



File System Implementation

Yajin Zhou (<http://yajin.org>)

Zhejiang University



Review

- File sharing
- File protection: what can be done by whom
- ACL
- Unix access control



errata

- What about conflict permissions

-

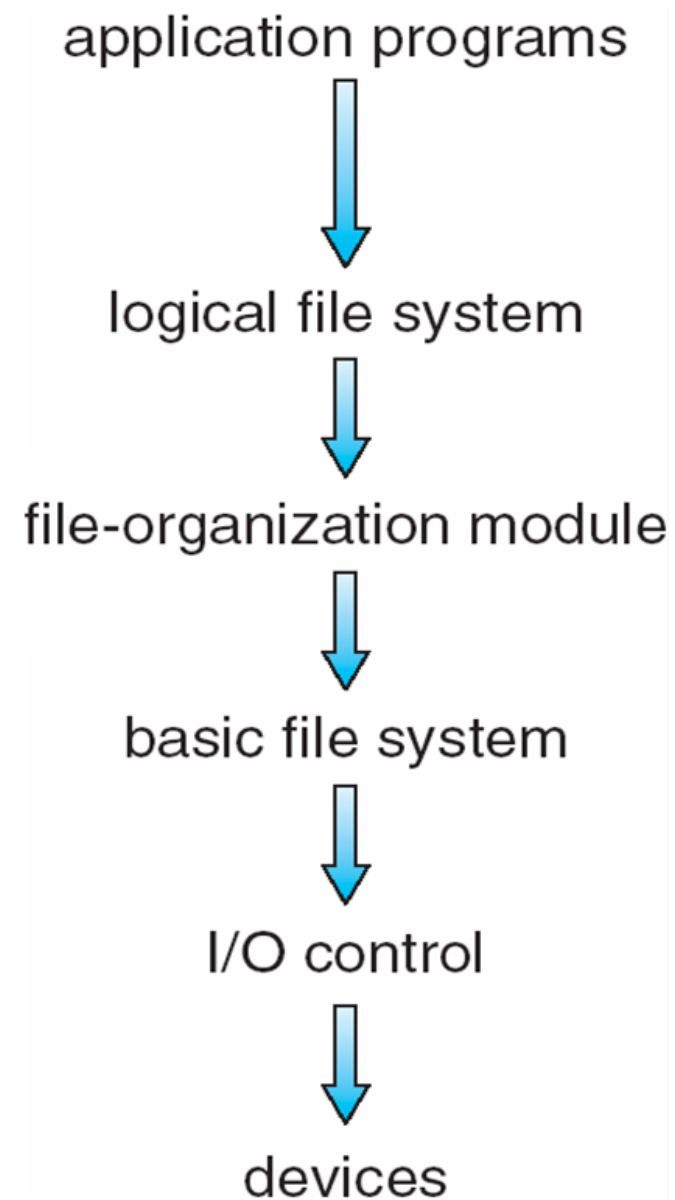
```
os@os:~/temp$ ls -l
total 20
-r--rw-r-- 1 os os   5 Dec 24 21:43 foo
-rw-rwxrwx 1 os os 8600 Dec 24 21:36 test
-rw-rw-r-- 1 os os  35 Dec 24 21:36 test.c
os@os:~/temp$ id
uid=1000(os) gid=1000(os) groups=1000(os),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(smbashare)
os@os:~/temp$
```



File-System Structure

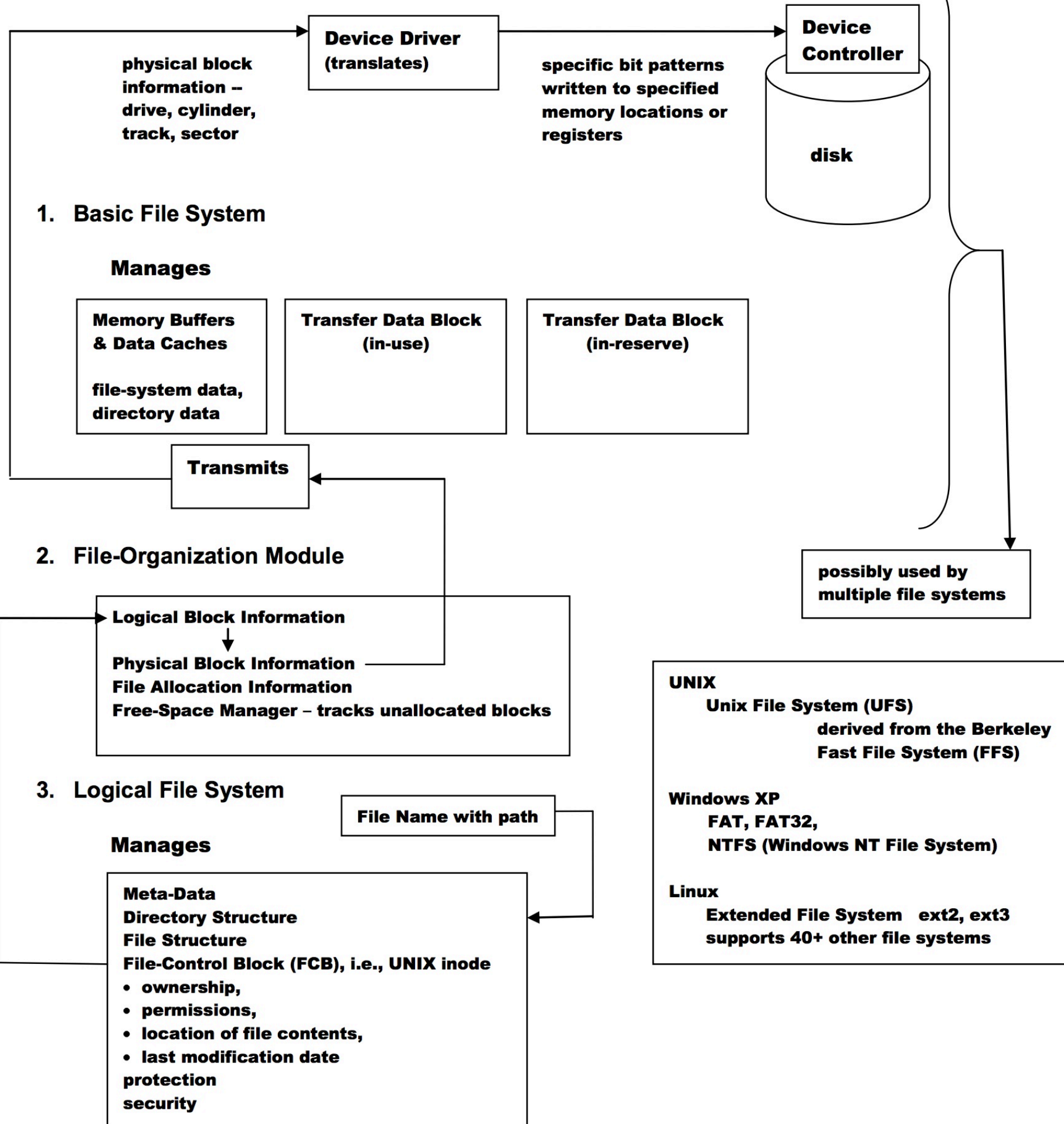
- File is a logical storage unit for a collection of related information
- There are many file systems; OS may support several **simultaneously**
 - Linux has Ext2/3/4, Reiser FS/4, Btrfs...
 - Windows has FAT, FAT32, NTFS...
 - new ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE
- File system resides on **secondary storage** (disks)
 - disk driver provides interfaces to read/write disk blocks
 - **fs provides user/program interface to storage, mapping logical to physical**
 - file control block – storage structure consisting of information about a file
- File system is usually implemented and organized into **layers**

