File System Implementation

November 8, 2017

File Structure

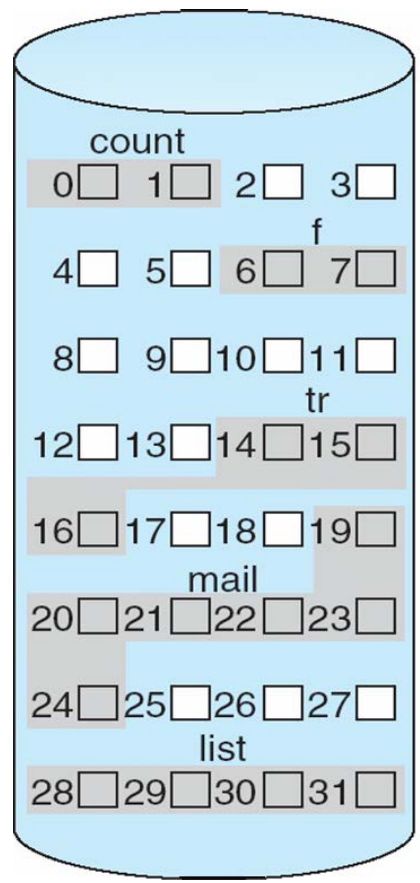
- A file is composed of contiguous logical blocks
  - Each logical block has a fixed size
- Secondary storage is composed of physical blocks each has the same size as a logical block
- When a file is stored, its logical blocks are stored to physical blocks
  - Physical blocks for a file may not be contiguous

Allocation Methods

- An allocation method refers to how disk blocks are allocated for files:
  - **Contiguous allocation**
  - **Linked allocation**
  - **Indexed allocation**

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
- Simple – only starting location (block #) and length (number of blocks) need to be recorded

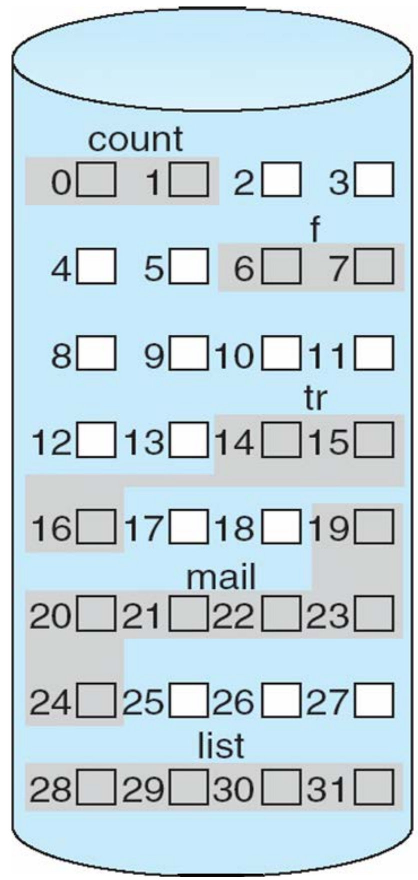


directory

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

Contiguous Allocation

- Random access is easy to implement
- Waste of space
  - External fragmentation problem
- Allocation algorithms:
  - First-fit, best-fit, worst-fit
- Files cannot grow



directory

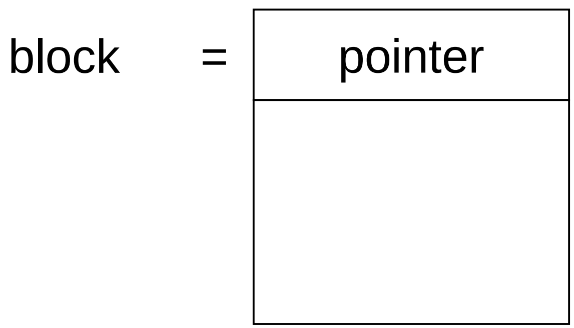
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-Based Systems: To allow files grow

- Many newer file systems (i.e. Veritas File System) use a modified contiguous allocation scheme
- A contiguous chunk of space is allocated initially
- If the amount is not large enough, another chunk of contiguous space, known as extent, is added
- A file consists of one or more extents.
  - For each extend, the location of the first block and the block count are recorded.

