Operating Systems 2017/2018 | Part VI – File System

Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

Slides based on the book

'Operating System Concepts, 9th Edition,

Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Wiley'

Chapters 10 and 11

File System Types

- Many file systems are in use today, and most operating systems support more than one file system
  - UNIX uses the **UNIX file system (UFS)** which is based on the Berkeley Fast File System (FFS)
  - The standard Linux file system is the **extended file system** (most common versions being **ext3** and **ext4**)
  - Windows uses the **FAT**, **FAT32** and **NTFS** file systems
  - CD-ROMs are written in the **ISO 9660** format
- New ones are still arriving since file system continues to be an active research area
  - **GoogleFS**
  - **Oracle ASM**
  - **FUSE**

File System Concept

- For most users, the file system is the most visible aspect of an operating system as it provides the mechanism for **storage and access to both data and programs** of the operating system
- A file system consists of two distinct parts:
  - A **collection of files**, each file defines a logical storage unit with related data
  - A **directory structure**, which organizes and provides information about all the files in the system
- The operating system **abstracts the physical properties of its storage devices** to organize the directory structure and its files
  - These storage devices are usually nonvolatile, so the contents are persistent between system reboots

File Attributes

- A file is defined by a set of attributes that vary from one operating system to another. Typical attributes are:
  - **Name** – for the convenience of the human users
  - **Type** – to indicate the type of operations that can be done on the file (included in the name as an extension, usually separated by a period)
  - **Location** – pointer to file location on device
  - **Size** – current file size
  - **Protection** – controls who can do reading, writing, executing, and so on
  - **Time, date, and user identification** – data useful for protection, security, and usage monitoring

Common File 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 Sharing and Protection

- When an operating system accommodates multiple users, it must **mediate how file sharing and file protection is done**
  - The system can either allow a user to access the files of other users by default or require that a user specifically grant access to its files

- Typical access rights are:
  - **Read** – read from the file
  - **Write** – write or rewrite the file
  - **Execute** – load the file into memory and execute it
  - **Append** – write new information at the end of the file
  - **Delete** – delete the file and free its space for possible reuse
  - **List** – list the name and attributes of the file

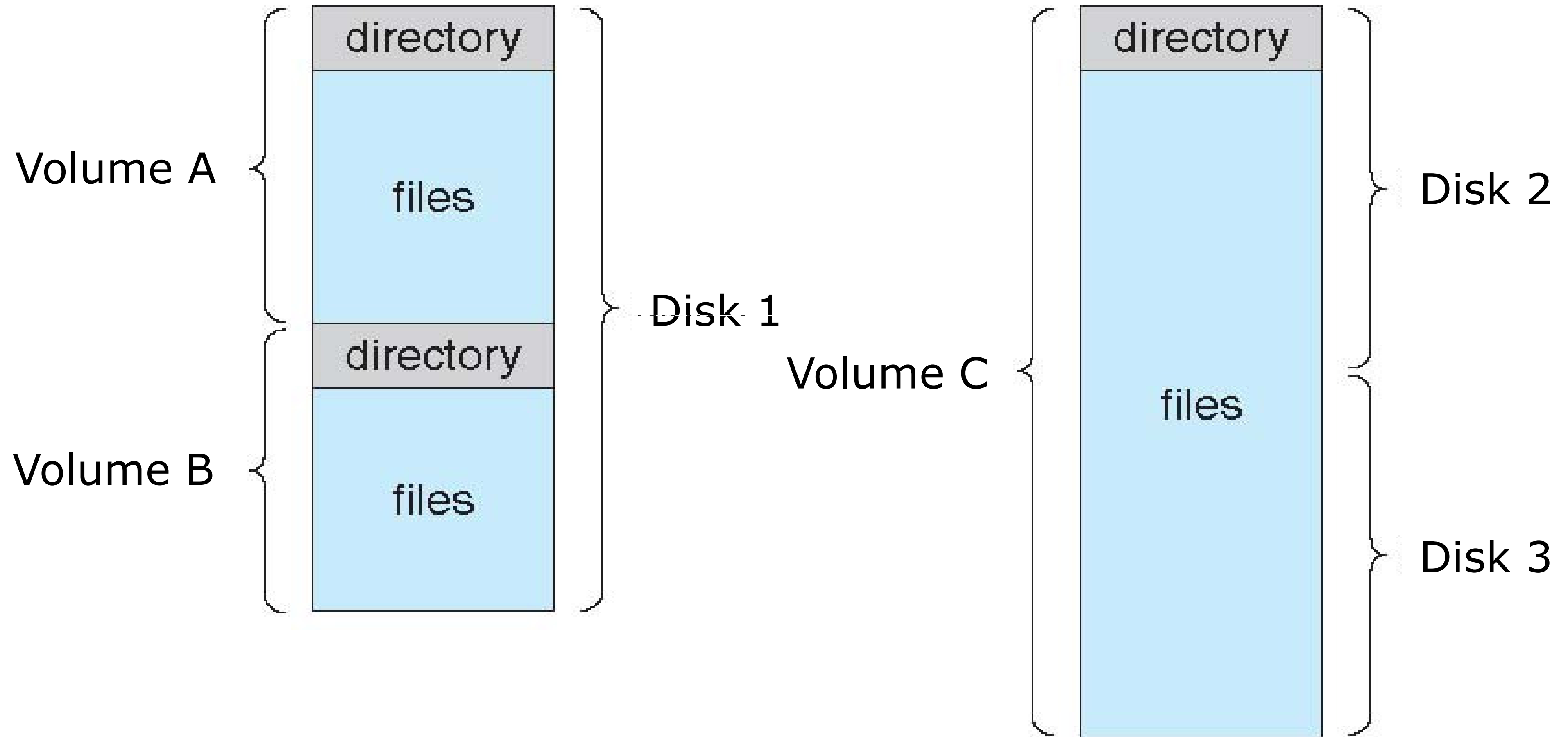
File System Organization

- A storage device can be used entirety for a file system or subdivided into partitions. Partitioning allows for:
  - Multiple file-system types to be on the same device
  - Specific needs where no file system is appropriate (**raw partition**), such as for swap space or databases that want to format data according to their needs
- A entity containing a file system is generally known as a **volume**
  - A volume may be a whole device, a subset of a device, or multiple devices linked together
- The information about the file system in a volume is kept in the:
  - **Boot control bock**
  - **Volume control block**
  - **Directory structure**

File System Organization

- **Boot control block**
  - Typically the first block of a volume which contains the info needed to boot an OS from that volume (can be empty if volume does not contain an OS)
  - Also called **boot block** in UFS and **partition boot sector** in NTFS
- **Volume control block**
  - Contains volume details, such as total number of blocks, total number of free blocks, block size, free block pointers, …
  - Also called **superblock** in UFS and is part of the **master file table** in NTFS
- **Directory structure**
  - Used to organize the directory and files
  - Part of the **master file table** in NTFS