Layered File System



Levels

0. I/O Control -- device drivers





Layered File System

- **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
 - Also manages **memory buffers and caches** (allocation, freeing, replacement)
 - Buffers hold data in transit
 - Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks
 - Translates **logical block # to physical block #**
 - Manages **free space**, disk allocation



Layered File System

- **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
 - FCB(file control block)
- Layering useful for **reducing complexity** and redundancy, but adds overhead and can **decrease performance**
 - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
 - Logical layers can be implemented by any coding method according to OS designer



File-System Implementation

- partition == volume == file system storage space
- File-system needs to maintain **on-disk** and **in-memory** structures
 - on-disk for data storage, in-memory for data access
- **On-disk structure** has several control blocks
 - **boot control block** contains info to boot OS from that volume - per volume
 - only needed if volume contains OS image, usually first block of volume
 - **volume control block** (e.g., *superblock*) contains volume details - per volume
 - total # of blocks, # of free blocks, block size, free block pointers, free FCB count, free FCB pointers
 - **directory structure** organizes the directories and files - per file system
 - A list of **(file names and associated inode numbers)**
 - **per-file file control block** contains many **details about the file - per file**
 - permissions, size, dates, data blocks or pointer to data blocks



A 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

- **In-memory structures** reflects and extends **on-disk structures**
 - **Mount table** storing file system mounts, mount points, file system types
 - In-memory directory-structure cache: holds the directory information about recently accessed directories
 - **system-wide open-file table** contains a copy of the FCB of each file and other info
 - **per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info
 - **I/O Memory Buffers**: hold file-system blocks while they are being read from or written to disk



File Creation

- application process requests the creation of a new file
- logical file system allocates a new FCB, i.e., inode structure
- appropriate directory is updated with the new file name and FCB, i.e., inode



Directory

- Unix – directories are treated as files containing special data
- Windows – directories differently from files;
 - they require a separate set of systems calls to create, manipulate, etc



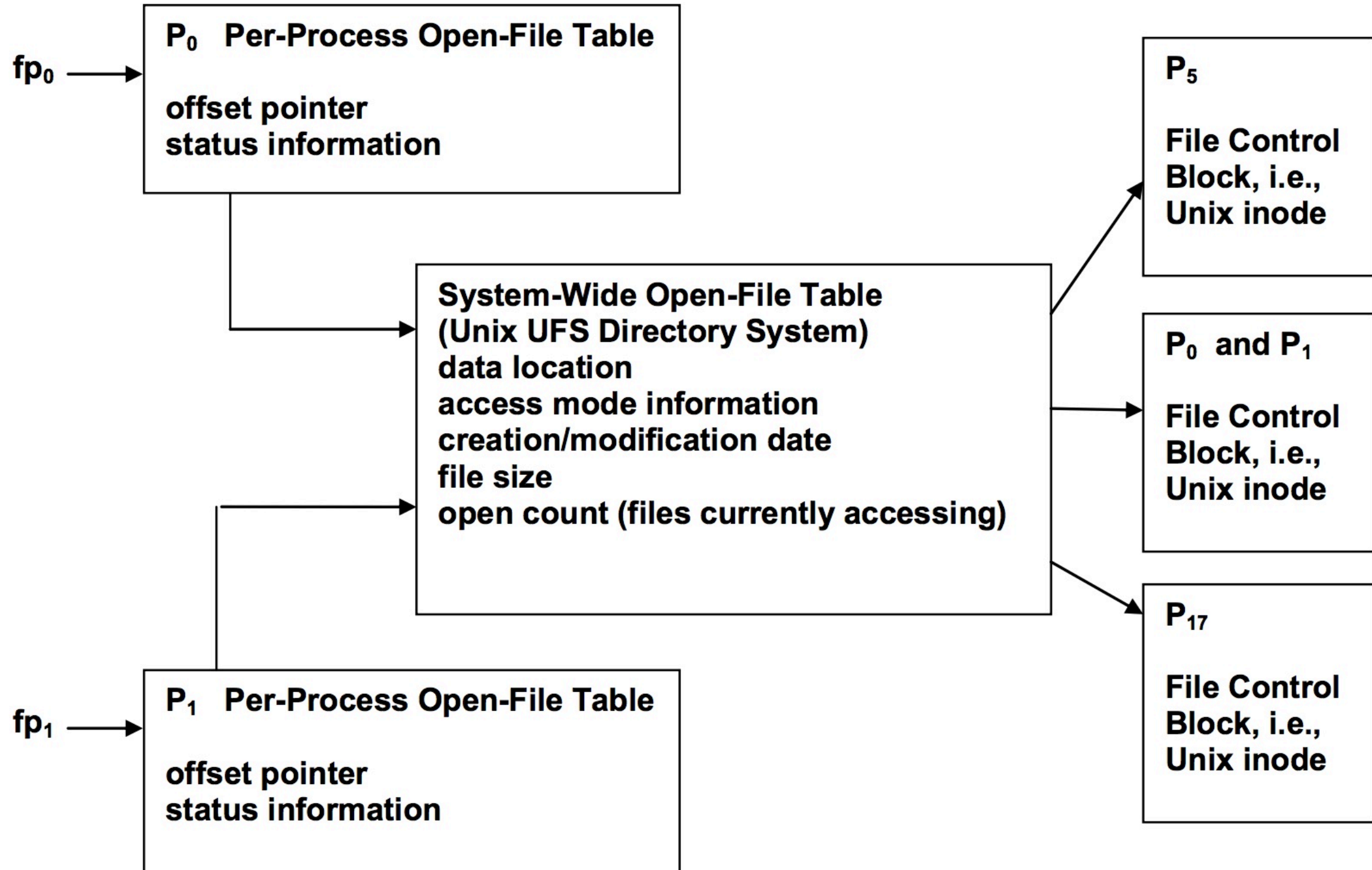
Operations - open()

- search **System-Wide Open-File Table** to see if file is currently in use
 - if it is, create a Per-Process Open-File table entry pointing to the existing System-Wide Open-File Table
 - if it is not, search the directory for the file name; once found, place the **FCB** in the System-Wide Open-File Table
- make an entry, i.e., Unix file descriptor, Windows file handle in the **Per-Process Open-File Table**, with pointers to the entry in the **System-Wide Open-File Table** and other fields which include a pointer to the current location in the file and the access mode in which the file is open