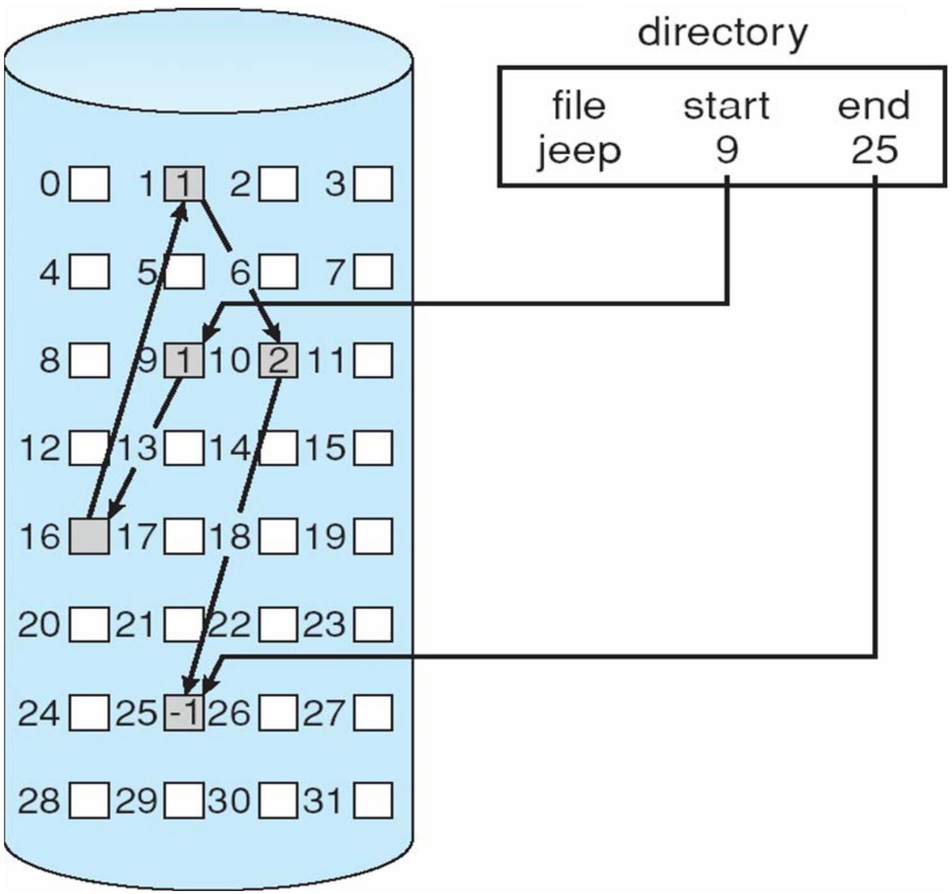
Linked Allocation

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

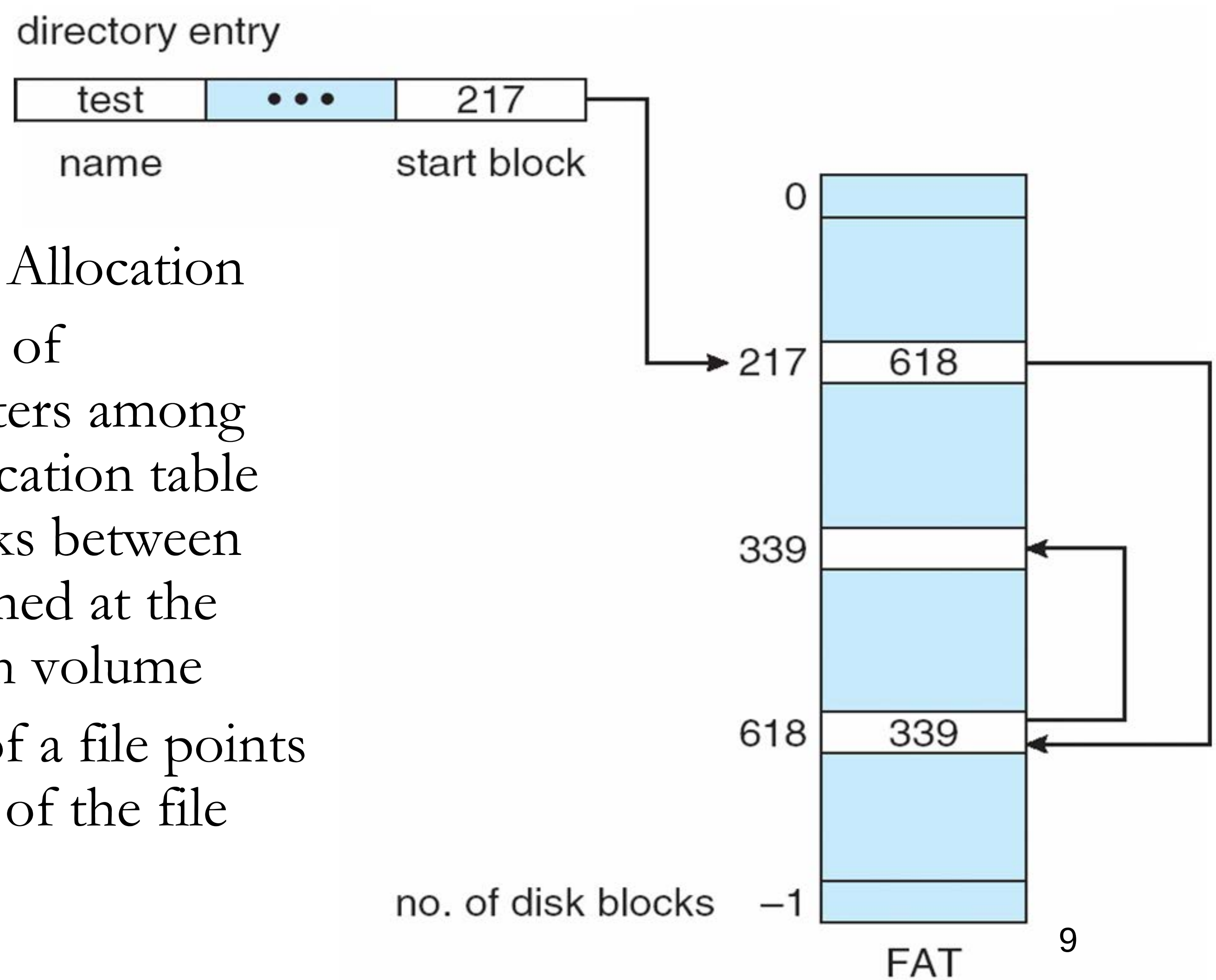


Linked Allocation

- Simple – need only starting address
- No external fragmentation
- No random access – have to traverse block by block
- Reliable? – What if a block is damaged?



