Chapter 12: File System Implementation

Zhi Wang
Florida State University

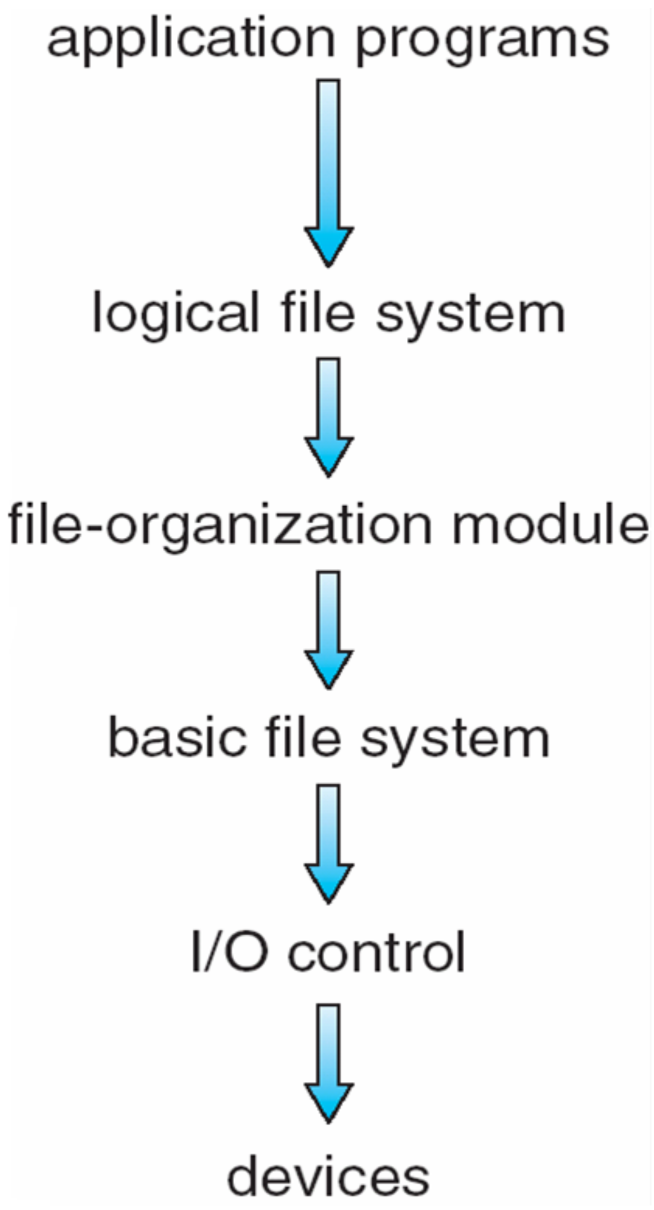
Content

- File system structure
- File system implementation
- Directory implementation
- Allocation and free-space management
- Recovery
- Examples: NFS

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**
  - layering can reduce implementation complexity and redundancy
  - it may increase overhead and decrease performance

Layered File System



File System Layers

- **Device drivers** manage disk devices at the **I/O control layer**
  - device driver accepts commands to access raw disk
  - command “read drive1, cylinder 72, track 2, sector 10, into memory 1060”
  - it converts the command to hardware devices access (i.e., using registers)
- **Basic file system** provides methods to **access physical blocks**
  - it translates commands like “retrieve block 123” to device driver
  - manages memory buffers and caches (allocation, freeing, replacement)

File System Layers

- **File organization module** understands **files**, **logical address**, and **physical blocks**
  - it translates logical block # to physical block #
  - it manages free space, disk allocation
- **Logical file system** understand **file system structures** (metadata)
  - it translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in Unix)
  - directory management and protection

File-System Implementation

- 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
    - only needed if volume contains OS image, usually first block of volume
  - **volume control block** (e.g., *superblock*) contains volume details
    - total # of blocks, # of free blocks, block size, free block pointers or array
  - **directory structure** organizes the directories and files
    - file names and layout
  - **per-file file control block** contains many details about the file
    - *inode* number, permissions, size, dates