File System Organization

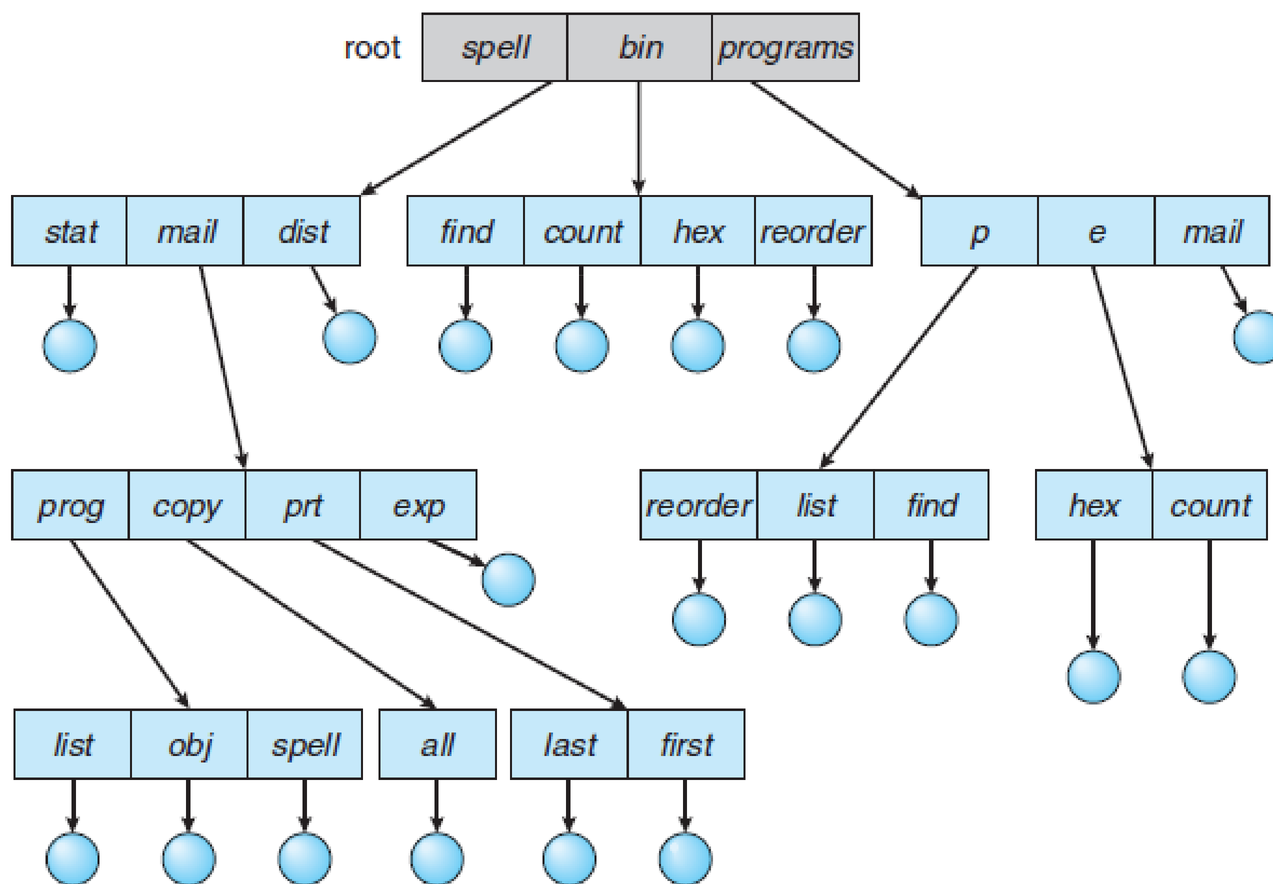


Directory Overview

- The directory structure can be viewed as a symbol table that **translates file names into directory entries**
- The directory itself can be organized in many different ways, but must:
  - Allow to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory
  - Be efficient (locate a file quickly)
  - Be convenient to users (e.g., allow two users to have same name for different files or allow grouping files by some characteristic)

Tree-Structured Directories

- Tree of arbitrary height with a unique root directory and with every file with a unique path name

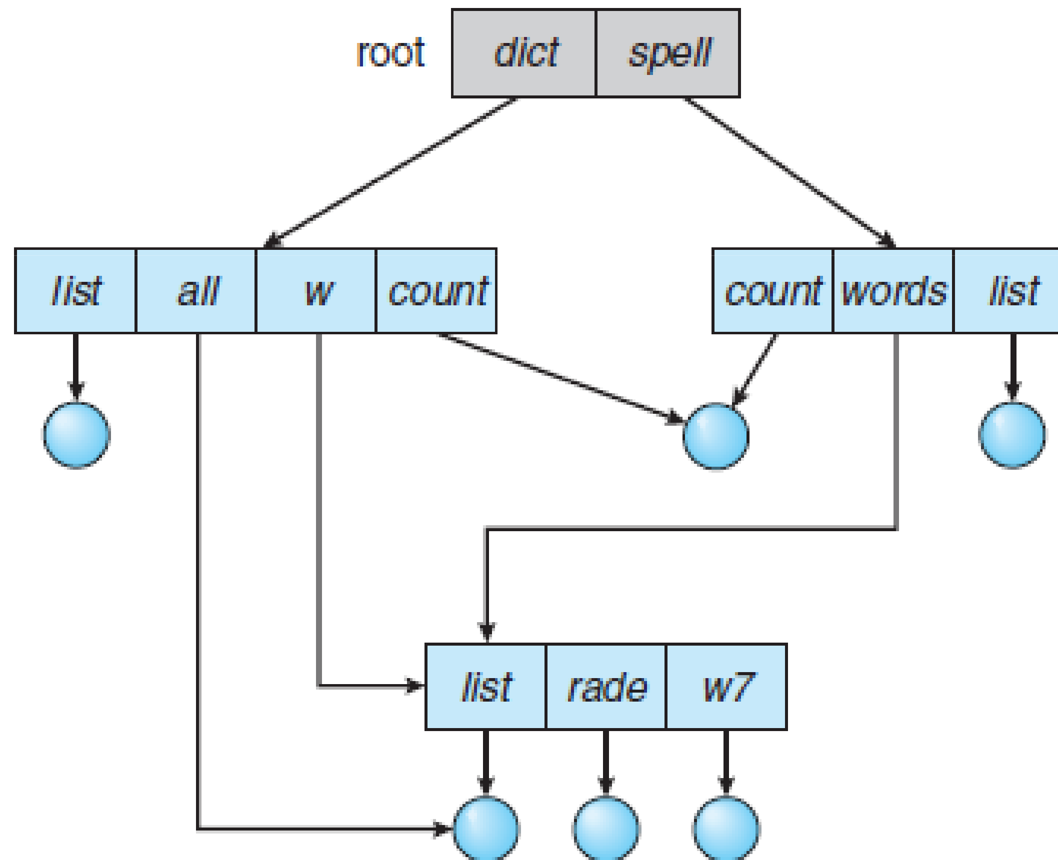


Tree-Structured Directories

- A directory contains a **set of files and/or subdirectories**
  - A subdirectory can be seen as a common file that is treated in a special way
- Each process has a **current (or working) directory**
  - When a reference to a file is made, the current directory is searched
- Path names can be of two types:
  - **Absolute path name** – begins at the root and follows a path down to the specified file, giving the directory names on the path (e.g., /spell/mail/exp)
  - **Relative path name** – defines a path starting from the current directory (e.g., mail/exp if the current directory is /spell)