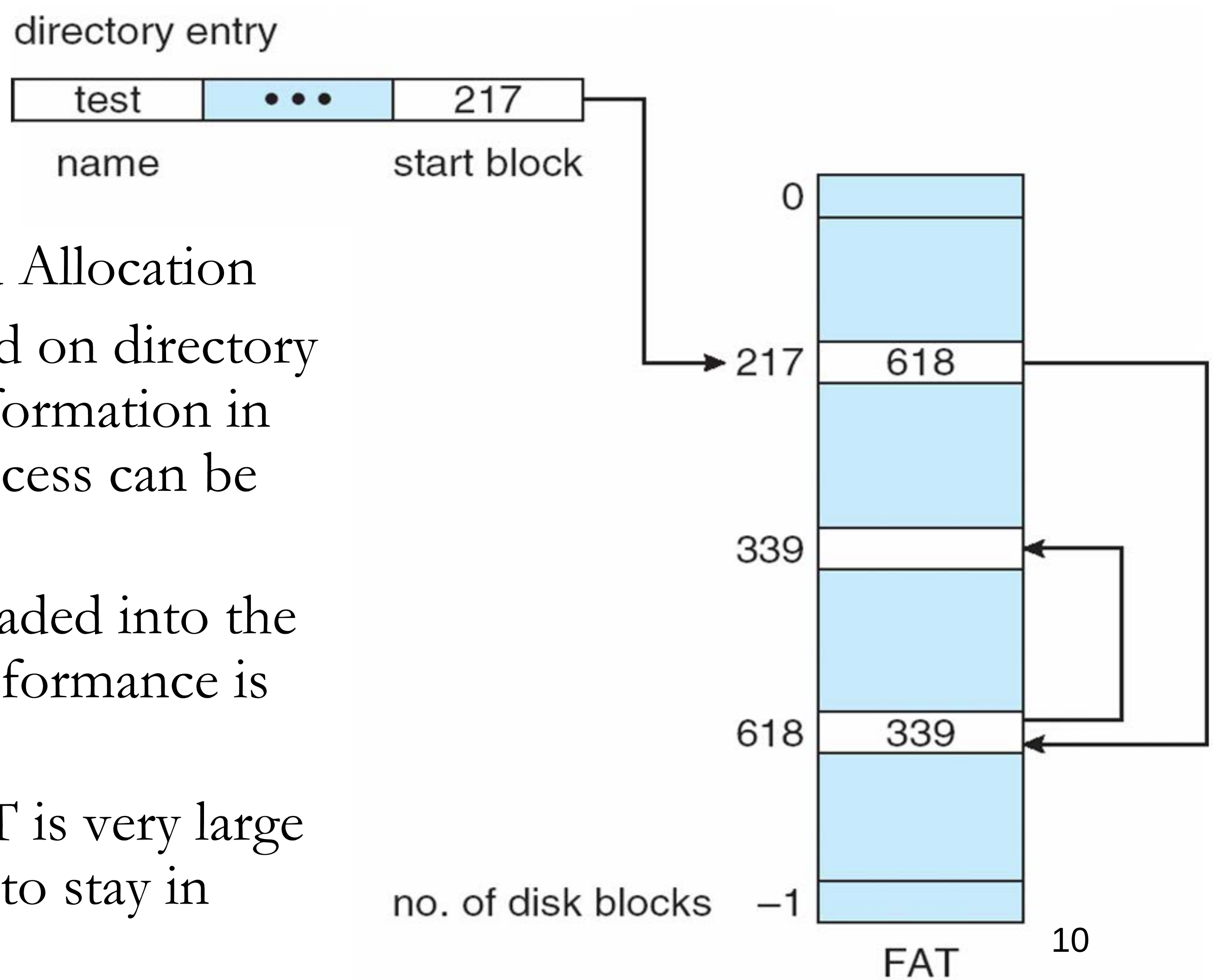
File-Allocation Table



A variant of Linked Allocation

- Key idea: instead of distributing pointers among blocks, a file-allocation table recording the links between blocks is maintained at the beginning of each volume
- Directory entry of a file points to the first block of the file