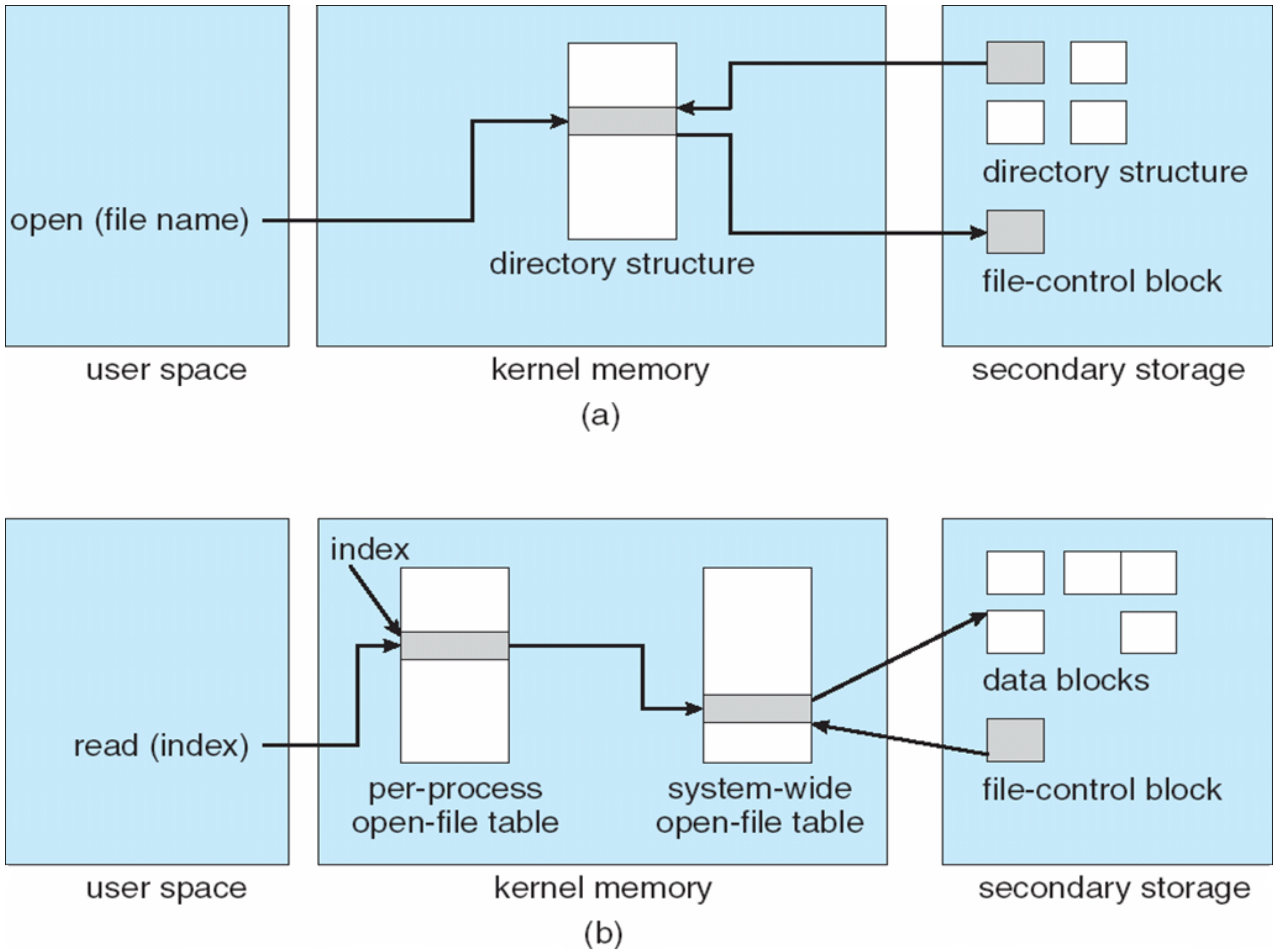
A Typical File Control Block

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**
  - it provides **API** for applications to access files
    - e.g., open file tables to store the currently open file
  - it create a **uniform name space** for all the files
    - e.g., partitions/disks can be mounted into this name space
  - **buffering** and **caching** to improve performance and bridge speed mismatch
    - e.g., in-memory directory cache to speed up file search