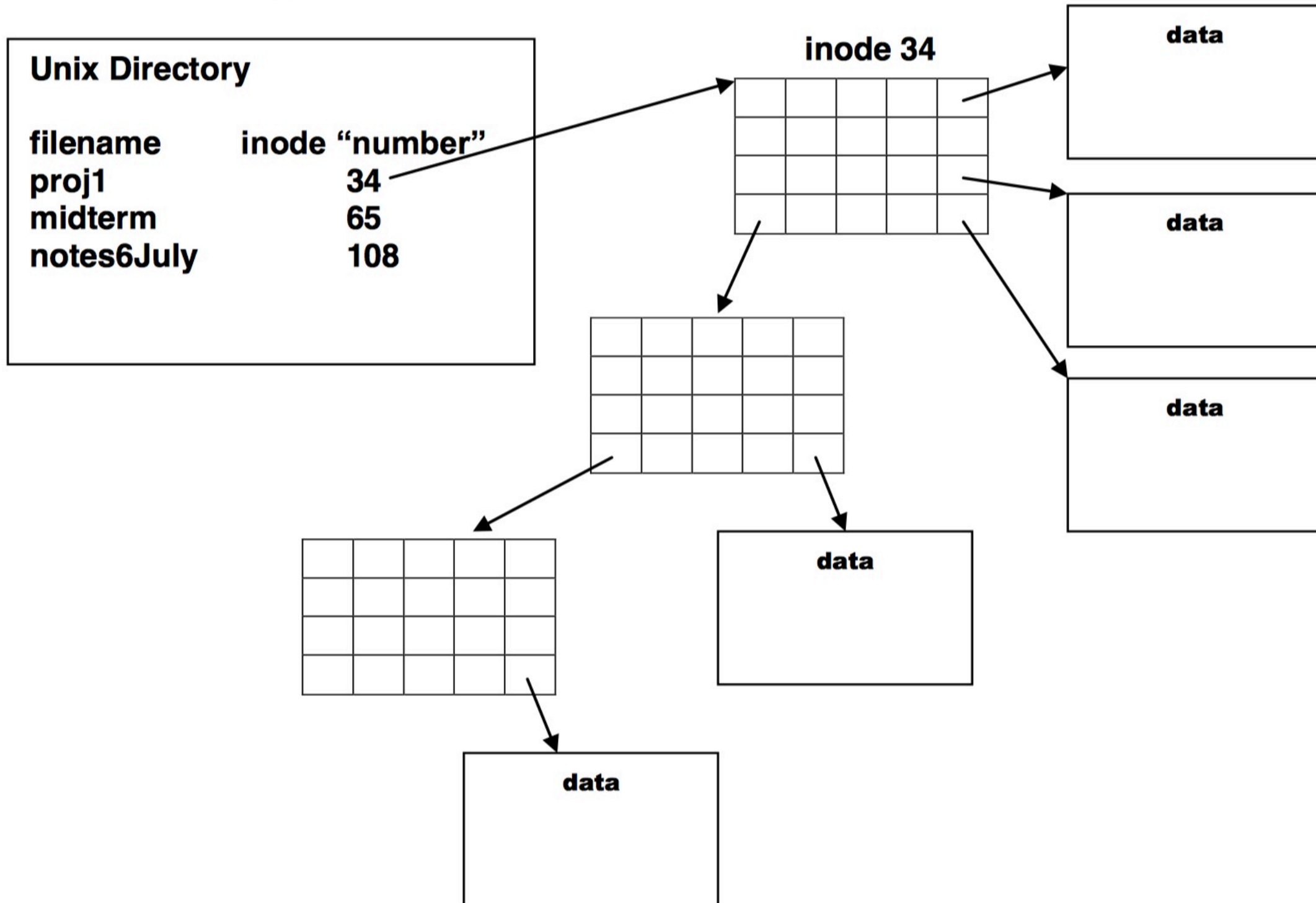


Operations - open()

- increment the open count in the System-Wide Open-File Table
- returns a pointer to the appropriate entry in the Per-Process Open-File Table
- all subsequent operations are performed with **this pointer**
- process closes file -> Per-Process Open-File Table entry is removed;
open count decremented
- all processes close file -> copy in-memory directory information to disk and System-Wide Open-File Table is removed from memory



Unix i-node System





Unix (UFS)

- System-Wide Open-File Table holds inodes for files, directories, devices, and network connections
- inode numbering system is only unique within a given file system



Mounting File Systems

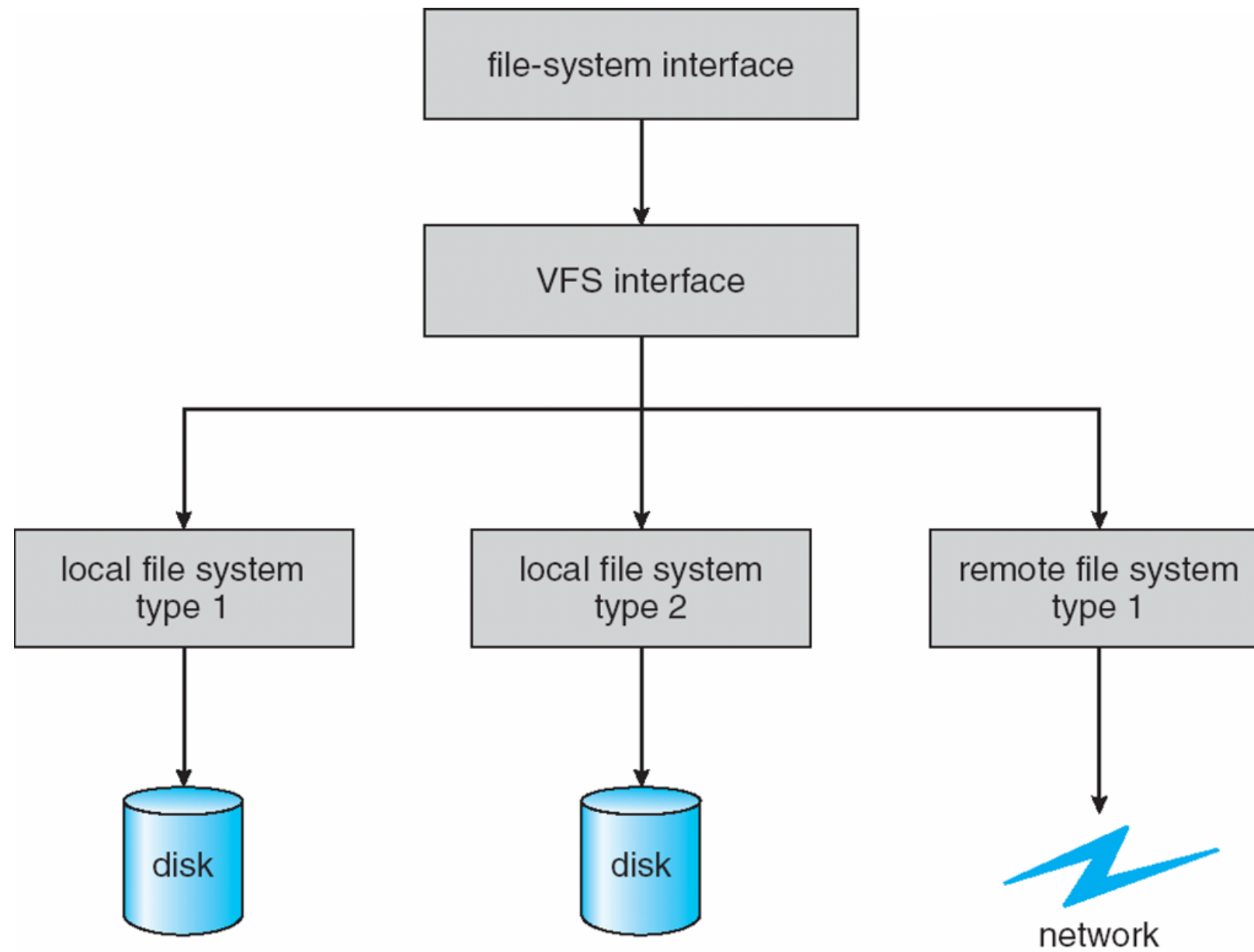
- Boot Block – series of sequential blocks containing a memory image of a program, call the boot loader, that locates and mounts the root partition; the partition contains the kernel; the boot loader locates, loads, and starts the kernel executing
- In-memory mount table – external file systems must be mounted on devices, the mount table records the mount points, types of file systems mounted, and an access path to the desired file system
- Unix – the in-memory mount table contains a pointer to the **superblock** of the file system on that device