File-Allocation Table

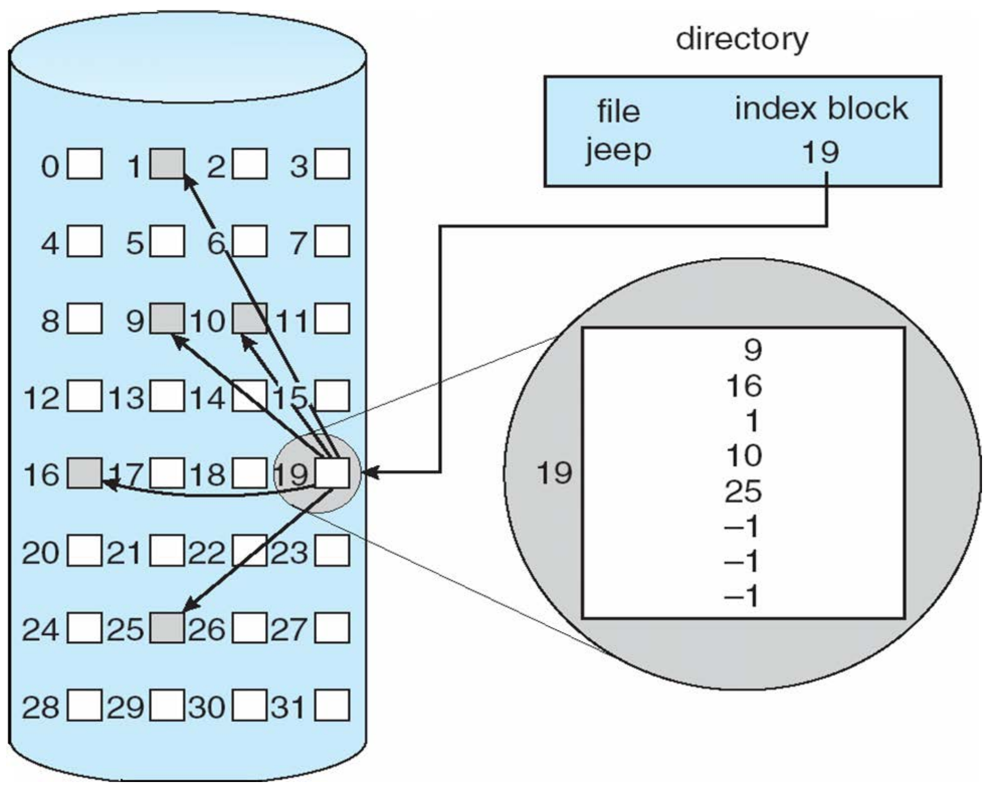


A variant of Linked Allocation

- Advantage: based on directory entry and the information in FAT, random access can be performed
- If the FAT is loaded into the memory, the performance is better
- What if the FAT is very large and not suitable to stay in memory?

