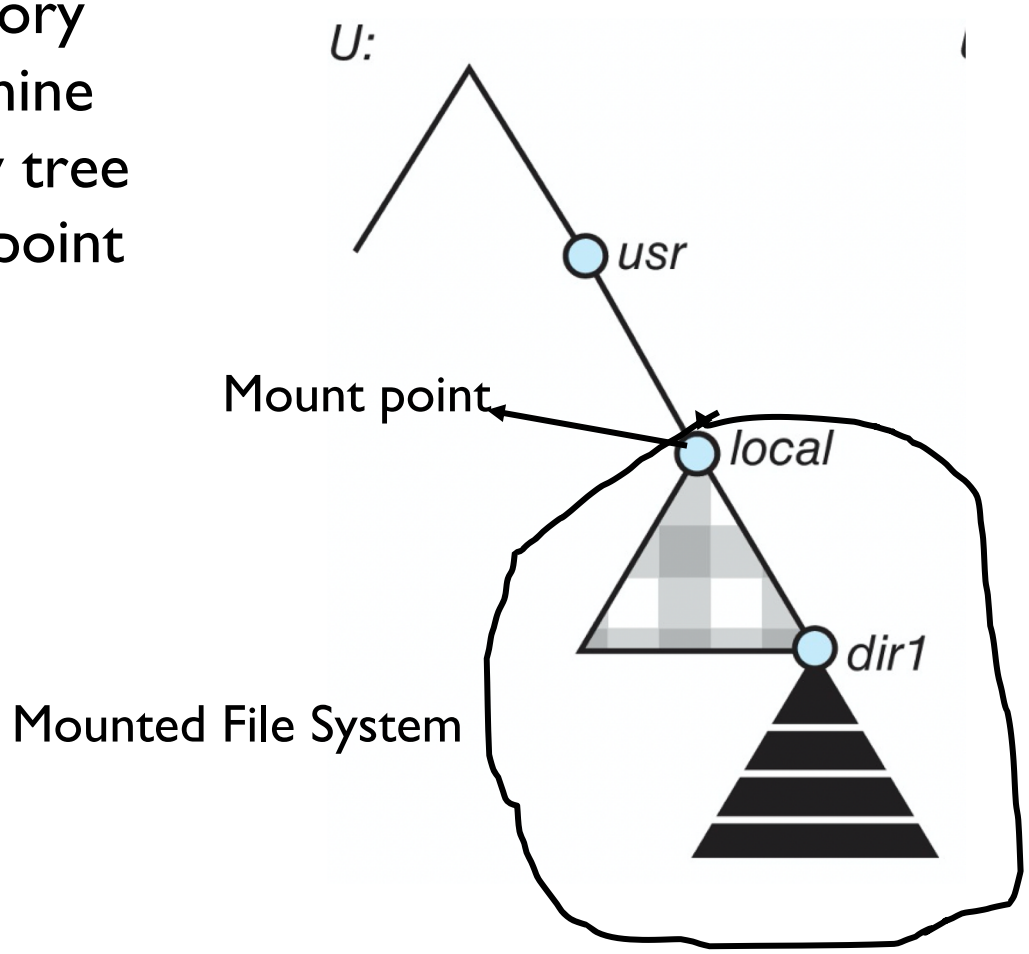
NFS mounting example

- Mounting remote directory /usr/local/shared in machine S1 to the local directory tree of machine U at mount point /usr/local

- S1:/user/local/shared

  over

  U:/usr/local



NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
- Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
- Export list – specifies local file systems that server exports for mounting, along with names of machines permitted to mount.
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses/.
- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- Mounting changes only the user's view (no effect on server)