Virtual File Systems

- **VFS** provides an **object-oriented** way of implementing file systems
 - OS defines **a common interface for FS**, all FSes implement them
 - *system call is implemented based on this common interface*
 - it allows the same syscall API to be used for different types of FS
- VFS separates FS generic operations from implementation details
 - implementation can be one of many FS types, or network file system
 - OS can dispatches syscalls to appropriate FS implementation routines

Virtual File System





Virtual File System Example

- Linux defines four **VFS object types**:
 - **superblock**: defines the file system type, size, status, and other metadata
 - **inode**: contains metadata about a **file** (location, access mode, owners...)
 - **dentry**: associates names to inodes, and the directory layout
 - **file**: actual data of the file
- VFS defines set of operations on the objects that must be implemented
 - the set of operations is saved in a function table

```
struct file_operations {
    int (*lseek) (struct inode *, struct file *, off_t, int);
    int (*read) (struct inode *, struct file *, char *, int);
    int (*write) (struct inode *, struct file *, const char *, int);
    int (*readdir) (struct inode *, struct file *, void *, filldir_t);
    int (*select) (struct inode *, struct file *, int, select_table *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    void (*release) (struct inode *, struct file *);
    int (*fsync) (struct inode *, struct file *);
    int (*fasync) (struct inode *, struct file *, int);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
};
```



Review

- File system layers
- File system implementation
 - On-disk structure, in-memory structure
- File creation(), open()
- VFS



Directory Implementation

- **Linear list of file names** with pointer to the file metadata
 - simple to program, but **time-consuming to search** (e.g., linear search)
 - could keep files ordered alphabetically via linked list or use B+ tree
- **Hash table:** linear list with hash data structure to reduce search time
 - collisions are possible: two or more file names hash to the same location



Disk Block Allocation

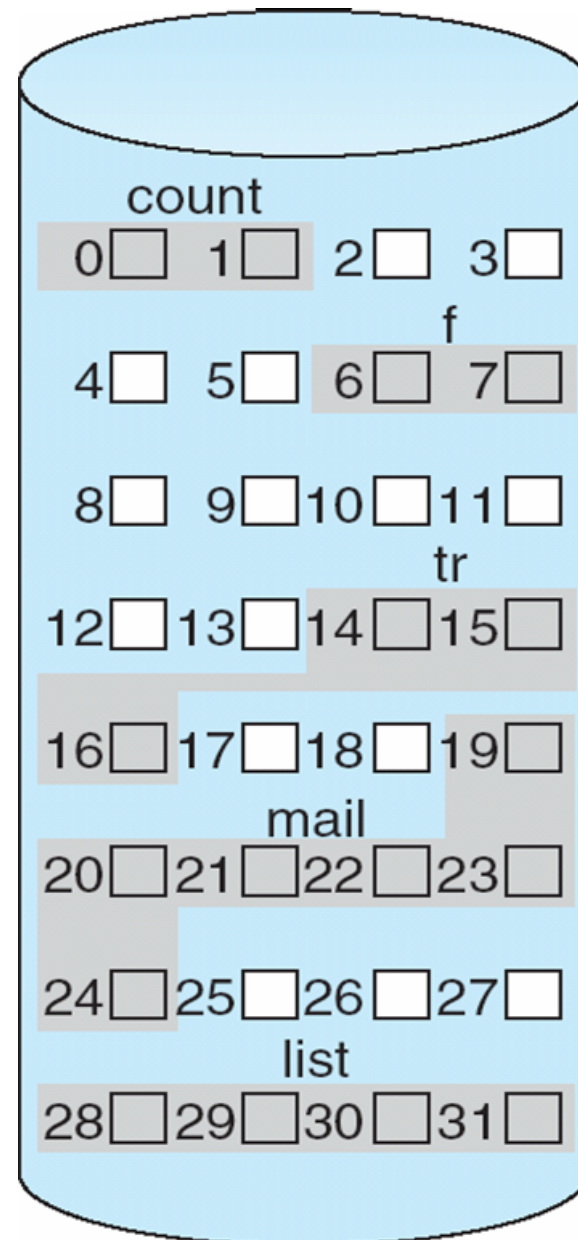
- **Files** need to be allocated with disk blocks to store data
 - different allocation strategies have different complexity and performance
- Many allocation strategies:
 - contiguous
 - linked
 - indexed
 - ...



Contiguous Allocation

- Contiguous allocation: each file occupies set of **contiguous blocks**
 - best performance in most cases
 - simple to implement: only starting location and length are required
- Contiguous allocation is not flexible
 - how to *increase/decrease* file size?
 - need to know file size at the file creation?
 - **external fragmentation**
 - how to compact files offline or online to reduce external fragmentation
 - need for **compaction** off-line (downtime) or on-line
 - appropriate for sequential disks like **tape**
- Some file systems use **extent-based contiguous allocation**
 - extent is a set of contiguous blocks
 - a file consists of extents, extents are not necessarily adjacent to each other