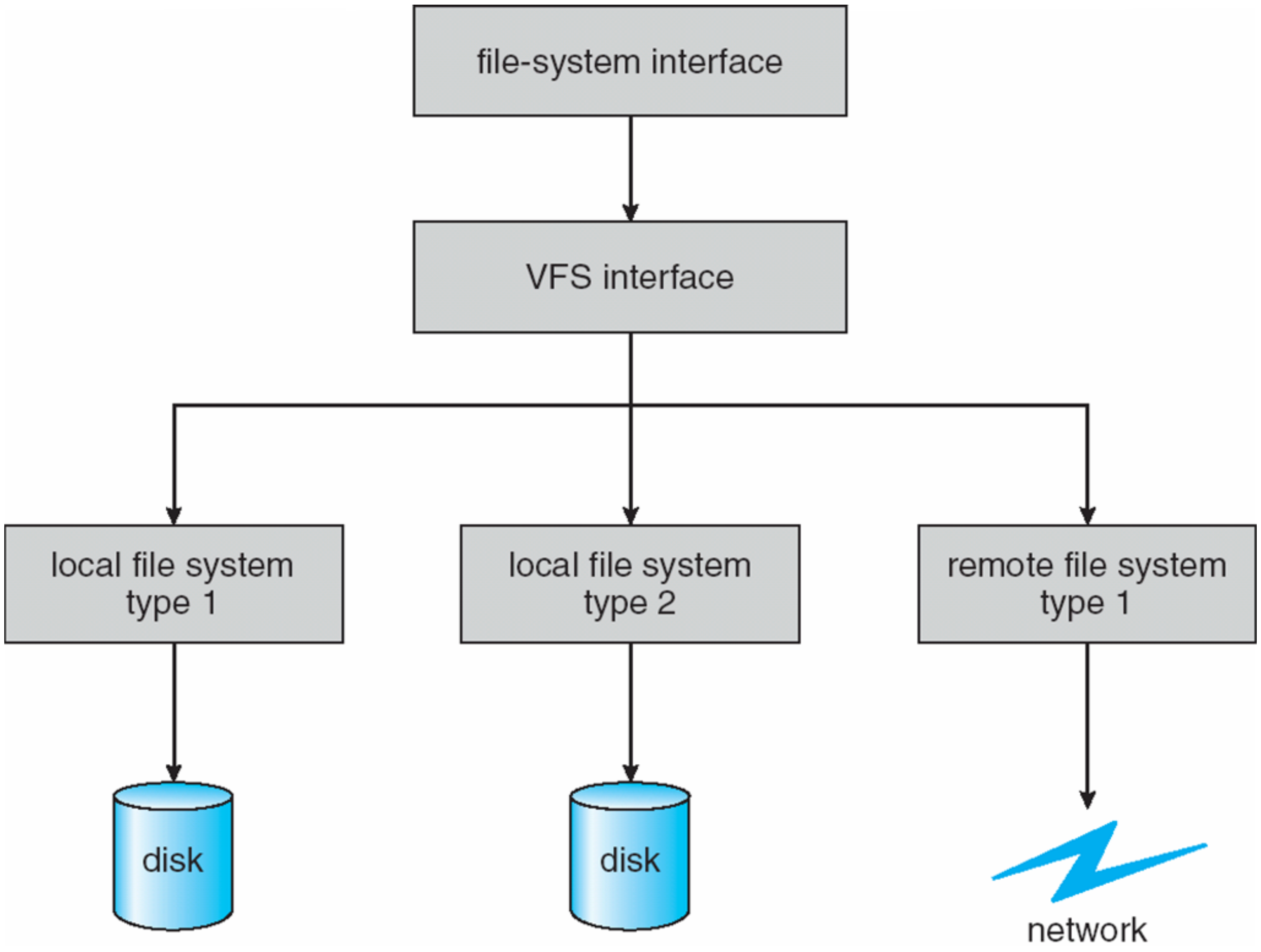
In-Memory File System Structures



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);
};
```

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

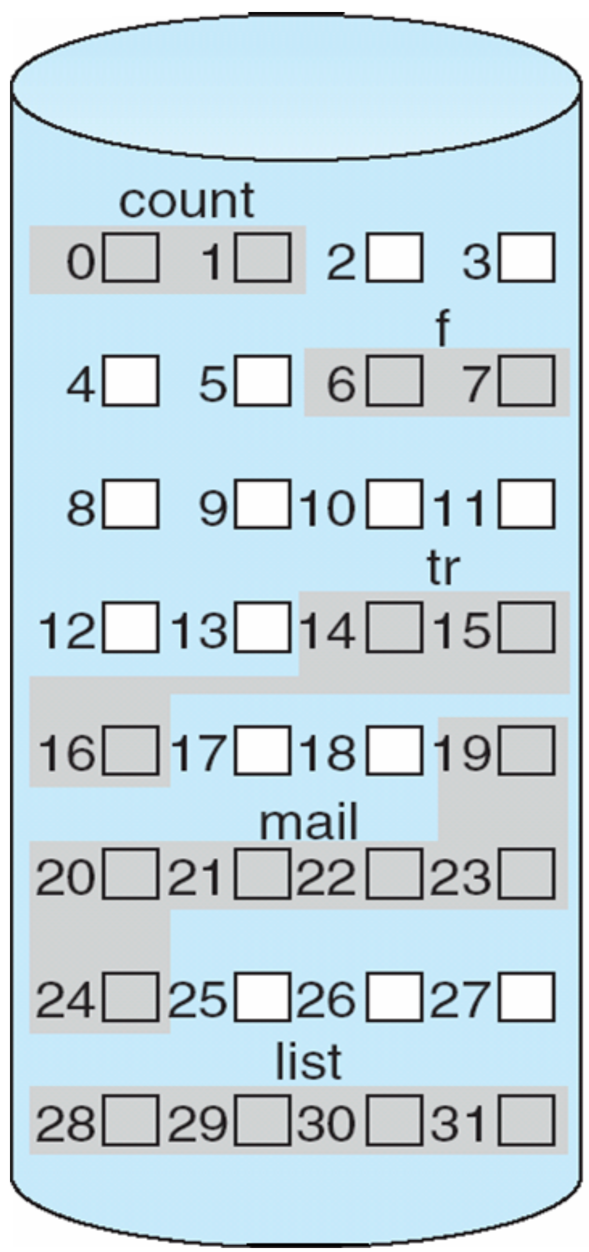
Disk Block Allocation

- 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
  - 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



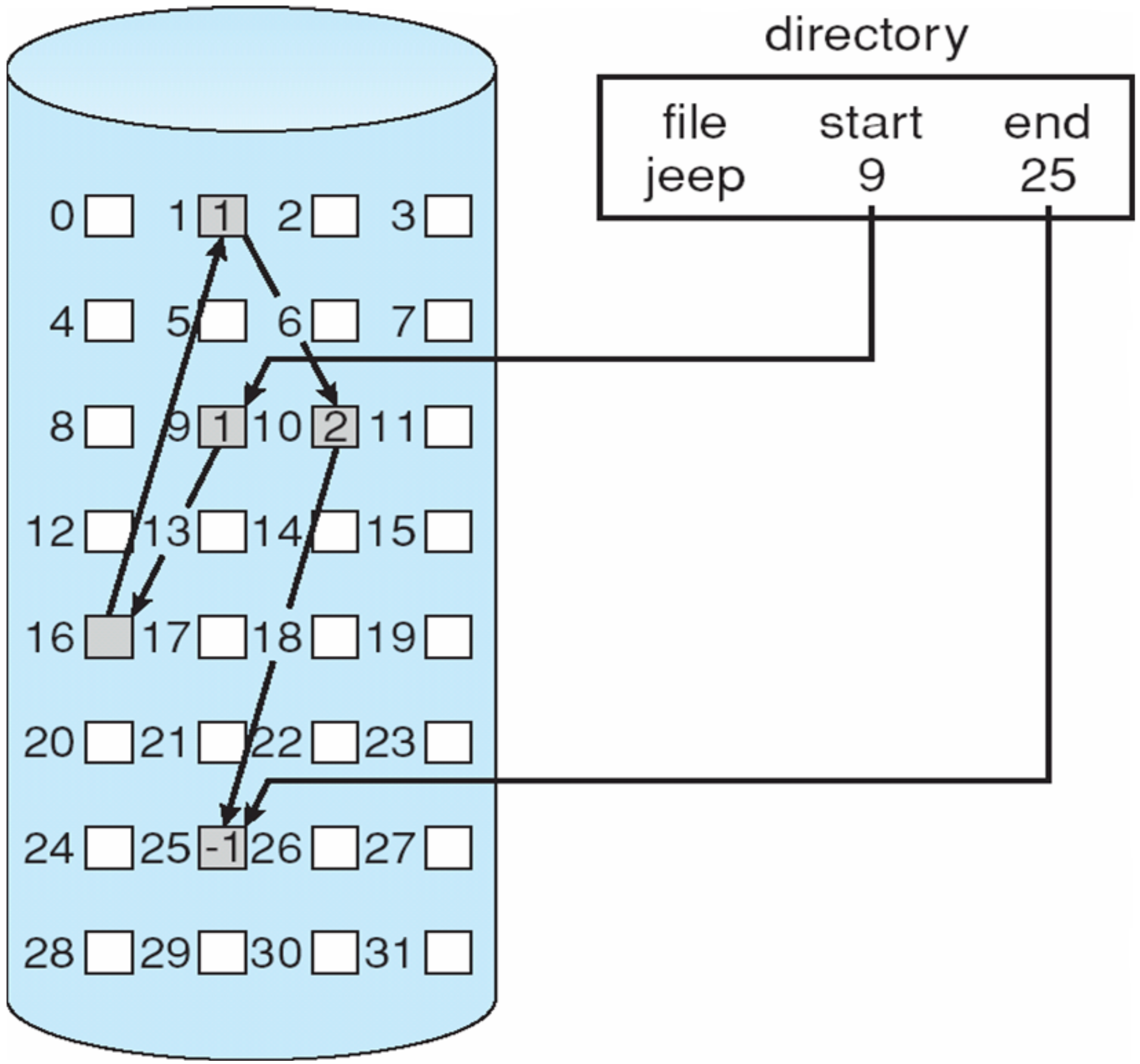
directory

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

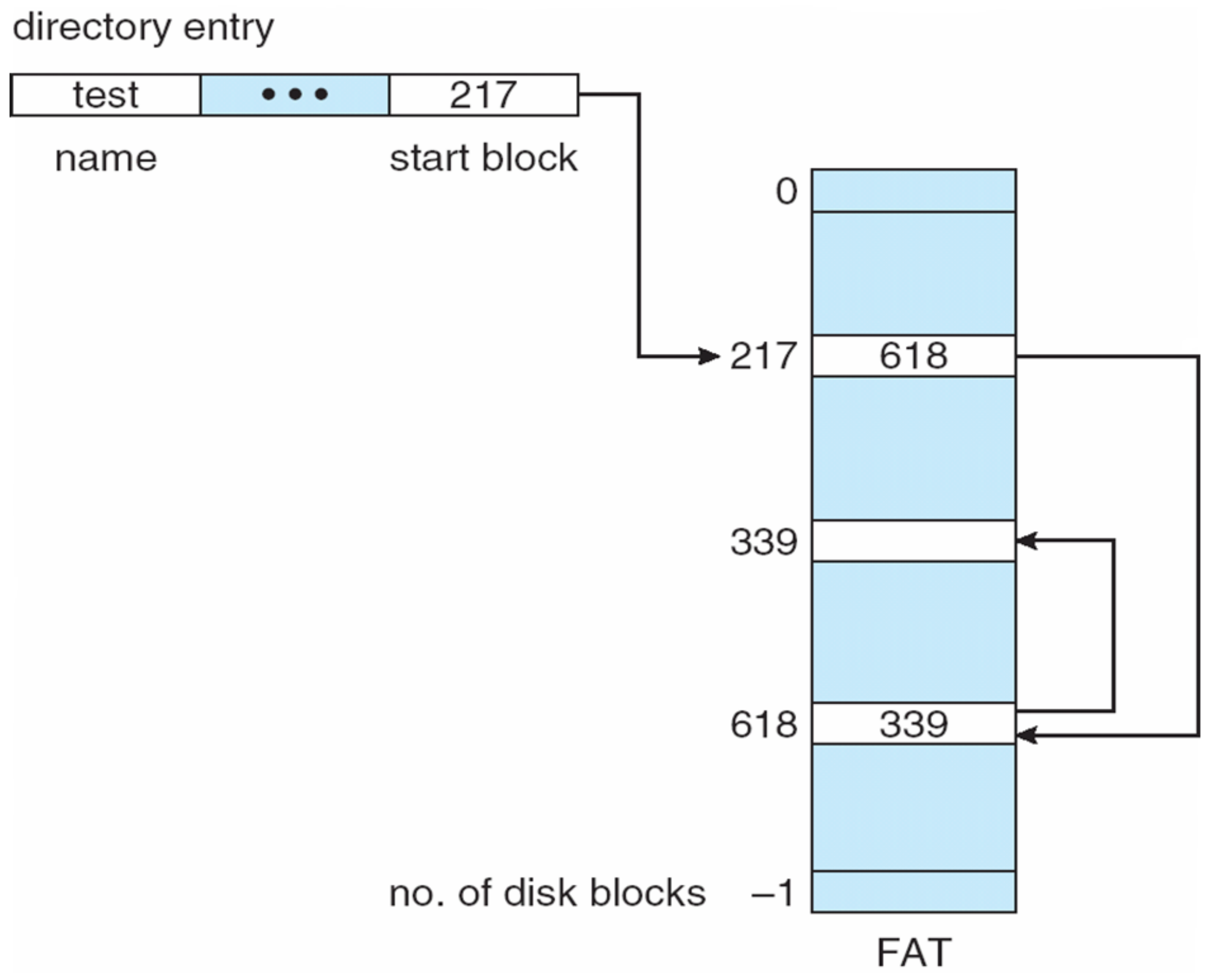
Linked Allocation

- 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)
  - *locating a file block can take many I/Os and disk seeks*
- FAT (File Allocation Table) uses linked allocation

Linked Allocation



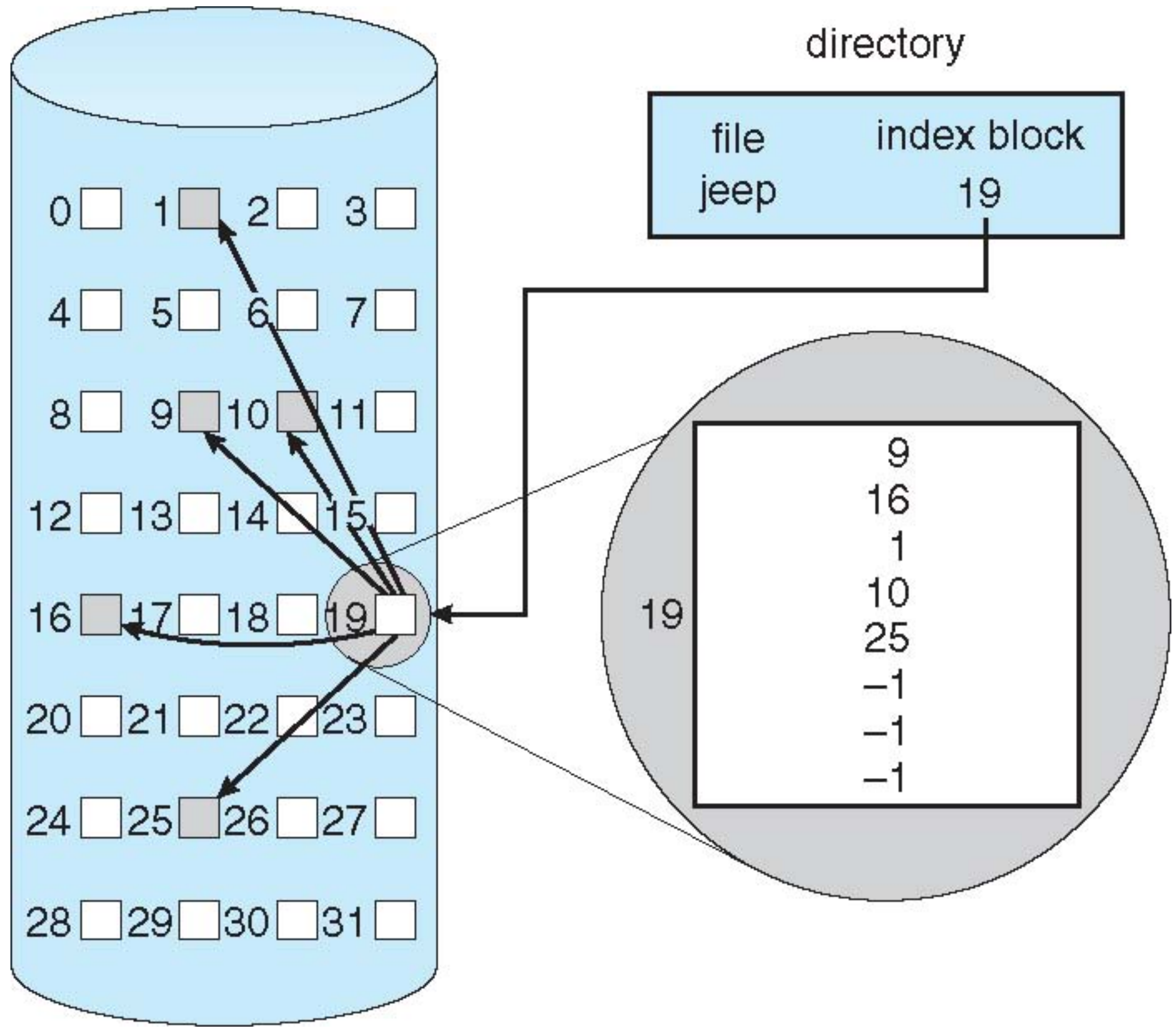
File-Allocation Table (FAT)



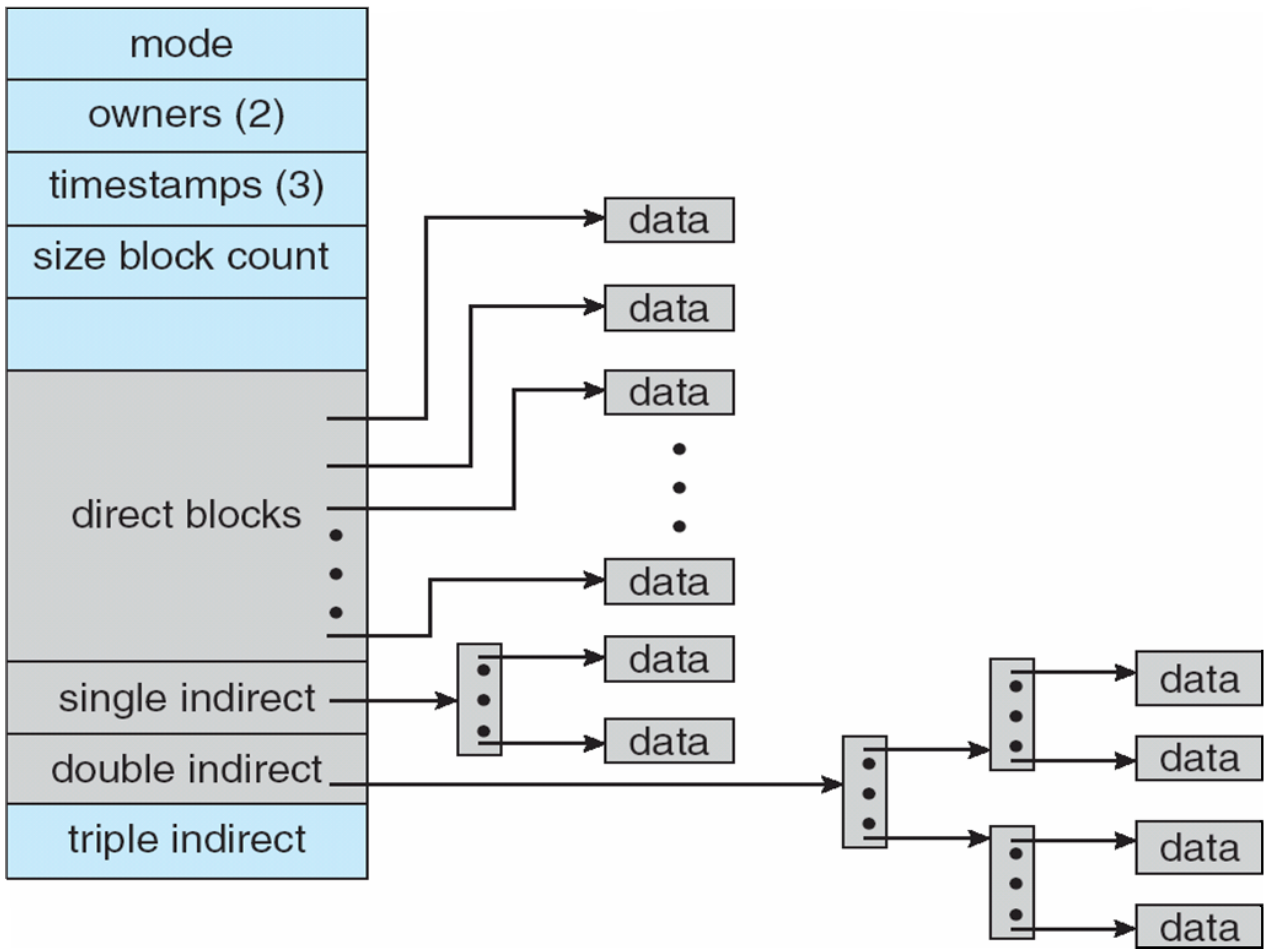
Indexed Allocation

- 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
- Need a method to allocate index blocks