Acyclic-Graph Directories

- Generalization of the tree-structured directory to allow directories to share files and subdirectories



Acyclic-Graph Directories

- A common way to implement shared files/subdirectories is to use **links (or shortcuts)**
  - A link is effectively a pointer (path name) to another file or subdirectory
  - A link may be implemented as an absolute or a relative path name
  - When a reference to a link is made, we resolve the link by following the pointer to locate the real file/subdirectory
- A file/subdirectory can now have **multiple absolute path names**
  - Can be problematic when traversing the file system to accumulate statistics on all files or to copy all files to backup storage, since we do not want to traverse shared structures more than once

Acyclic-Graph Directories

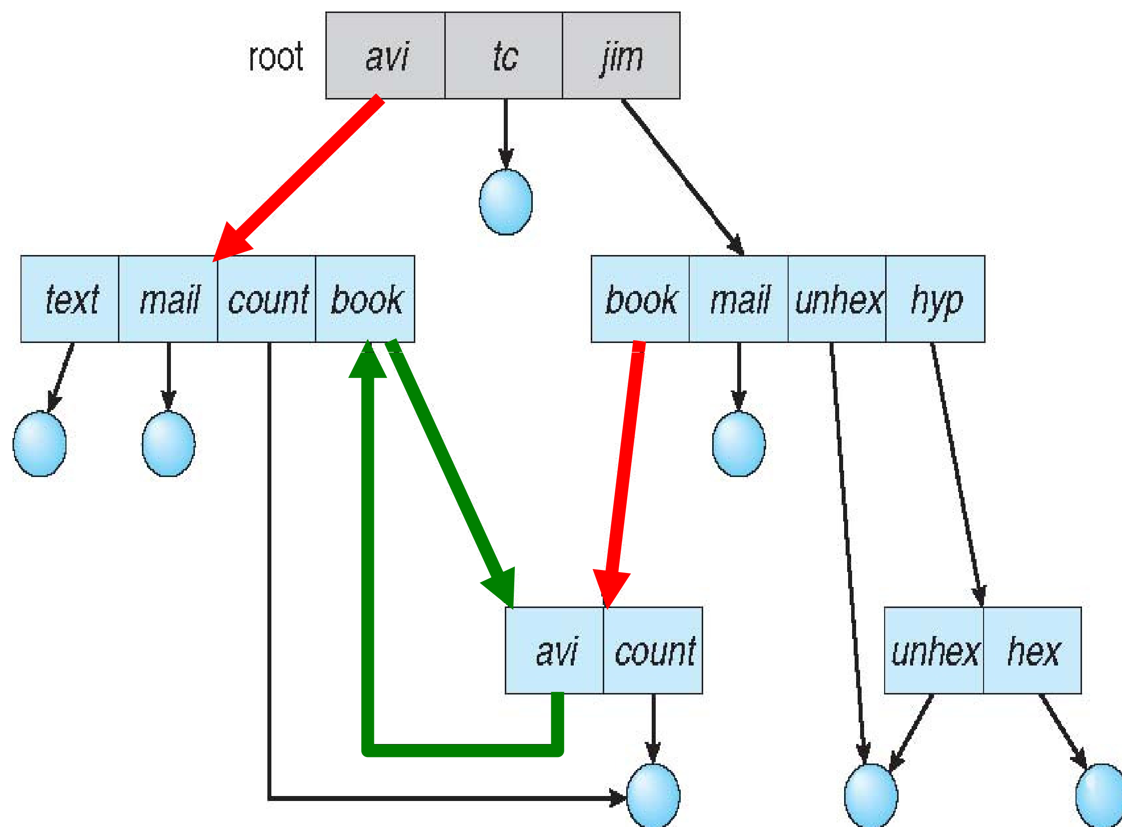
- How to handle the deletion of shared files/subdirectories?
  - It depends if links are **symbolic links** or **hard (nonsymbolic) links**

- Deletion with symbolic links
  - The deletion of a link does not affect the original file/subdirectory (only the link is removed)
  - If original file/subdirector is deleted, the existing links turn invalid and are treated as illegal file name (links become valid if another file/subdirectory with the same name is created)

- Deletion with hard links
  - We keep a counter of the number of references to a shared file/subdirectory
  - Adding/deleting a link increments/decrements the counter
  - When the count reaches 0, the file/subdirectory can be deleted since no remaining references exist

General Graph Directory

- A serious problem with a graph structure is the **creation of cycles**
  - Might be difficult to detect if a file/subdirectory can be deleted
  - The reference count may not be 0 even when it is no longer possible to refer to a file/subdirectory
  - Some **garbage collection mechanism** might be needed to determine when the last reference has been deleted and the disk space can be reallocated
  - Problem can be avoided if we only allow links to files and not to subdirectories



Open Files

- Typically, to manage open files, operating systems use **two levels of internal tables**
  - **System-wide open-file table** – contains process-independent information for all open files
  - **Per-process open-file table** – contains specific info for the files that a process has open
- Most of the operations on files involve searching for the entry associated with the named file
  - To avoid this constant searching, many operating systems require that an **open() system call be made before a file is first used**
  - On success, the open() system call returns an **index to the per-process open-file table** – called **file descriptor** in UNIX systems and **file handle** in Windows systems – for subsequent use

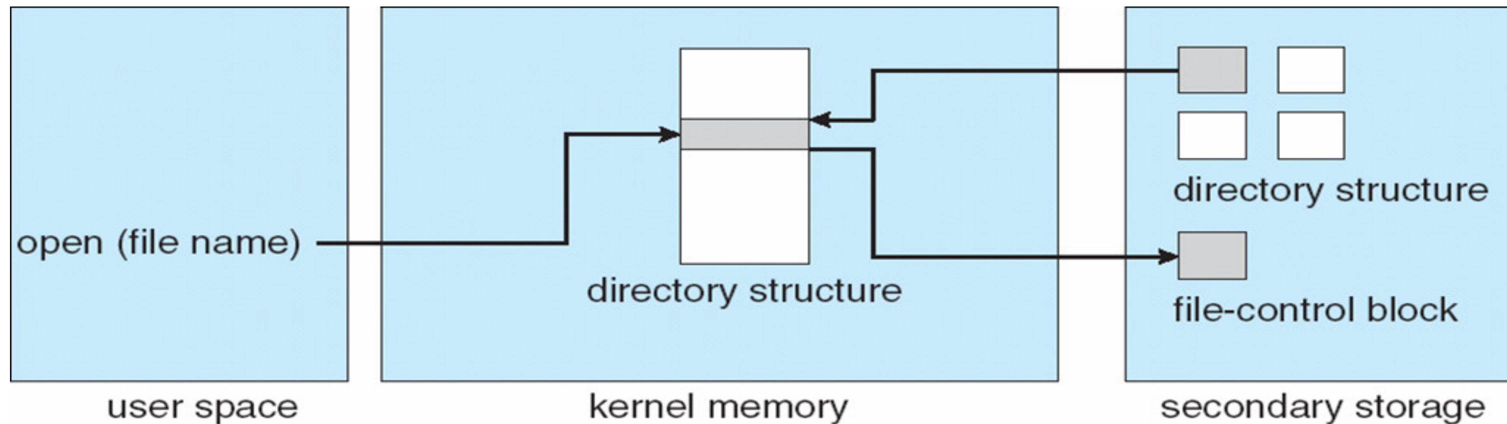
Open Files

- The **system-wide open-file table** stores per open file the following info:
  - **File control block (FCB)** which contains information about the file so that the system does not have to read this information from disk on each operation
  - File-open count to indicate how many processes have the file open – allows removal of FCB when the last process closes the file
- The **per-process open-file table** stores per open file the following info:
  - Reference to the appropriate entry in the system-wide open-file table
  - Access mode in which the file is open
  - File pointer to last read/write location