Contiguous Allocation



directory

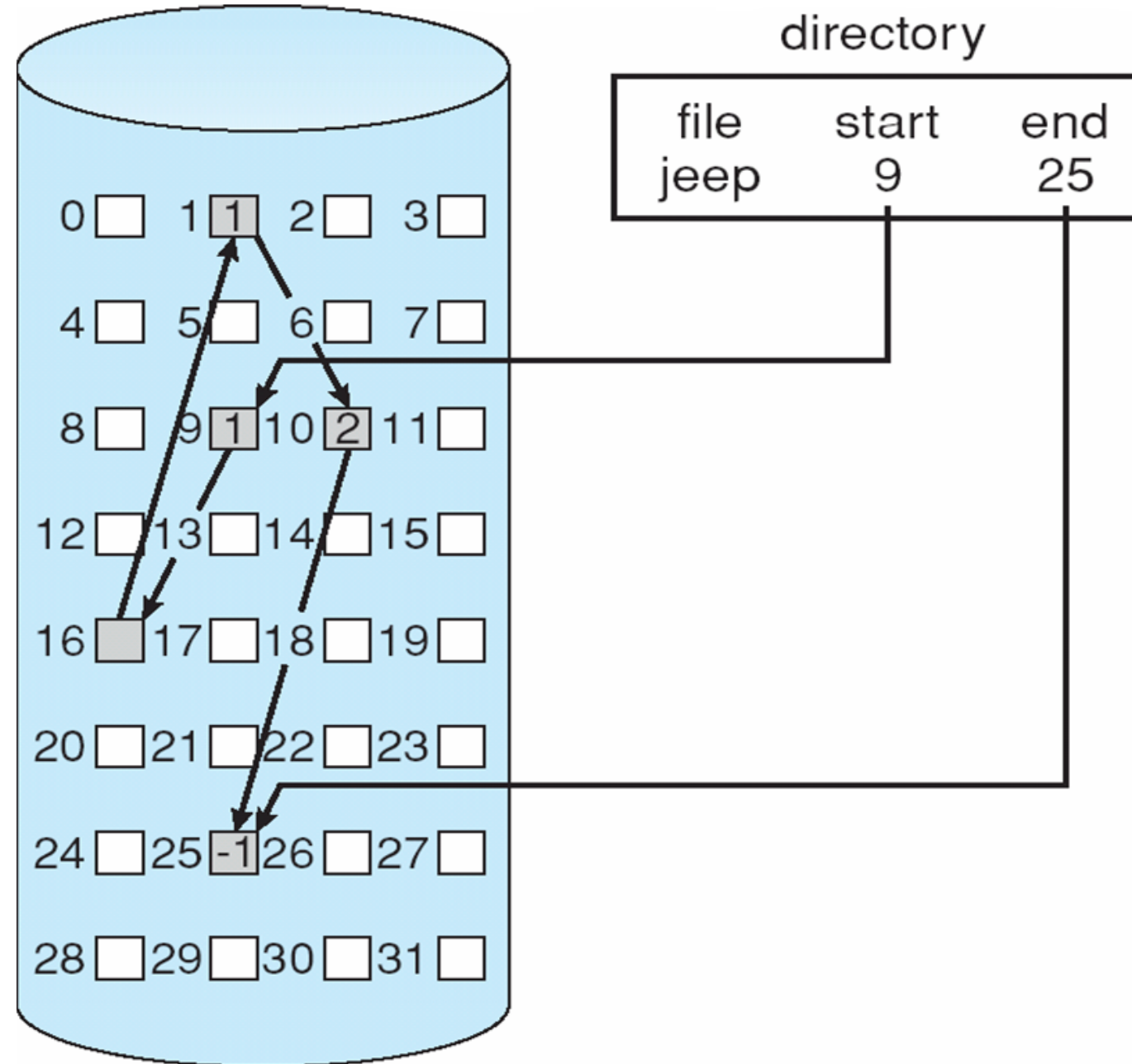
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2



Linked Allocation

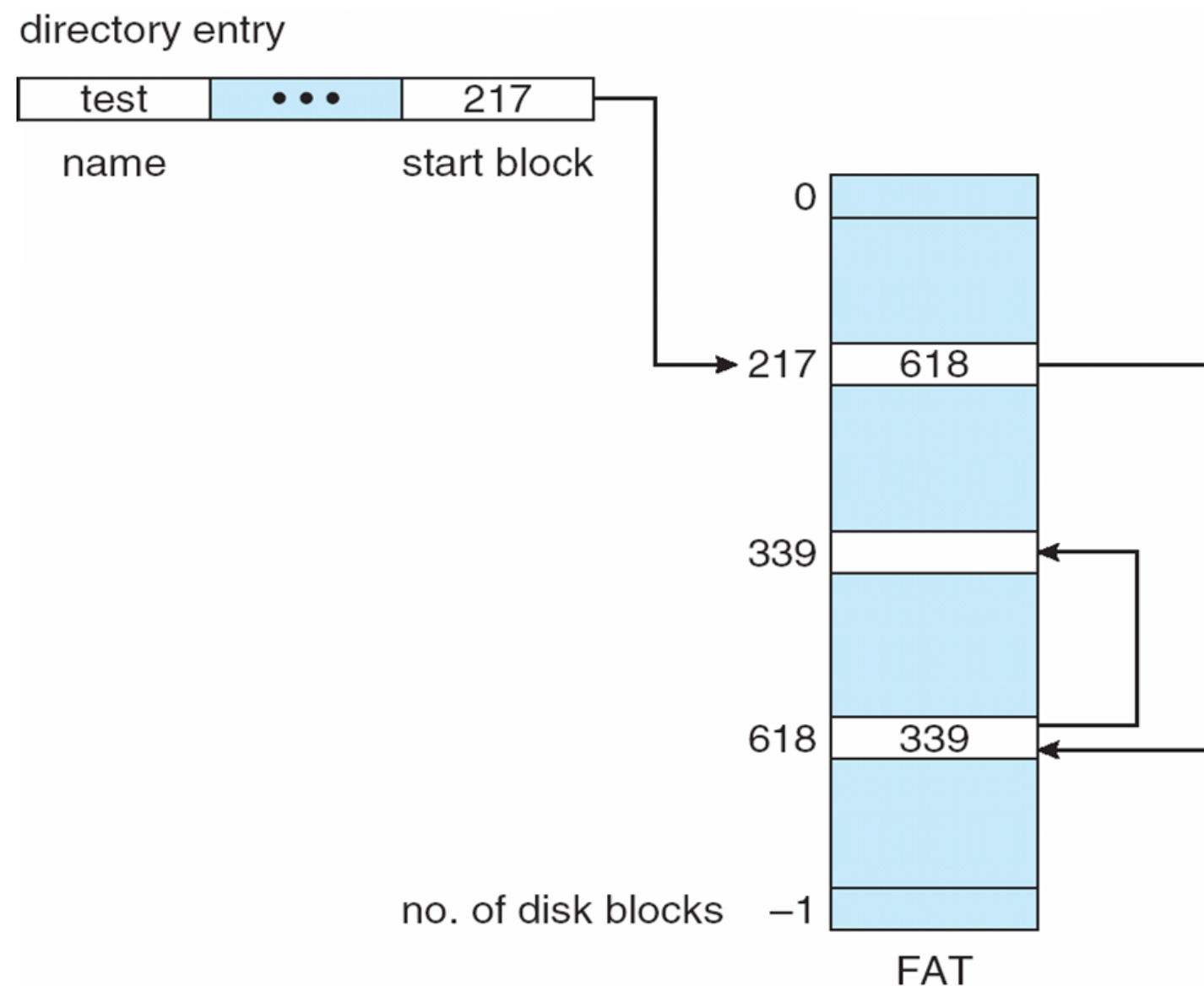
- Linked allocation: each file is a **linked list of disk blocks**
 - each block contains pointer to **next block**, file ends at nil pointer
 - blocks may be scattered anywhere on the disk (no **external fragmentation, no compaction**)
- *Disadvantages*
 - *locating a file block can take many I/Os and disk seeks*
 - *Pointer size: 4 of 512 bytes are used for pointer - 0.78% space is wasted*
 - *Reliability: what about the pointer has corrupted!*
- *Improvements: cluster the blocks - like 4 blocks*
 - *however, has internal fragmentation*