  - linked index blocks
  - multiple-level index blocks (e.g., 2-level)
  - combined scheme

Indexed Allocation



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 MIPS / 250 = 630 million instructions during one disk I/O
  - fast SSD drives provide 60,000 IOPS
    - 159,000 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
- Many allocation methods:
  - bit vector or bit map
  - linked free space
  - …

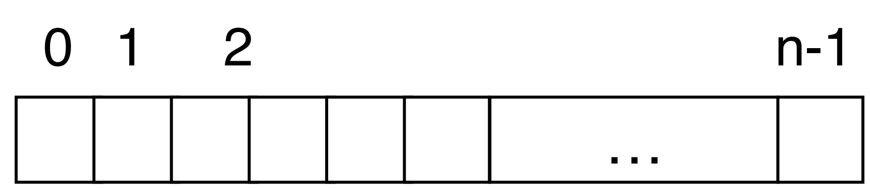
Bitmap Free-Space Management

- 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

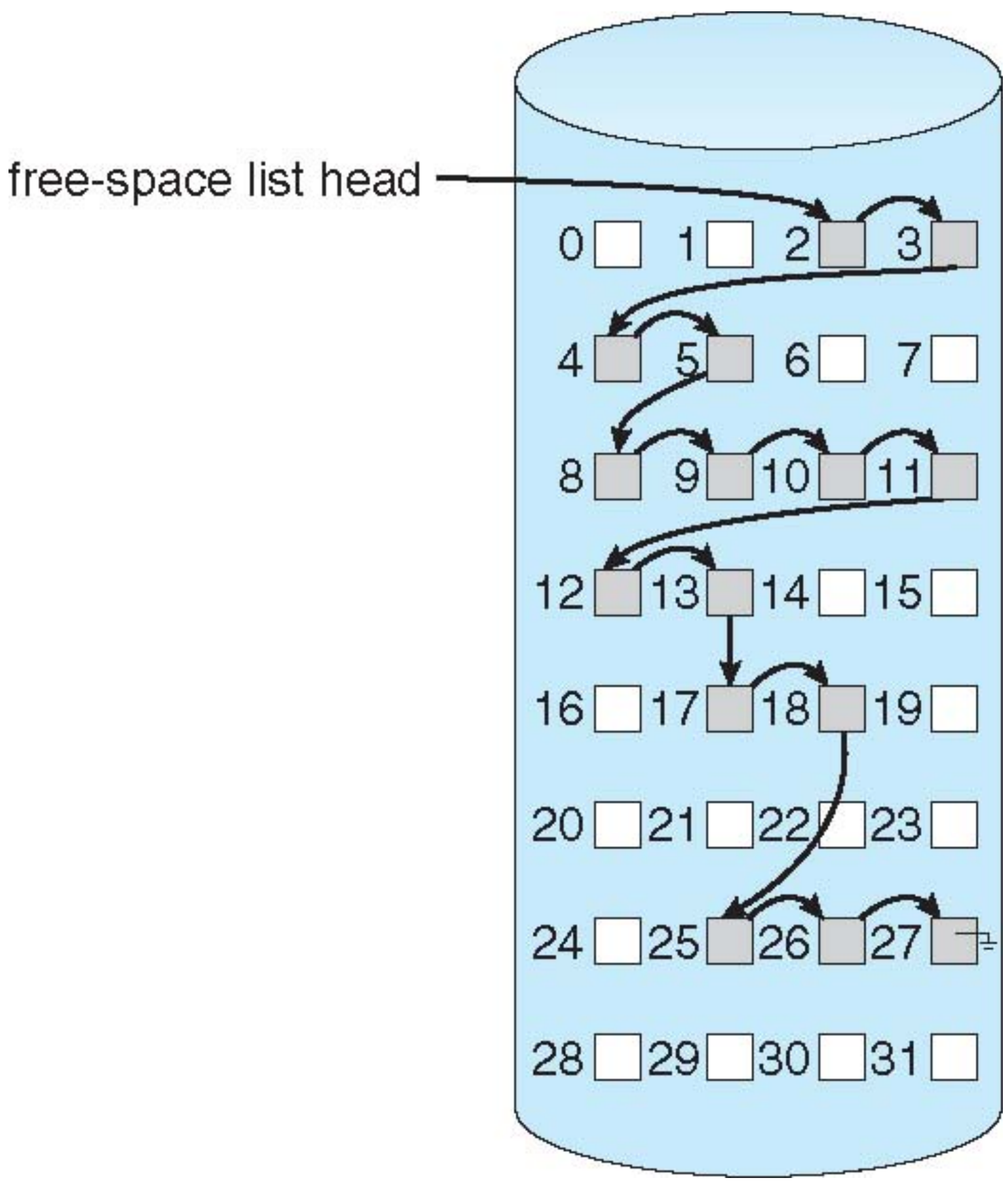


$$\text{bit}[i] = \begin{cases} 1 \rightarrow \text{block}[i] \text{ free} \\ 0 \rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

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
  - no need to traverse the entire list (if # free blocks recorded)

Linked Free Space



Linked Free-Space

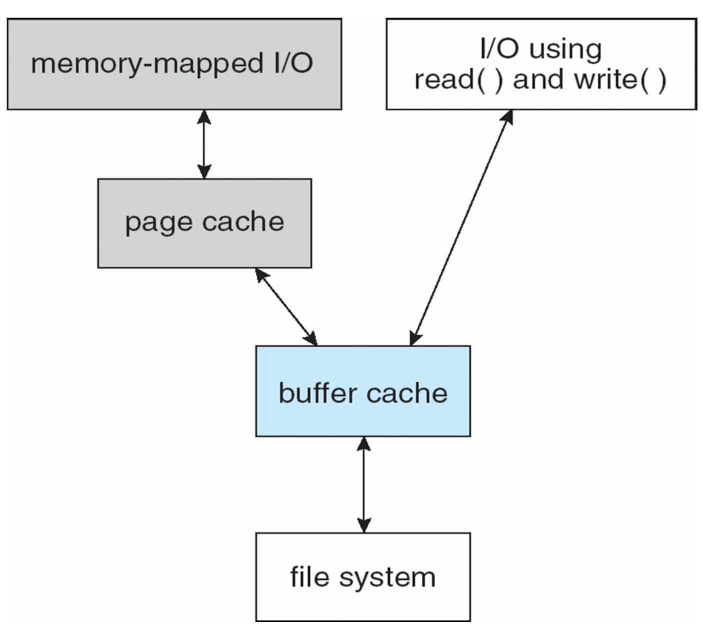
- Simple linked list of free-space is inefficient