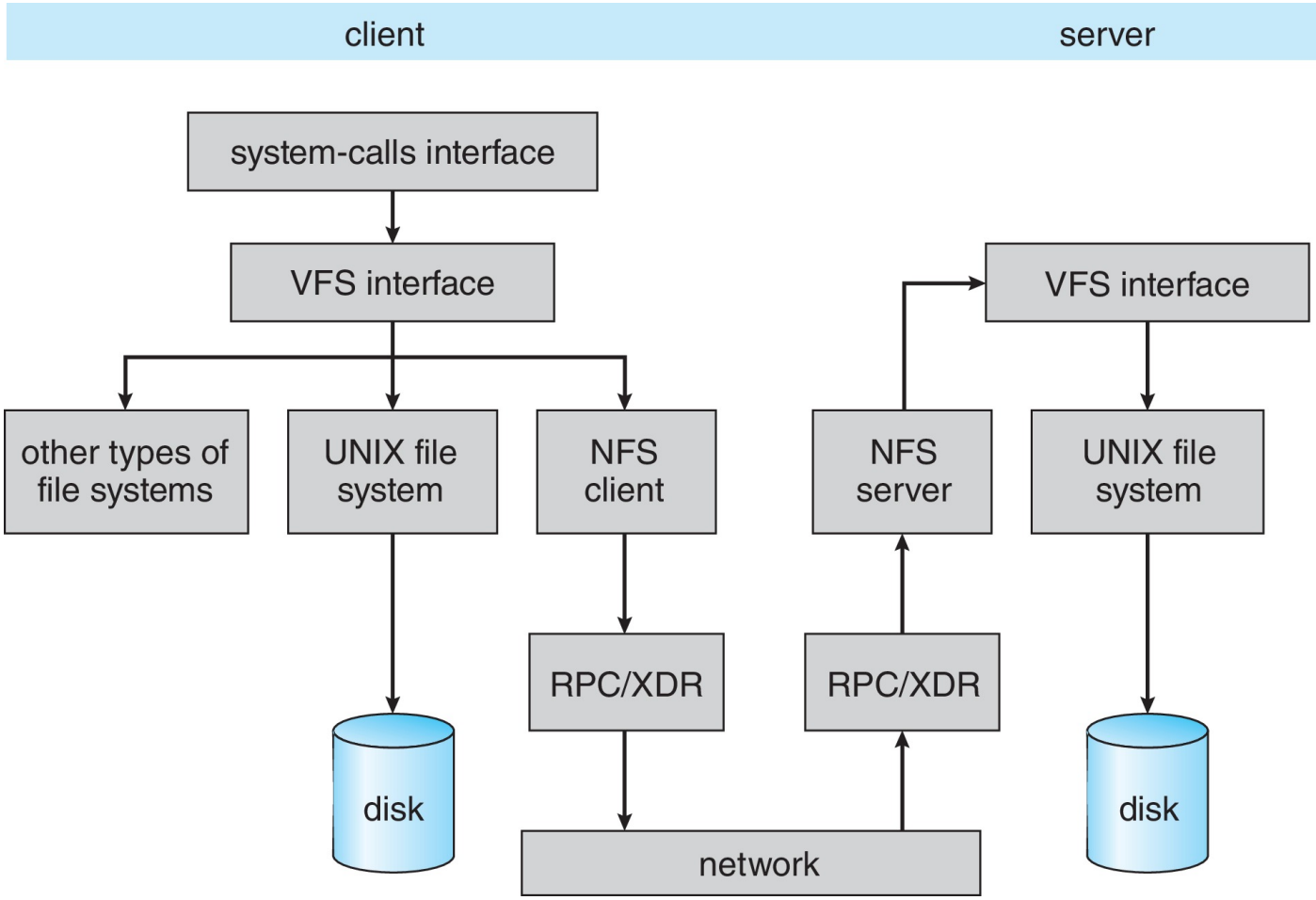
Remote File Access: NFS Protocol

- 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
- NFSv3 servers are stateless; each request has to provide a full set of arguments
- NFSv4 is stateful. It is newer, less used – very different.
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- NFS 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



NFS Pathname Translation

- Performed by breaking the pathname into component names and performing a separate NFS lookup call for every pair of component name and directory vnode.

- To make lookup faster, a directory name lookup cache on the clients side holds the vnodes for remote directory names.

NFS Remote Operations

- Nearly one-to-one correspondence between UNIX system calls and NFS protocol RPCs (except opening and closing files)
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance
- File-blocks cache – when a file is opened, kernel checks with the remote server whether to fetch/revalidate cached attributes.
- Cached file blocks are used only if the corresponding cached attributes are up to date.
- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server.
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk.

References

- Operating System Concepts, Silberschatz et al.
- Modern Operating Systems, Andrew S. Tanenbaum et al.
- Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau et al.
- Linux Kernel Development, Robert Love.