Indexed Allocation

- For each file, the blocks it uses are listed in a block called *index block*. (similar to page table)
- Directory entry for the file points to the index block



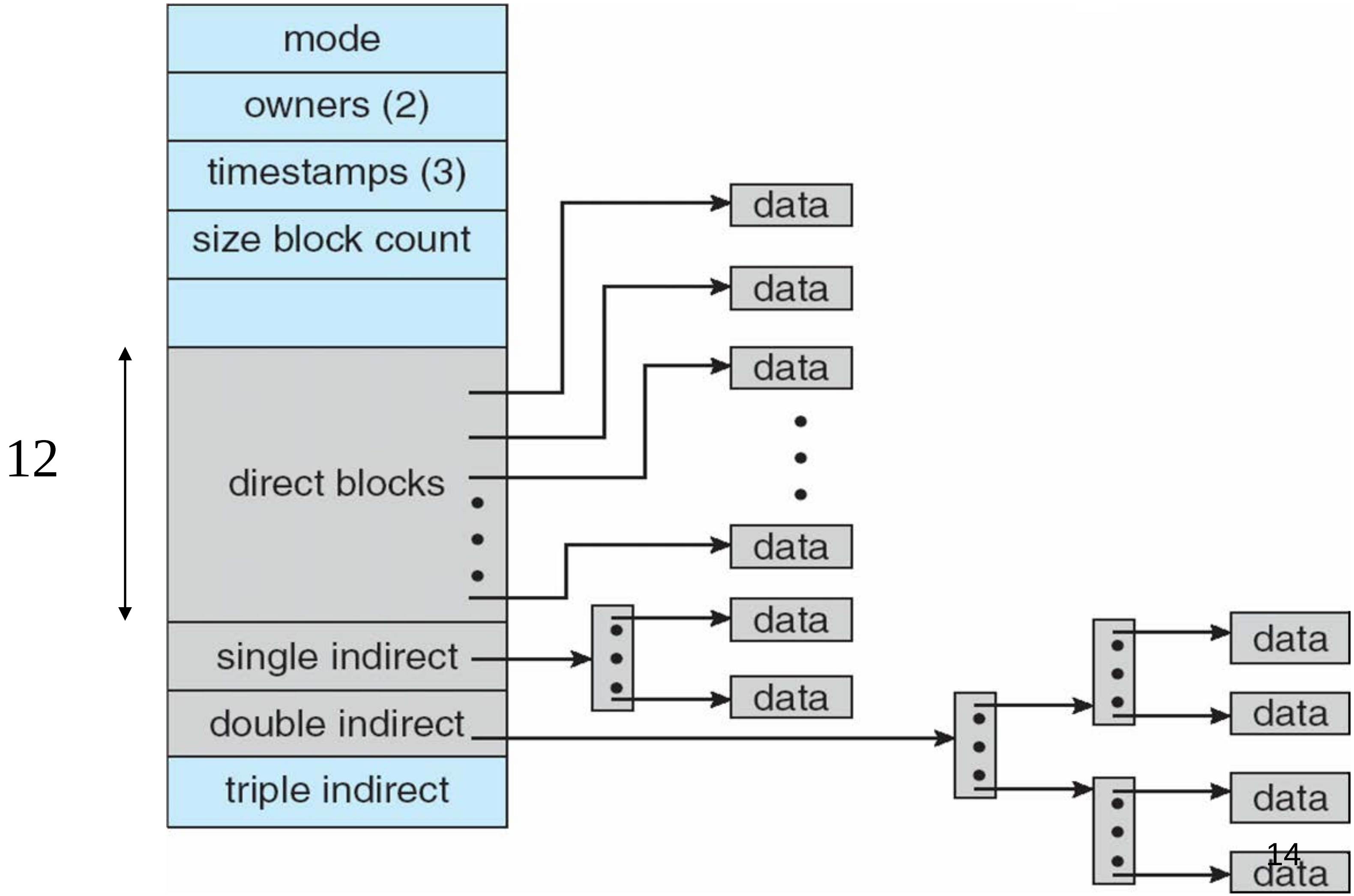
Indexed Allocation

- Need an index table for each file
- Random access
- No external fragmentation
- File size can change