  - one extra disk I/O to allocate one free block (disk I/O is extremely slow)
    - allocating multiple free blocks require traverse the list
  - difficult to allocate contiguous free blocks
- **Grouping**: use indexes to group free blocks
  - store 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

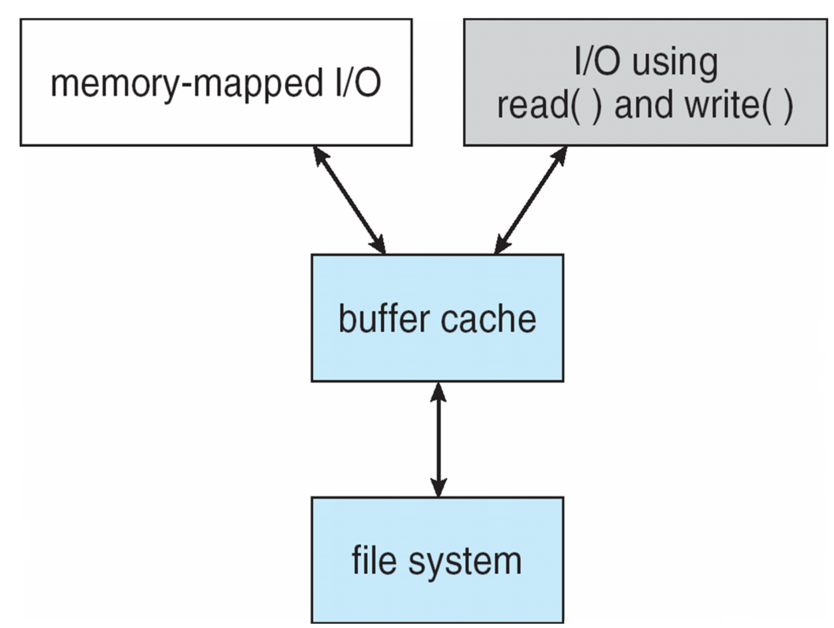
File System Performance

- 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
  - …
- 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

Page Cache and MMIO

- 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



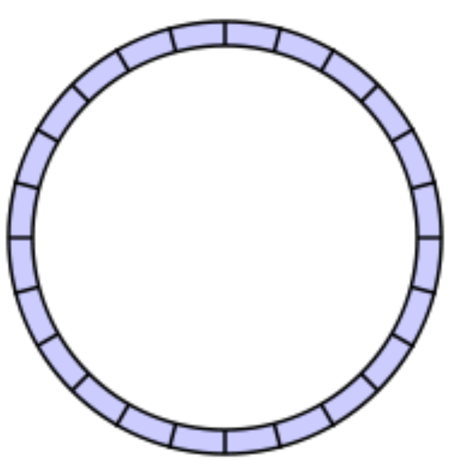


Recovery

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

Log Structured File Systems

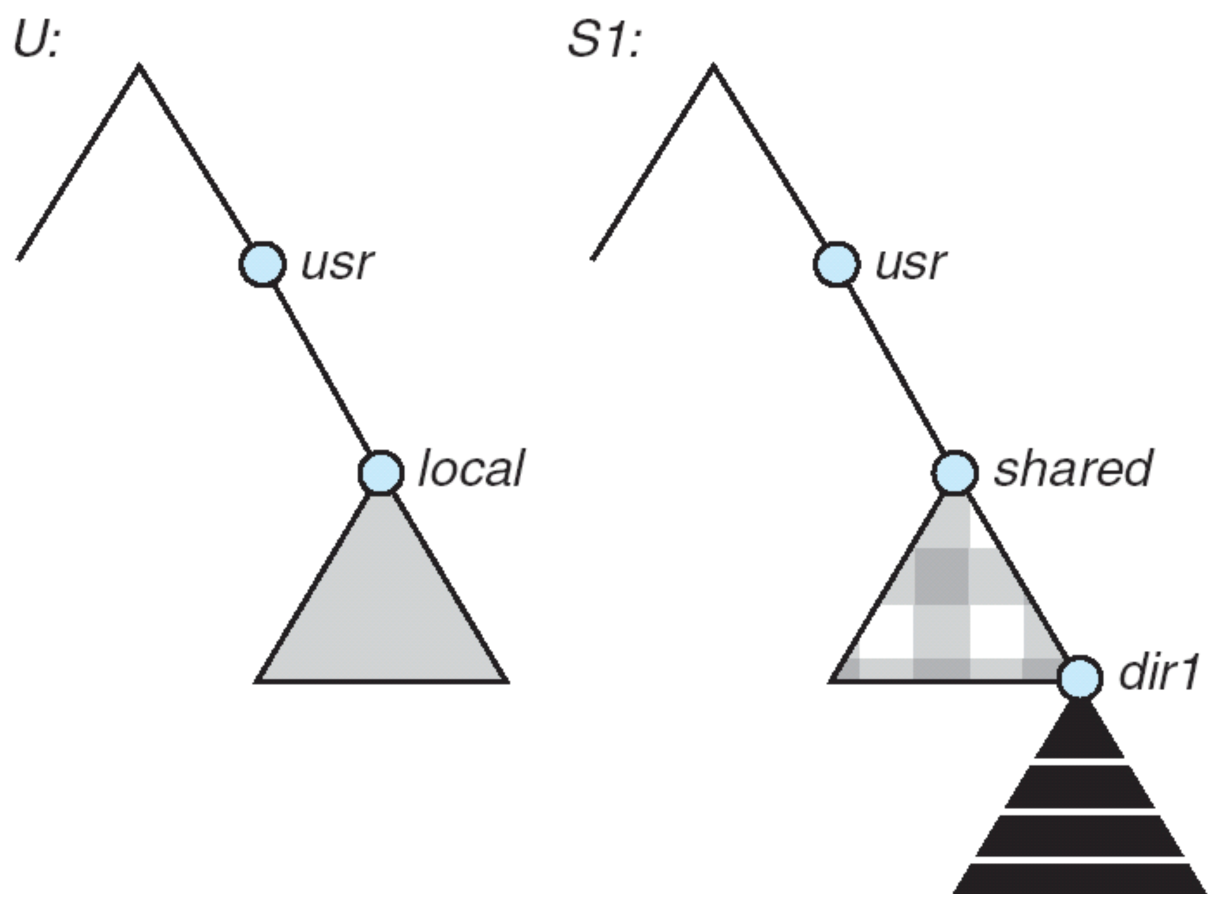
- In LSFS, metadata for updates sequentially written to a **circular log**
  - once changes written to the log, it is committed, and syscall can return
    - log can be located on the other disk/partition