Linked Allocation



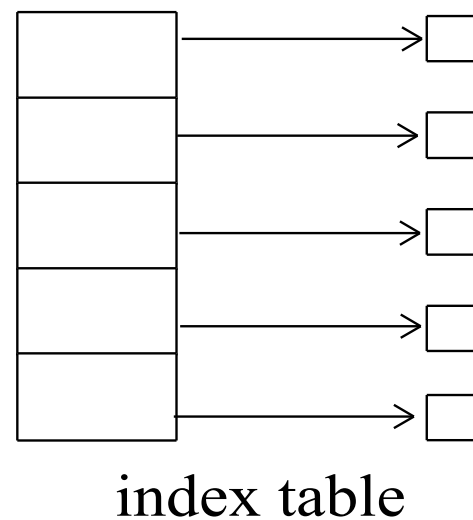
File-Allocation Table (FAT): MS-DOS

- FAT (File Allocation Table) uses linked allocation

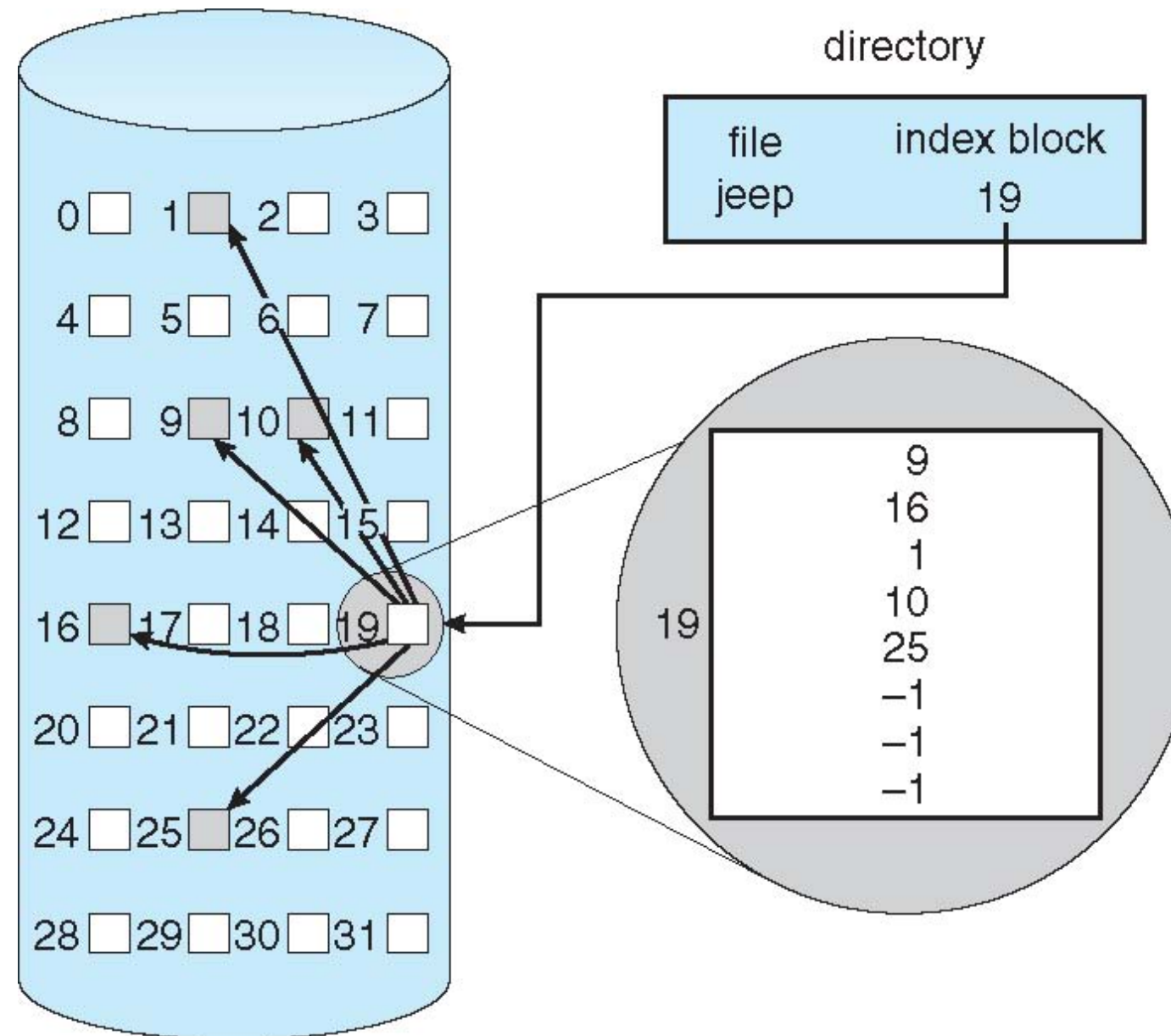


Indexed Allocation

- Indexed allocation: each file has its own **index blocks of pointers to its data blocks**
 - index table provides **random access** to file data blocks
 - no **external fragmentation**, but overhead of index blocks
 - allows **holes** in the file
 - Index block needs space - waste for small files