Indexed Allocation

- Scalability
  - Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.
  - What if the max size of a file exceeds 256K words? Multiple blocks needed for storing index table.
    - Strategy 1: Linked index blocks
    - Strategy 2: Multilevel index (similar to multi-level page table)

Combined Scheme: UNIX (4K bytes per block)



Free-Space Management

- Strategy 1: Bit vector (n blocks)

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

Finding the first free block: 1. Scan the bits sequentially until reaches the first non-0 word. 2. Calculate the free block number:

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

Free-Space Management

- Bit map requires extra space
  - Example:

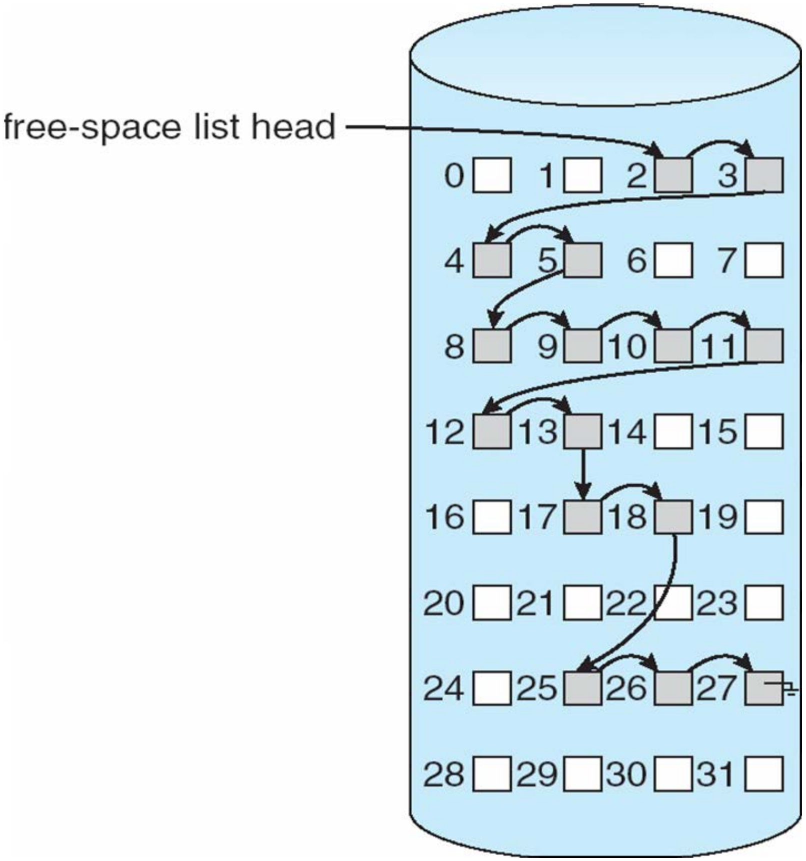
    block size = 2^{12} bytes

    disk size = 2^{30} bytes (1 gigabyte)

    $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)

Free-Space Management

- Strategy 2: Linked list of free blocks (free list)
  - Cannot get contiguous space easily
  - No waste of space