  - meanwhile, log entries are **replayed** on the file system to actually update it
    - when 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



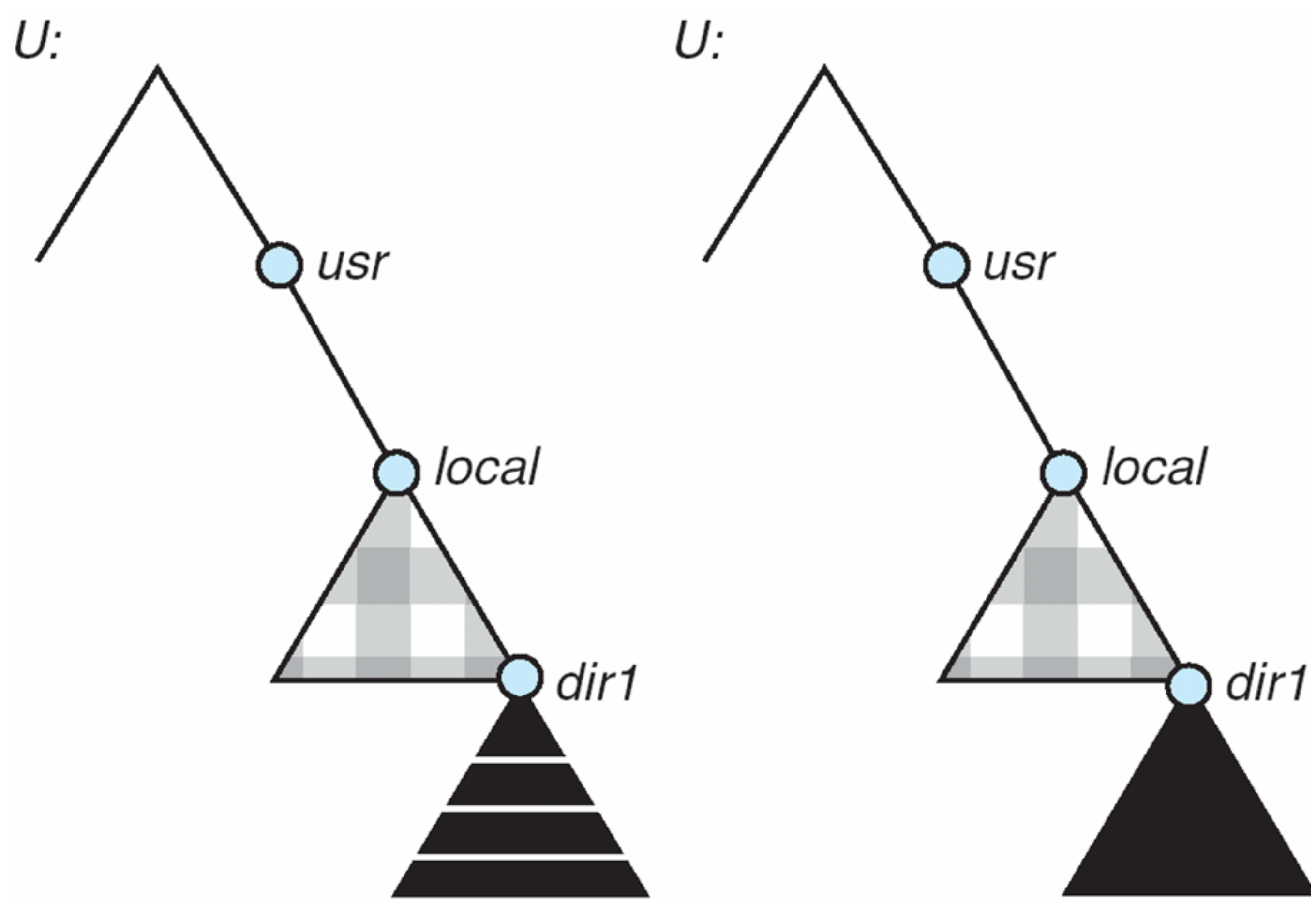
Example: Network File System (NFS)

- NFS is a software system for accessing **remote files**
  - support both LAN and WAN
  - implementation is a part of the Solaris and SunOS
    - for Sun workstations, using UDP and Ethernet
- NFS transparently enables sharing of FS on independent machines
  - each machine can have its own (different) file system
  - a remote directory is **mounted** over (and cover) a local file system directory
    - mounting operation is not transparent
    - the host name of the remote directory has to be provided
  - designed for heterogeneous environment with the help of **RPC**
    - different machine architecture, OS, or network architecture
  - to improve performance, NFS employs many caches
    - directory name cache, file block cache, file attribute cache…

NFS Client and Servers