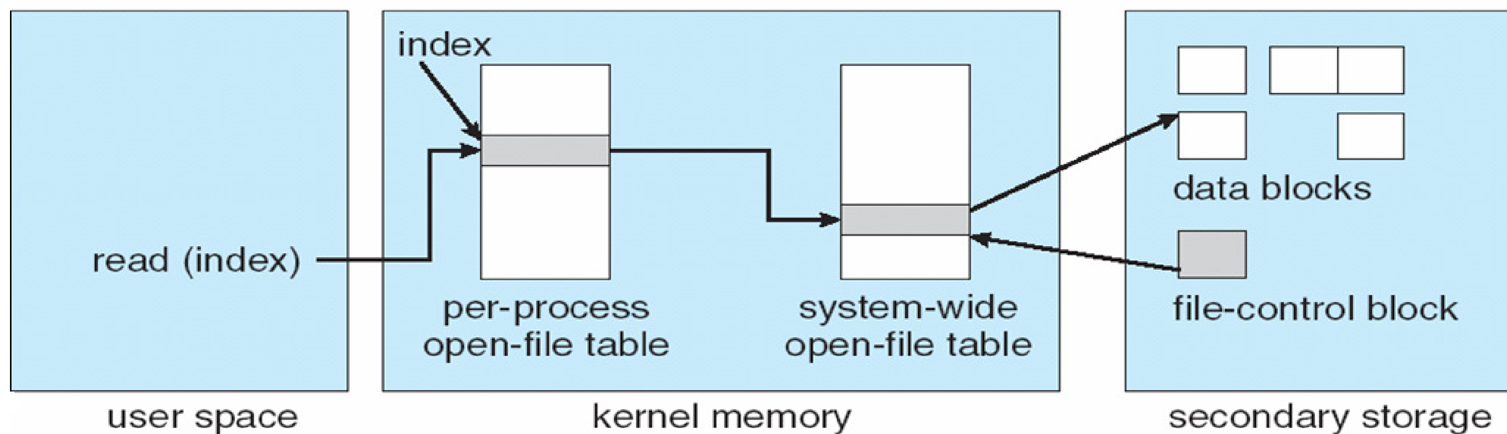
Typical File Control Block

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

In-Memory File System Structures



Opening a file

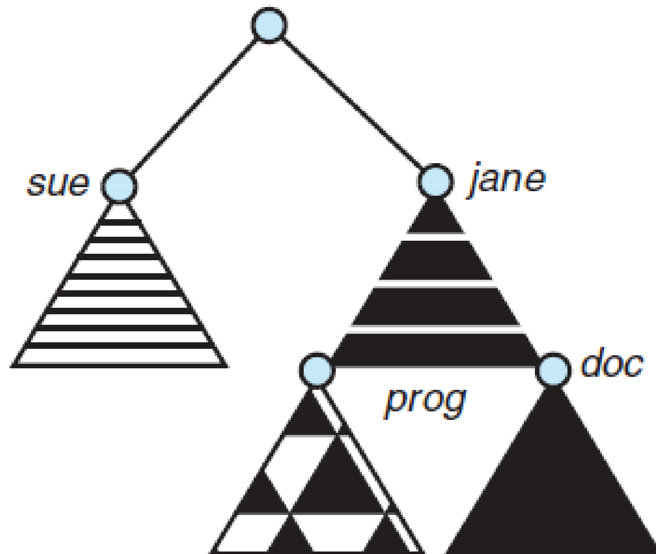


Reading from an open file

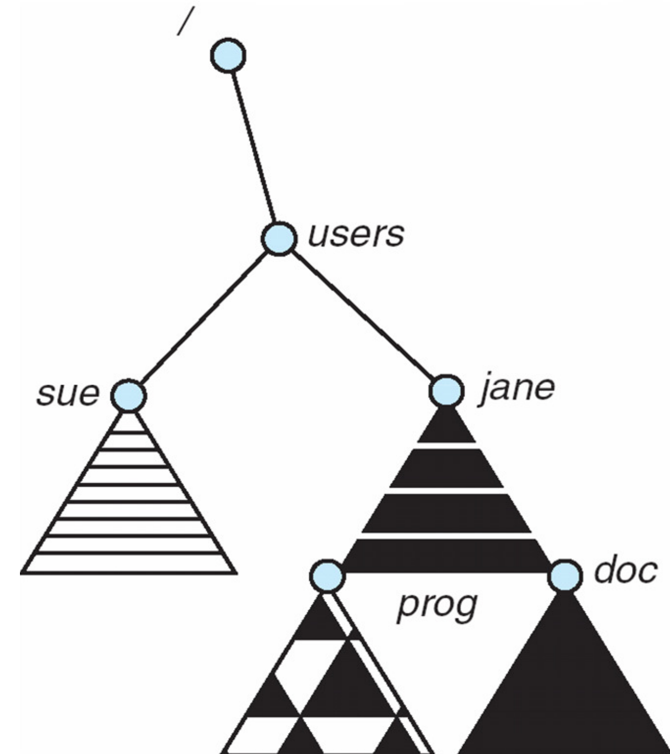
File System Mounting

- Just as a file must be opened before it is used, a **file system must be mounted before it can be accessed**
  - The directory structure may be built out of multiple volumes, which must be mounted to make them available within the file-system name space
  - Volumes can be mounted at boot time or later, either automatically or manually
  - The mount procedure is straightforward, the operating system is given the name of a volume and the **mount point** – the location within the file structure where the new file system is to be attached
  - Typically, a mount point is an empty directory