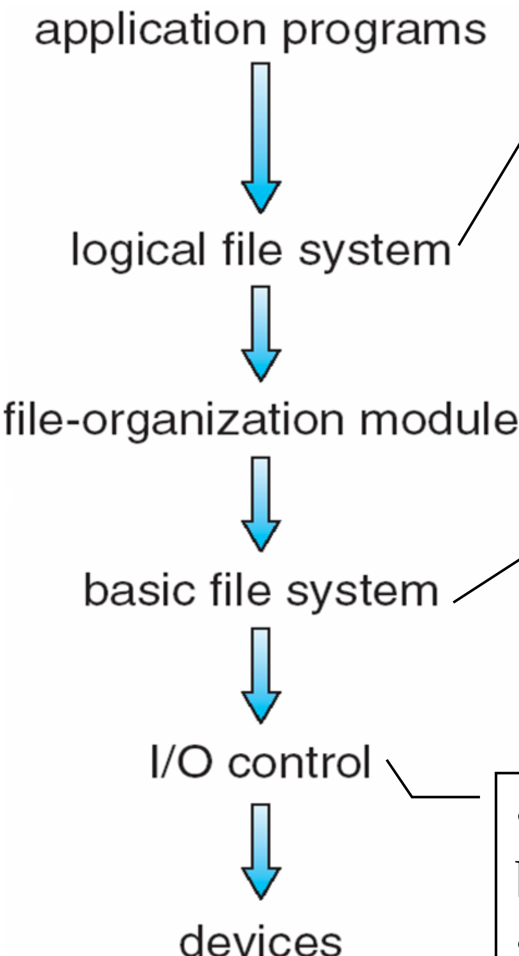
Free-Space Management

- Strategy 3: Grouping
  - A modification of the linked list
  - The addresses of n free blocks are stored in the first free block
    - The first n-1 of these blocks are actually free blocks
    - The n-th free block contains another (next) n free blocks
    - …

Free-Space Management

- Strategy 4: Counting
  - Intuition: free blocks form a set of clusters of contiguous free blocks, especially when contiguous allocation is used. Example: free blocks are 0, 1, 2, 5, 6, 7, 8, 9, 100,101,102, …
  - The free-space list is composed of tuples Ti, where each Ti =<Bi, Ni>, Bi is the first block of a set of contiguous free blocks and Ni is the number of such free blocks. Example: (0,3), (5,5), (100,3), …

Layered File Management System



• manage metadata information (i.e., all of the file-system structure except the actual data)

•Translate logical blocks to physical blocks
•Manage free space

•Issue commands to I/O control level
•Manage memory buffers and caches to hold file-system information, directory, and data blocks.

•Device drivers & interrupt handlers: transfer information between the main memory and the disk system
•Translator: accept high-level command such as “retrieve block 123” and output low-level, hardware-specific instructions

On-disk Structures for File System

- Boot control block (per volume)
  - Containing info needed by the system to boot an OS from this volume
  - Called *boot block* in UFS (unix FS), *partition boot sector* in NTFS
  - Can be empty if the volume does not contain an OS
- Volume control block (per volume)
  - Containing volume (or partition) details: # of blocks in the partition, size of the blocks, free-block count and free-block pointers, free-FCB (file control block) count and free-FCB pointers
  - Called *superblock* in UFS, *master file* table in NTFS

On-disk Structures for File System

- Directory structure (per file system)
  - In UFS, this includes file names and associated *inode* numbers.
  - In NTFS, it is stored in the master file table
- Per-file FCB (file control block)
  - Containing many details about the file
  - Having a unique identifier number to allow association with a directory entry
  - In NTFS, this is information is stored in the master file table, which uses a relational database structure, with a row per file.

In-memory Structures for File System

- In-memory structures
  - Purposes: to facilitate file-system management; performance improvement via caching
  - The data are loaded when a file system is mounted, updated during file-system operations, and discarded when the file system is dismounted.

In-memory Structures for File System

- Examples
  - In-memory mount table: info about each mounted volume
  - In-memory directory-structure cache: directory information of recently accessed directories
  - System-wide open-file table: a cope of the FCB of each open file as well as other information (e.g., the open count)
  - Per-process open-file table: a pointer to the entry in the system-wide open-file table, as well as other information (e.g., current position, access rights)
  - Buffers: hold file-system blocks when they are being read from disk or written to disk

Create a File

- Application program calls the logical file system
- Logical file system:
  - Allocating a new FCB
  - Reading the appropriate directory into memory, updating it with the new file name and FCB, and writing it back to the disk
  - Logical file system use lower levels to implement the above