After Client Mounts



(a) (b)

NFS Mount Protocol

- Mount establishes initial connection between **server** and **client**
  - mount request includes the server name and remote directory name
  - mount request is mapped to a RPC to the server
  - server has an **export list**
    - local file systems that server exports for mounting
    - names of machines that are permitted to mount them
  - if request allowed by the export list the server returns a file handle
    - a **file handle** is a number to identify the mounted directory within server
- A remote FS can be mounted over a local FS, or a remote FS (cascading mount)

NFS Protocol

- NFS provides a set of **RPCs** for remote file operations
  - read and write files
  - read/search a set of directory entries
  - manipulate links and directories
  - access file attributes
- NFS servers are **stateless**
  - each request has to provide a full set of arguments (new NFS v4 is stateful)
- Updates must be committed to disk before server returns to the client
  - caching is not allowed
- NFS protocol does not provide concurrency-control mechanisms

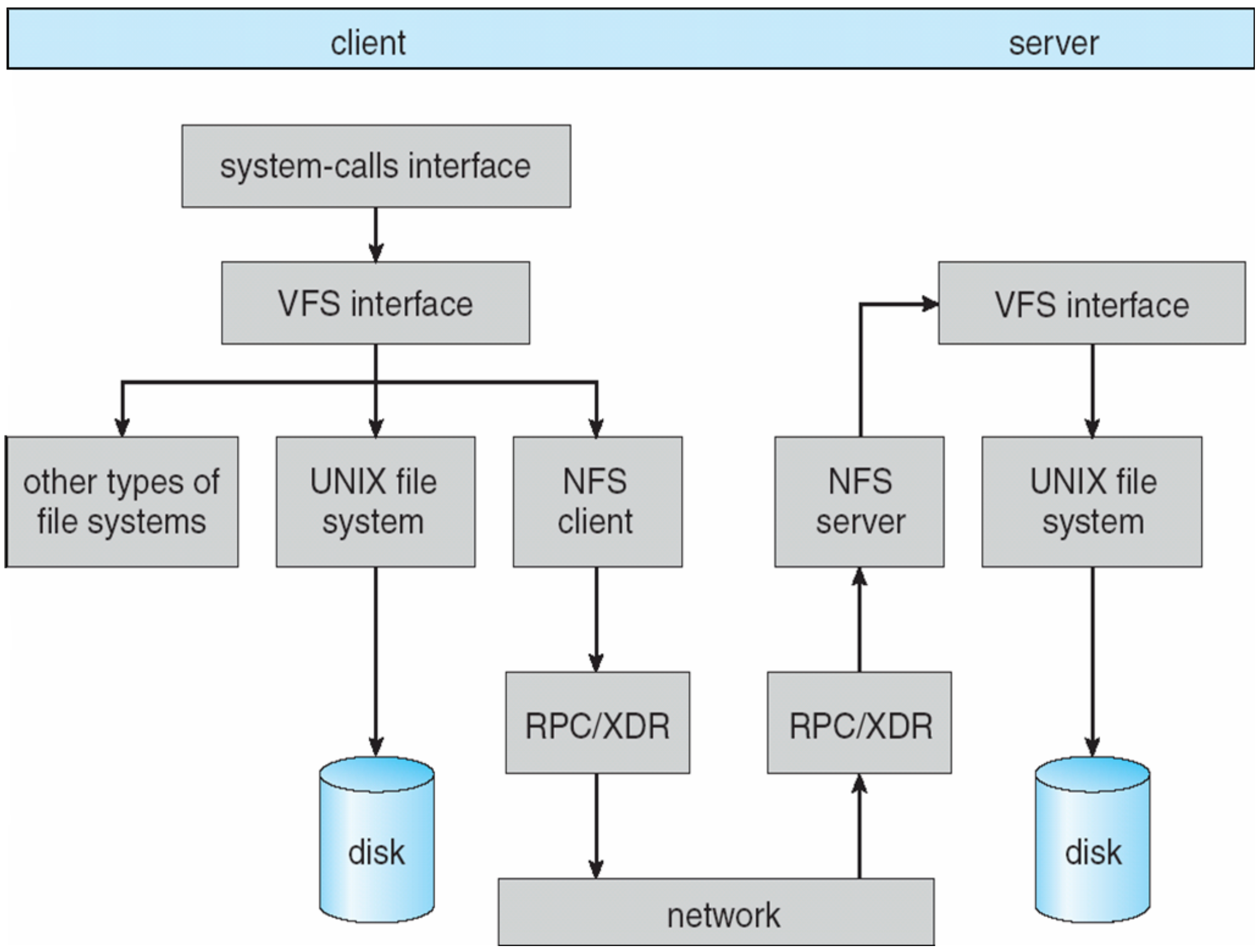
NFS Remote Operations

- One-to-one correspondence between UNIX syscalls and NFS RPCs
  - except opening and closing files that needs special parameter
- NFS employs buffers/caches to reduce network overhead
  - file-blocks cache: caches data of a file
  - file-attribute cache: cache the file attributes
  - cached data can only be used if fresh (check with the server)

Integration of NFS

- Syscall API is based on **virtual file system** (VFS), no need to change
  - open, read, write, and close calls, and file descriptors
- VFS layer dispatches file access to NFS
  - VFS calls the NFS protocol procedures for remote requests
  - VFS does not know/care whether file system is local or remote
- NFS service layer actually implements the NFS protocol

Integration of NFS



End of Chapter 11