Indexed Allocation

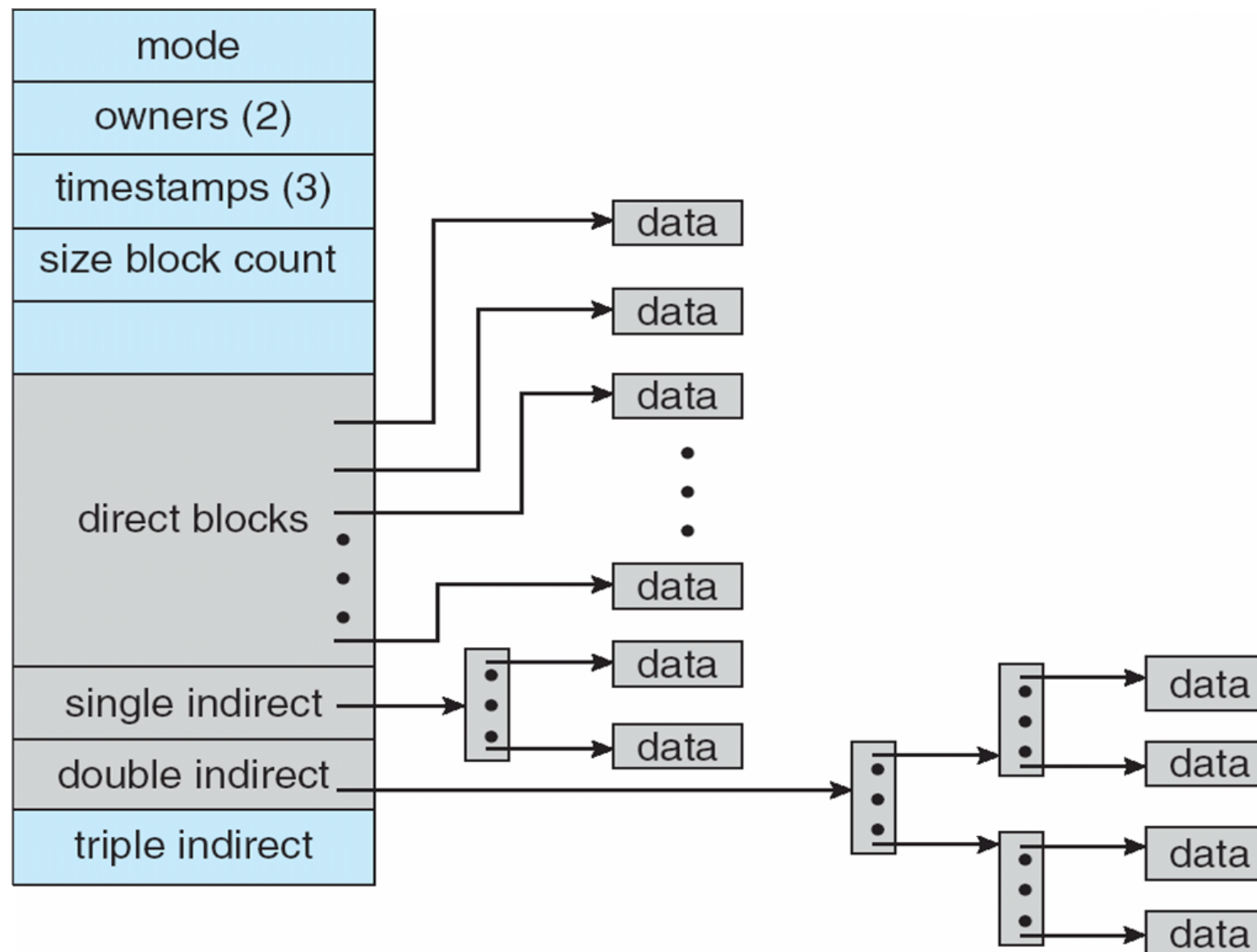




Indexed Allocation

- Need a method to allocate index blocks - cannot too big or too small
 - linked index blocks: link index blocks to support huge file
 - multiple-level index blocks (e.g., 2-level)
 - combined scheme
 - First 15 pointers are in inode
 - Direct block: first 12 pointers
 - Indirect block: next 3 pointers

Combined Scheme: UNIX UFS





Allocation Methods

- Best allocation method depends on file access type
 - **contiguous** is great for **sequential** and **random**
 - **linked** is good for **sequential**, not random
 - **indexed** (combined) is more complex
 - single block access may require 2 index block reads then data block read
 - clustering can help improve throughput, reduce CPU overhead
 - cluster is a set of contiguous blocks
- Disk I/O is slow, reduce as many disk I/Os as possible
 - Intel Core i7 extreme edition 990x (2011) at 3.46Ghz = 159,000 MIPS
 - typical disk drive at 250 I/Os per second
 - $159,000 \text{ MIPS} / 250 = 630$ million instructions during one disk I/O
 - fast SSD drives provide 60,000 IOPS
 - $159,000 \text{ MIPS} / 60,000 = 2.65$ millions instructions during one disk I/O



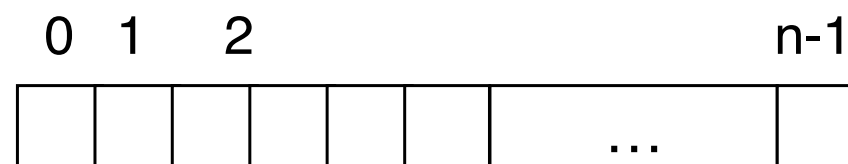
Free-Space Management

- File system maintains free-space list to track available blocks/clusters
 - The space of deleted files should be reclaimed
- Many allocation methods:
 - bit vector or bit map
 - linked free space
 - ...



Bitmap Free-Space Management

- Use one bit for each block, track its allocation status
 - relatively easy to find contiguous blocks
 - bit map requires extra space
 - example: block size = 4KB = 2^{12} bytes
- disk size = 2^{40} bytes (1 terabyte)
- $n = 2^{40}/2^{12} = 2^{28}$ bits (or 256 MB)
- if clusters of 4 blocks -> 64MB of memory



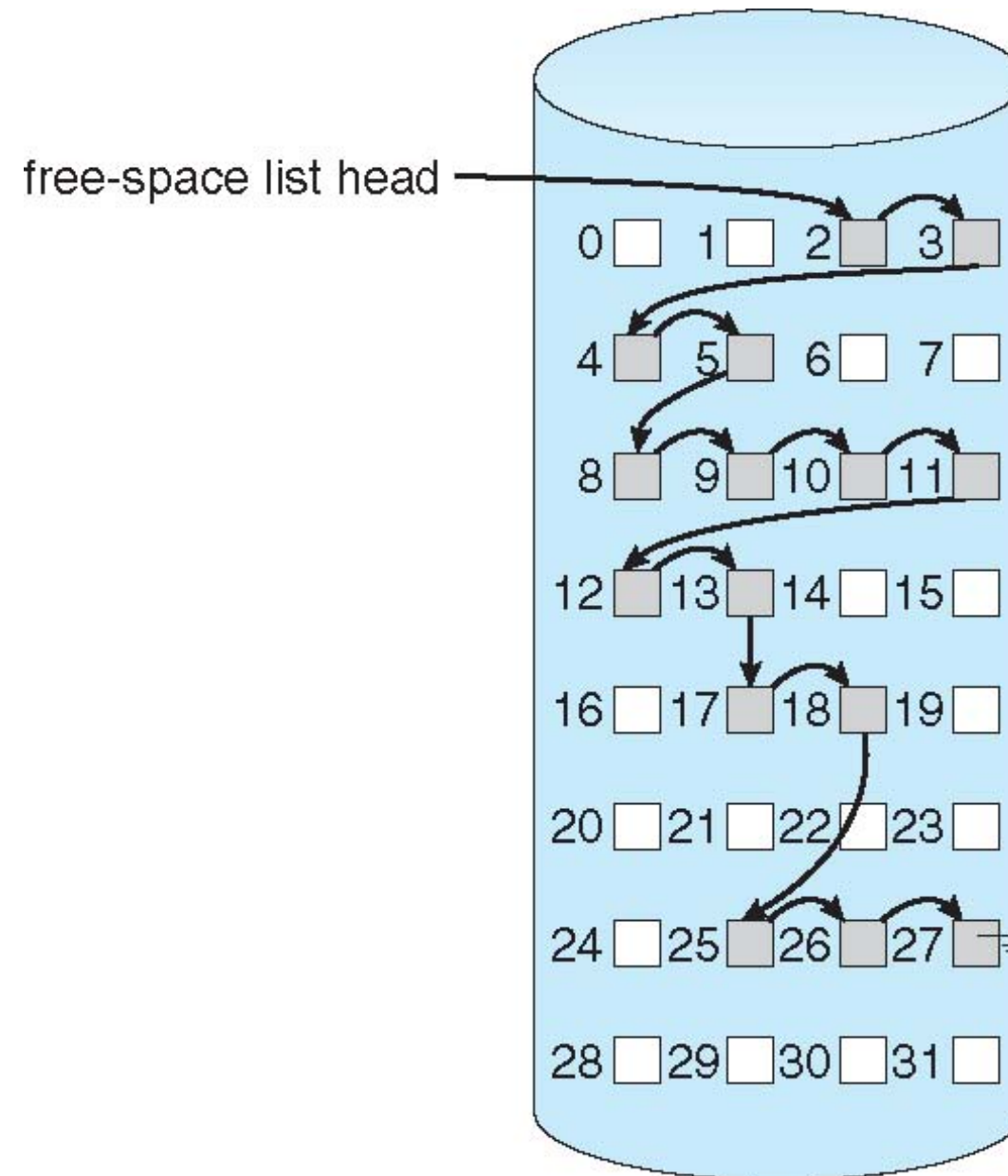
$$\text{bit}[i] = \begin{cases} 1 \rightarrow \text{block}[i] \text{ free} \\ 0 \rightarrow \text{block}[i] \text{ occupied} \end{cases}$$