File System Mounting



Unmounted volume

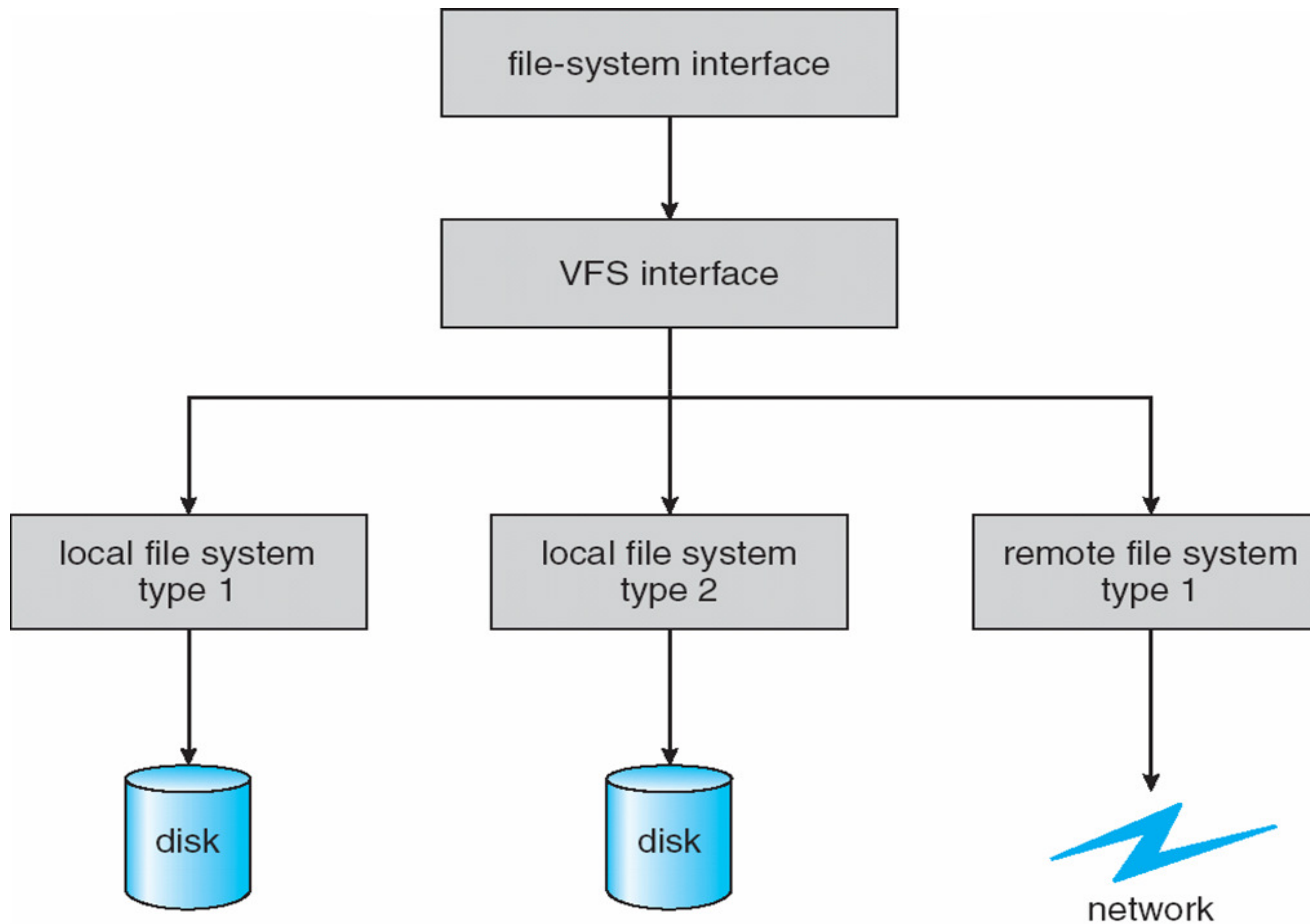


After mounting volume on mount point /users

Virtual File System

- To **simplify and modularize the support for multiple types of file systems**, most modern operating systems use a **virtual file system (VFS) layer**
- The file-system interface, based on the open(), read(), write() and close() calls, rather than interacting with each specific type of file system, interacts only with the VFS layer
  - Allows the same system call interface to be used with different types of file systems
  - Separates the file system interface from the file system implementation details
  - The VFS then dispatches the calls to appropriate file system implementation routines
  - The VFS is based on a unique file-representation structure, called a **vnode**

Virtual File System



Designing the File System

- The direct-access nature of disks gives us flexibility in the implementation of files but the key design problem is **how to allocate files so that disk space is utilized effectively and files can be accessed quickly**

- File access patterns:
  - **Sequential access** – bytes read/write in order (most file accesses are of this flavor)
  - **Random access** – bytes read/write without order (less frequent, but still important – don't want to read all bytes to get to a particular point of the file)

- Usage patterns:
  - Most files are small (for example, .doc, .txt, .c, .java files)
  - A few files are big (for example, executables, swap, core files, …)
  - Large files use most of the disk space and bandwidth to/from disk

Designing the File System

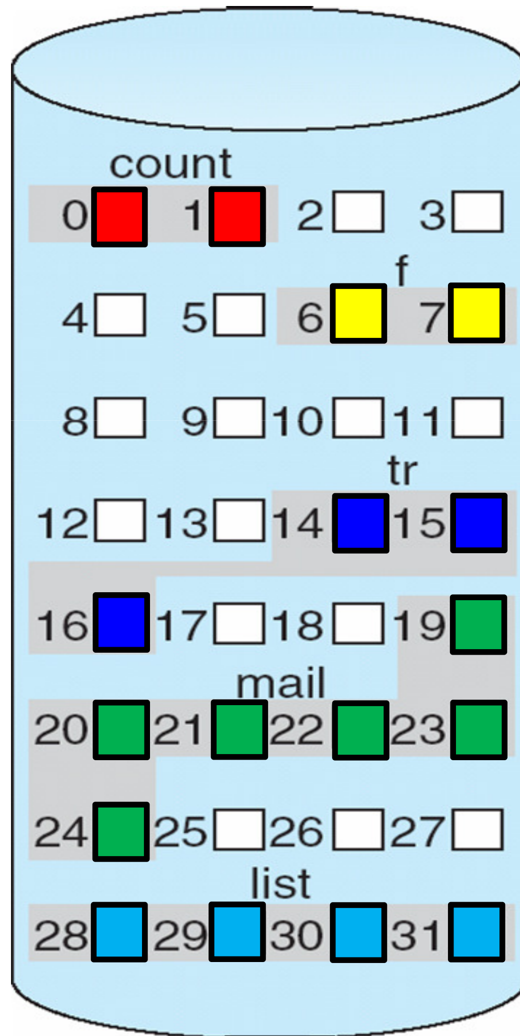
- File system main goals:
  - Maximize sequential access performance
  - Efficient random access to file
  - Easy management of files (growth, truncation, …)

- Three major methods of allocating disk space are in wide use:
  - **Contiguous allocation**
  - **Linked allocation**
  - **Indexed allocation**

Contiguous Allocation

- With contiguous allocation, each file **occupies a set of contiguous blocks** on disk
  - Disk addresses define a linear ordering on the disk
  - Only starting location (block #) and length (number of blocks) are required
- **Sequential access** – we only need to keep a reference R to the last block read since the next one to read is always R+1
- **Random access** – given a block N of a file that starts at block B, we can immediately access block B+N