Linked Free Space

- Keep free blocks in linked list
 - no waste of space, just use the memory in the free block for pointers
 - cannot get contiguous space easily
 - Usually no need to traverse the entire list: return the first one

Linked Free Space





Grouping and Counting

- Simple linked list of free-space is inefficient
 - one extra disk I/O to allocate one free block (disk I/O is extremely slow)
 - allocating **multiple free blocks** require traverse the list
 - difficult to allocate contiguous free blocks
- **Grouping**: use indexes to group free blocks
 - store address of **n-1** free blocks in the first free block, plus a pointer to the next **index block**
 - allocating **multiple free blocks does not need to traverse the list**
- **Counting**: a link of clusters (starting block + # of contiguous blocks)
 - space is frequently contiguously used and freed
 - in link node, keep address of first free block and # of following free blocks



File System Performance

- File system efficiency and performance dependent on:
 - disk allocation and directory algorithms
 - types of data kept in file's directory entry
 - pre-allocation or as-needed allocation of metadata structures
 - fixed-size or varying-size data structures

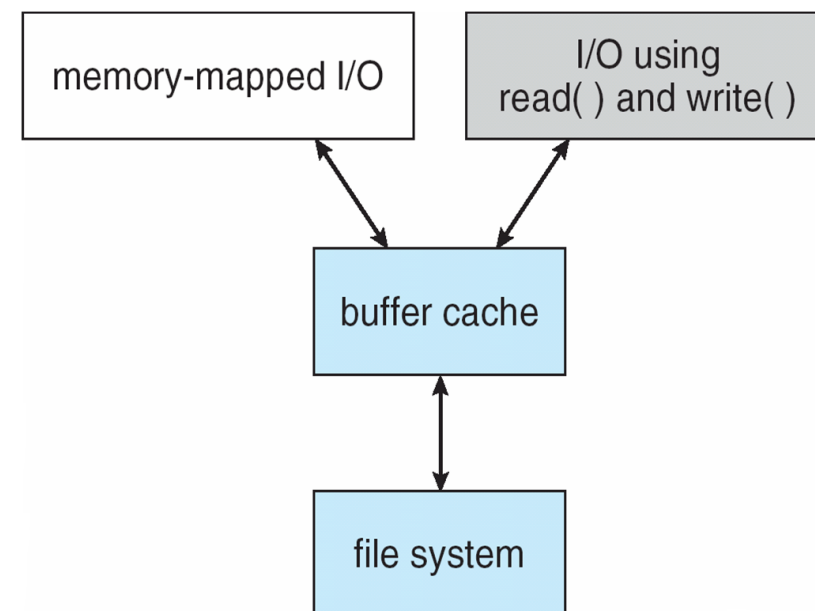
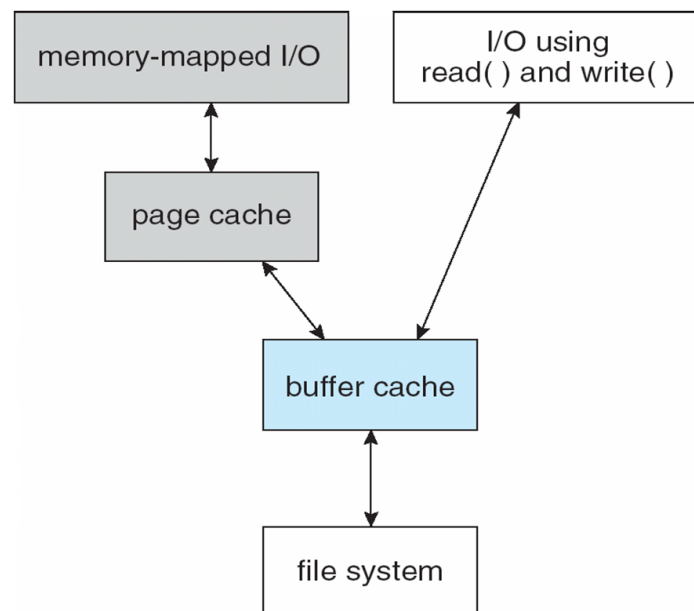


File System Performance

- To improve file system performance:
 - keeping **data and metadata close together**
 - use cache: separate section of main memory for frequently used blocks
 - use **asynchronous writes**, it can be buffered/cached, thus faster
 - cannot cache synchronous write, writes must hit disk before return
 - synchronous writes sometimes requested by apps or needed by OS
 - **free-behind and read-ahead**: techniques to optimize sequential access
 - **Reads frequently slower than write**: really?

Page Cache

- OS has different levels of cache:
 - a **page cache** caches pages for MMIO, such as memory mapped files
 - file systems uses **buffer** (disk) **cache** for disk I/O
 - memory mapped I/O may be cached twice in the system
- A unified buffer cache uses the same page cache to cache both memory-mapped pages and disk I/O to avoid double caching





Recovery

- File system needs consistency checking to ensure consistency
 - compares data in directory with some metadata on disk for consistency
 - fs recovery can be slow and sometimes fails
- File system recovery methods
 - backup
 - log-structured file system



Log Structured File Systems

- In LSFS, metadata for updates sequentially written to a **circular log**
 - once changes written to the log, it is committed, and syscall can return
 - log can be located on the other disk/partition
 - meanwhile, log entries are replayed on the file system to actually update it
 - when a transaction is replayed, it is removed from the log
 - a log is circular, but un-committed entries will not be overwritten
 - garbage collection can reclaim/compact log entries
 - upon system crash, only need to replay transactions existing in the log

Will Talk Log Structured File System Later

hw14 is out