Contiguous Allocation



directory

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

Contiguous Allocation

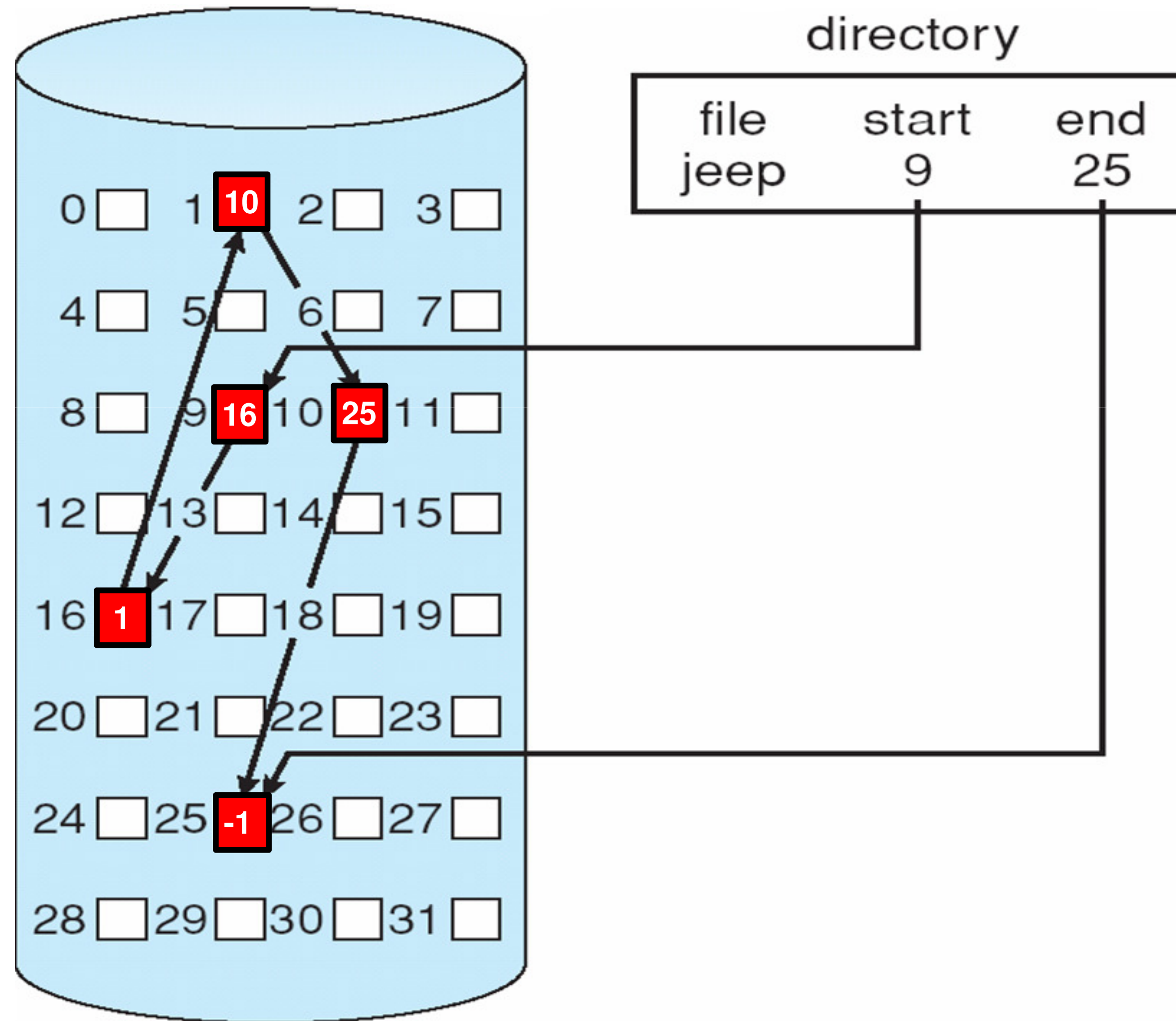
- **Advantages**
  - The number of disk seeks required for accessing contiguously allocated files is minimal (disk head only needs to move from one track to the next when reaching the last sector on a track)

- **Problems**
  - Finding free space (first-fit, best-fit, worst-fit, …)
  - Knowing file size beforehand (might be difficult to know)
  - Hard to grow files (if we allocate too little space to a file, we may not be able to extended it later)
  - External fragmentation (requires regular use of defragmentation techniques to compact all free space into one contiguous space)

Linked Allocation

- With linked allocation, each file is a **linked list of disk blocks** (blocks may be scattered anywhere on the disk)
  - The directory contains a pointer to the first and last blocks of the file
  - Each block contains pointer to next block
  - Last block ends with nil pointer
- **Sequential access** – keep reference to last block read and follow the pointers from block to block
- **Random access** – must follow N pointers until we get to the Nth block

Linked Allocation



Linked Allocation

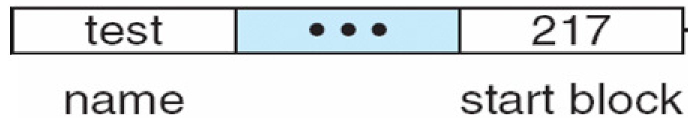
- **Advantages**
  - Easy to grow and no need to know file size beforehand (a file can continue to grow as long as free blocks are available)
  - No external fragmentation (any free block can be used to satisfy a request, consequently, it is never necessary to compact disk space)

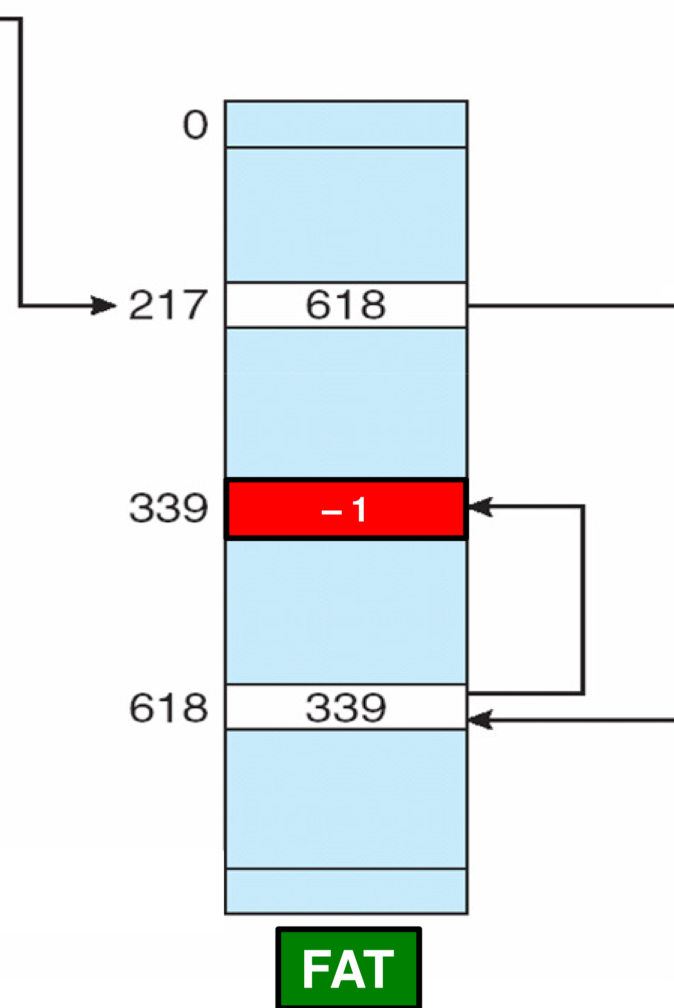
- **Problems**
  - Less efficient sequential access support and very inefficient random access support – each access to a block requires a disk read, and some require a disk seek
  - Requires space for pointers in data blocks (if a pointer requires 4 bytes out of a 512-byte block, then 0.78% of the disk is wasted on pointers)
  - Reliability can be a problem (a bug or disk failure might result in picking up a wrong pointer, which could result in corrupting other parts of the file system)

File-Allocation Table (FAT)

- FAT is an important variation of linked allocation
  - Section at the **beginning of each volume** is set aside to contain the FAT
  - Has **one entry for each disk block** and is indexed by block number
  - **Used as a linked list**, the directory entry stores the first block of a file and then each FAT's block entry contains pointer to next block
- **Advantages**
  - Requires no space for pointers in data blocks
  - Free blocks can also be stored as a linked list
  - Random access more efficient and really efficient if FAT cached in memory
- **Problems**
  - Sequential access still less efficient even if FAT cached in memory

File-Allocation Table (FAT)

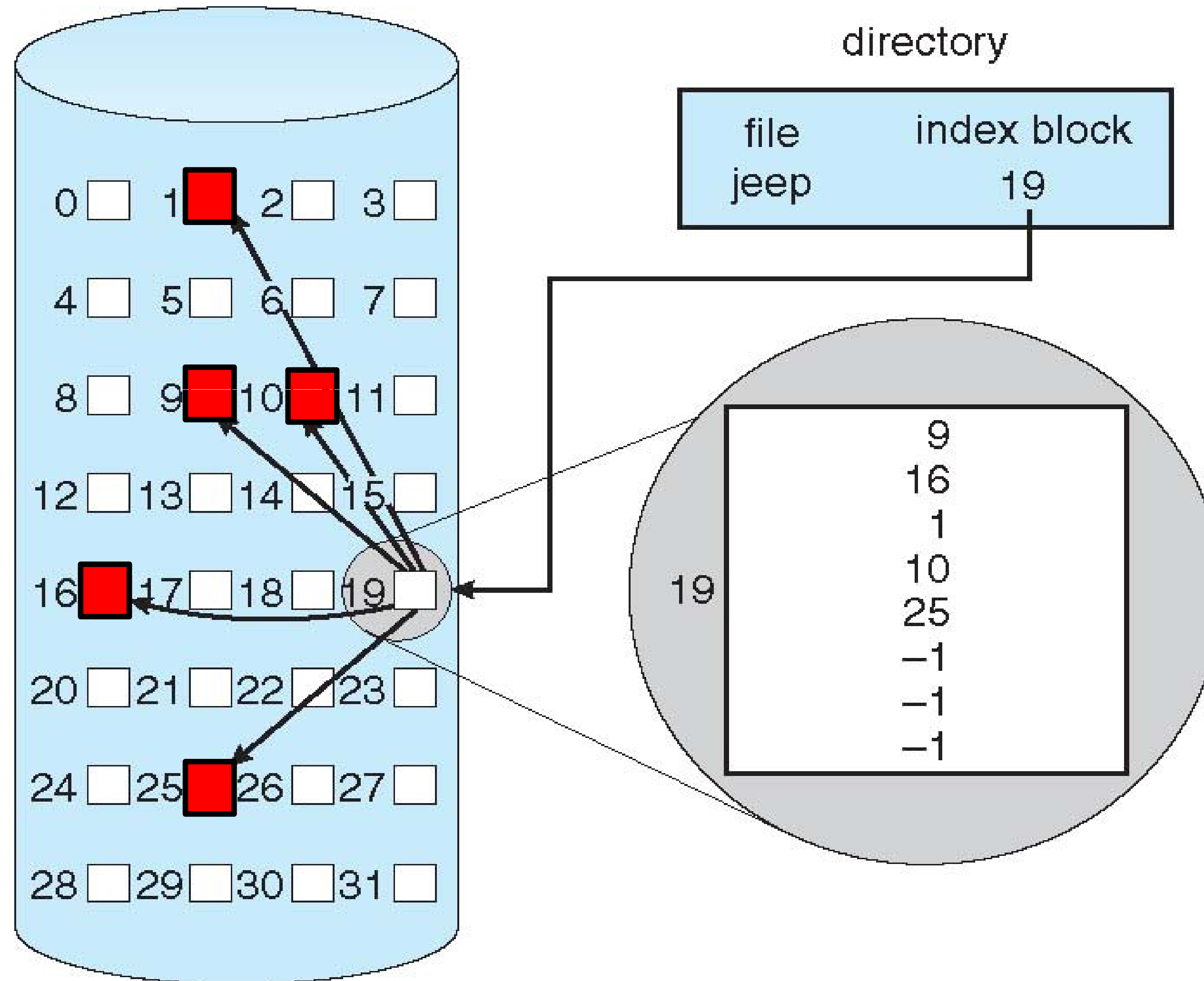




Indexed Allocation

- With indexed allocation, each file has its own **index block, which is an array of disk block pointers**
  - The directory contains the address of the index block
  - The Nth entry in the index block points to the Nth block of the file
- **Sequential/random access** – both require a first access to the index block

Indexed Allocation



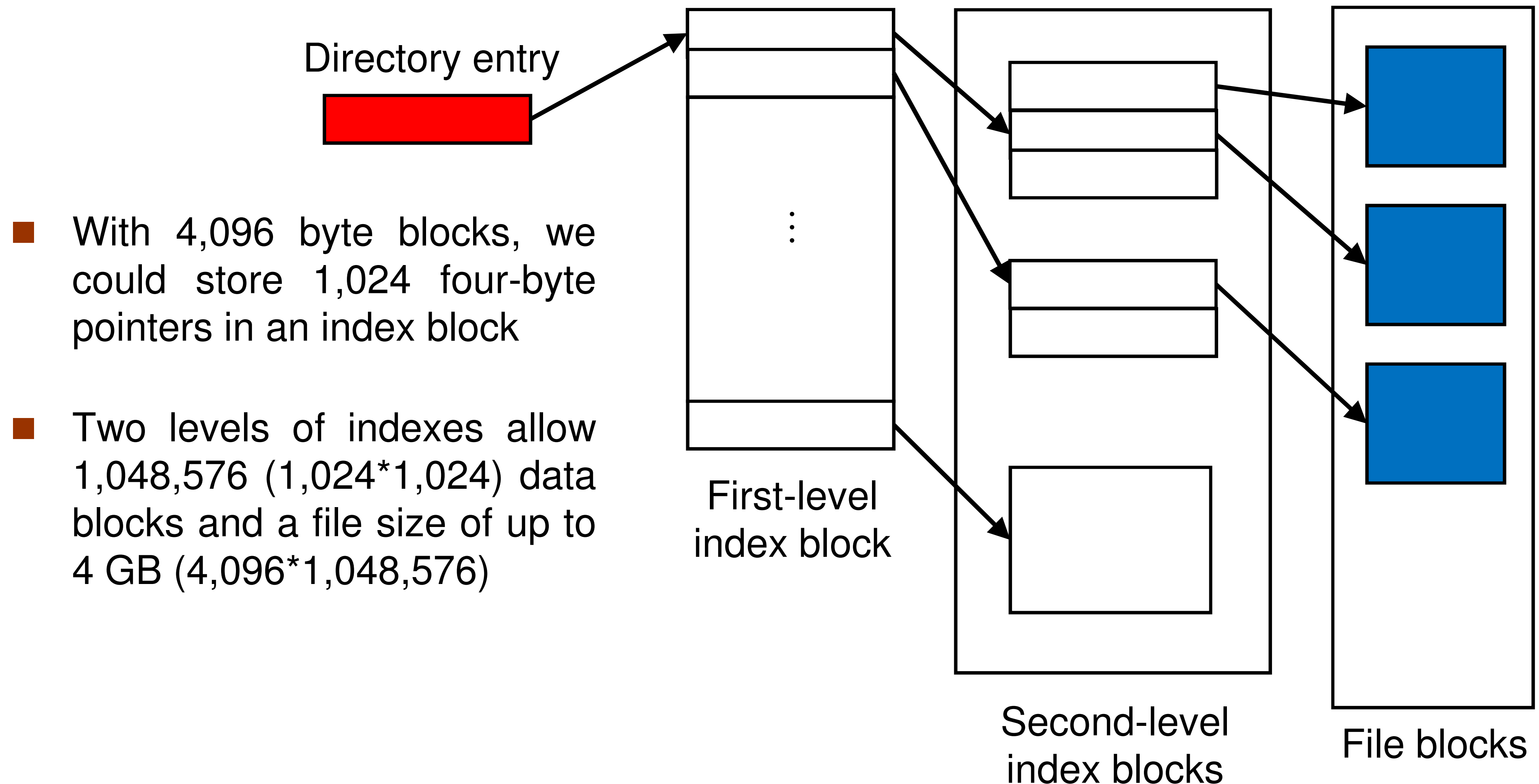
Indexed Allocation

- **Problems**
  - Overhead of the index block
  - If index block not enough, requires some sort of linked allocation
- This raises the question of how large the index block should be?
  - On one hand, we want the index block to be as small as possible since every file requires an index block
  - On the other hand, if the index block is too small, it will not be able to hold enough pointers for larger files

Indexed Allocation

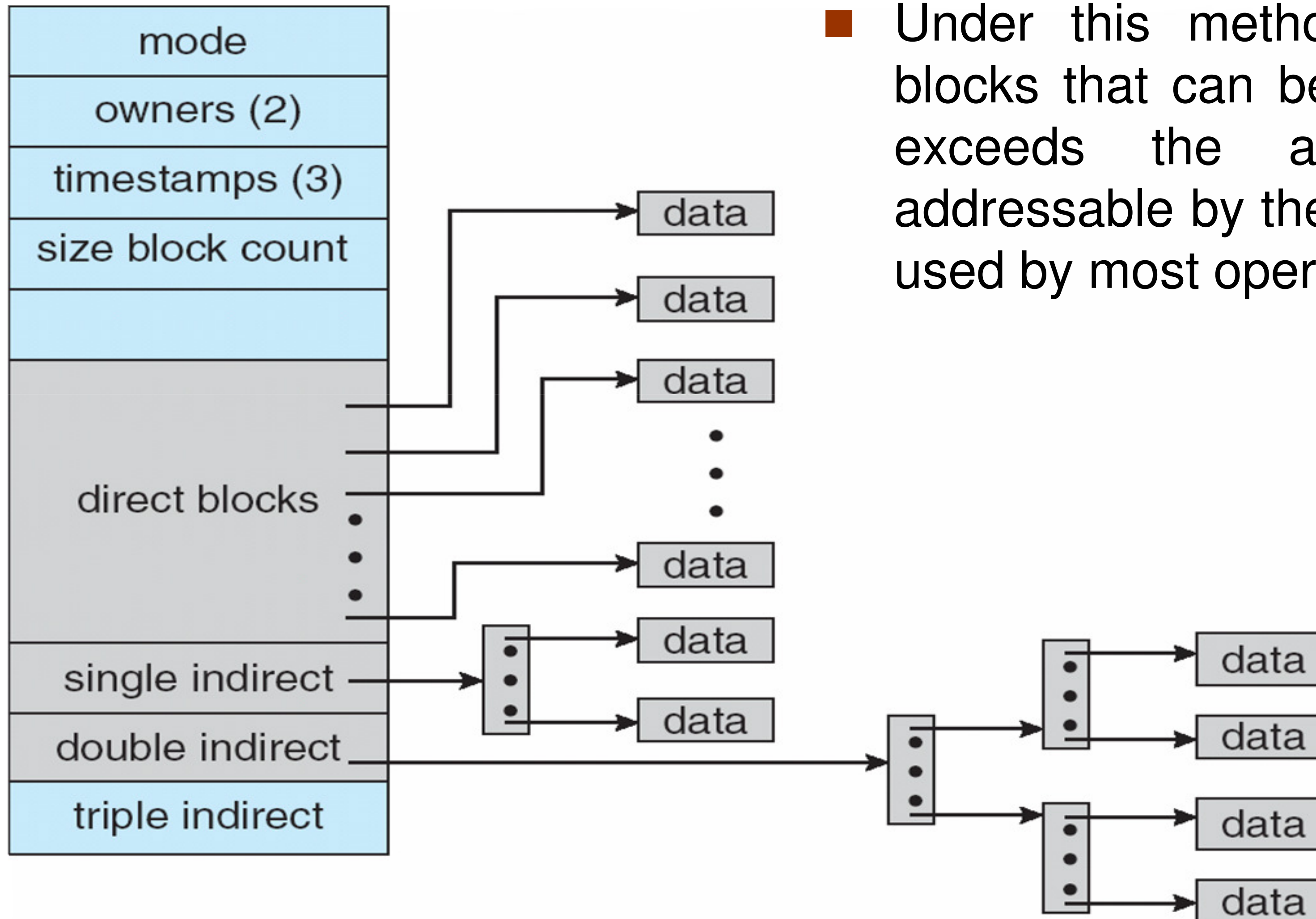
- Schemes to allow for larger files:
  - **Linked scheme** – links together several index blocks (no limit on file size)
  - **Multilevel index** – a first-level index block points to a set of second-level index blocks (could be continued to more levels), which in turn point to the file blocks (limit on file size)
  - **UNIX inodes** – combined scheme that uses some pointers as pointers to **direct blocks** and the last three pointers as pointers to **indirect blocks**: the first points to a **single indirect block**, the second points to a **double indirect block** and the third points to a **triple indirect block**

Indexed Allocation – Multilevel Index



- With 4,096 byte blocks, we could store 1,024 four-byte pointers in an index block
- Two levels of indexes allow 1,048,576 (1,024*1,024) data blocks and a file size of up to 4 GB (4,096*1,048,576)

Indexed Allocation – UNIX Inodes



- Under this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the 4-byte file pointers used by most operating systems