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

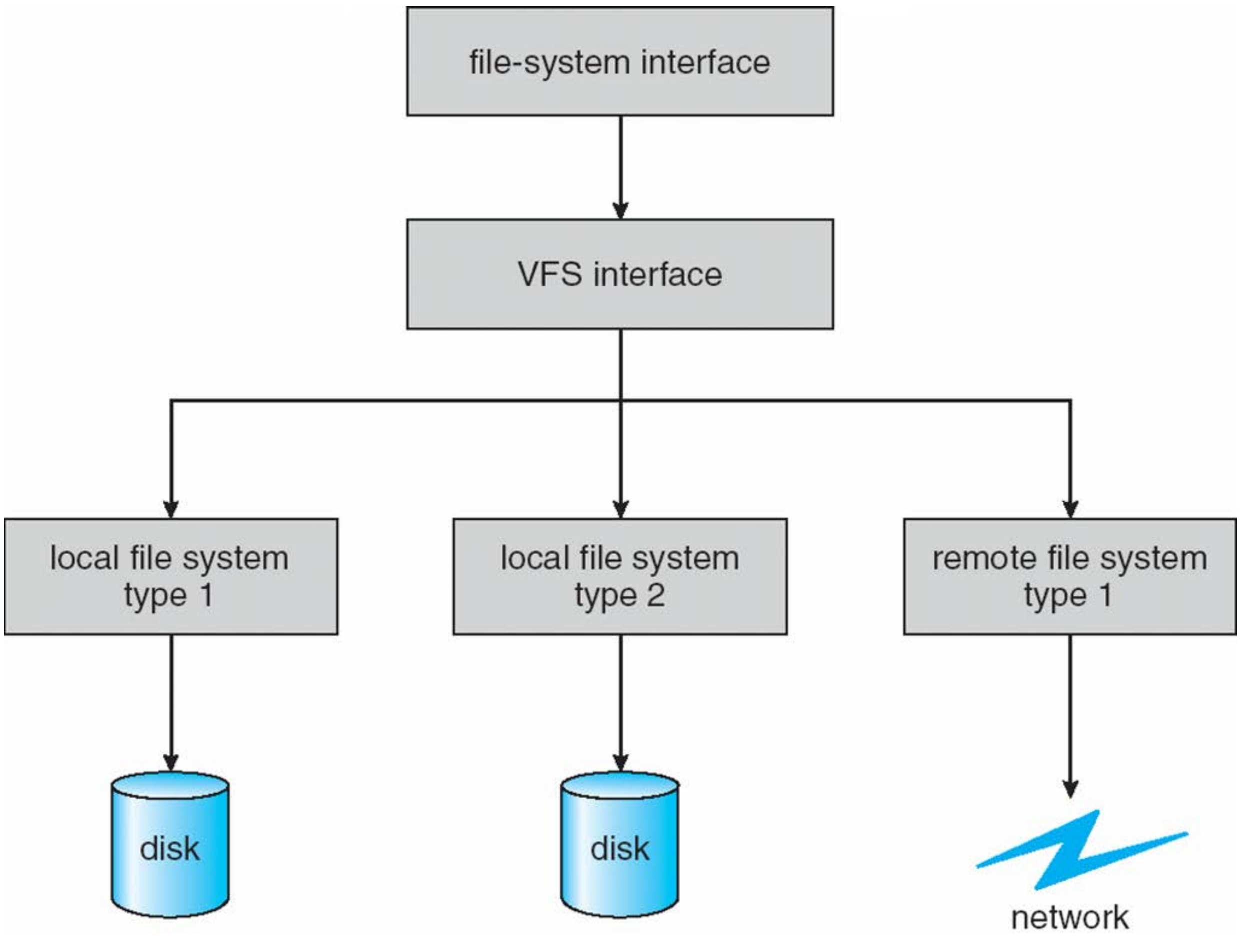
Open a File

- The open() call passes a file name to the logical file system
- Logical file system:
  - Searching the system-wide open-file table to see if the file is already in use by another process
    - If not, (i) the directory structure is searched (in the disk or in the memory cache), and (ii) an entry of the system-wide open-file table, with the FCB copied, is created
  - A per-process open-file table entry is created
    - pointing to the existing system-wide open-file table
    - Containing other fields: pointer to the current location in the file; access mode in which the file is open; …

ext3: A native Linux file system

- Space allocation
  - index-based allocation (like i-node)
  - attempt to allocate contiguous clusters of blocks to store the data of a file
  - de-fraction
- Journaling
  - log operations before really performing the operations
  - higher reliability
  - lower latency

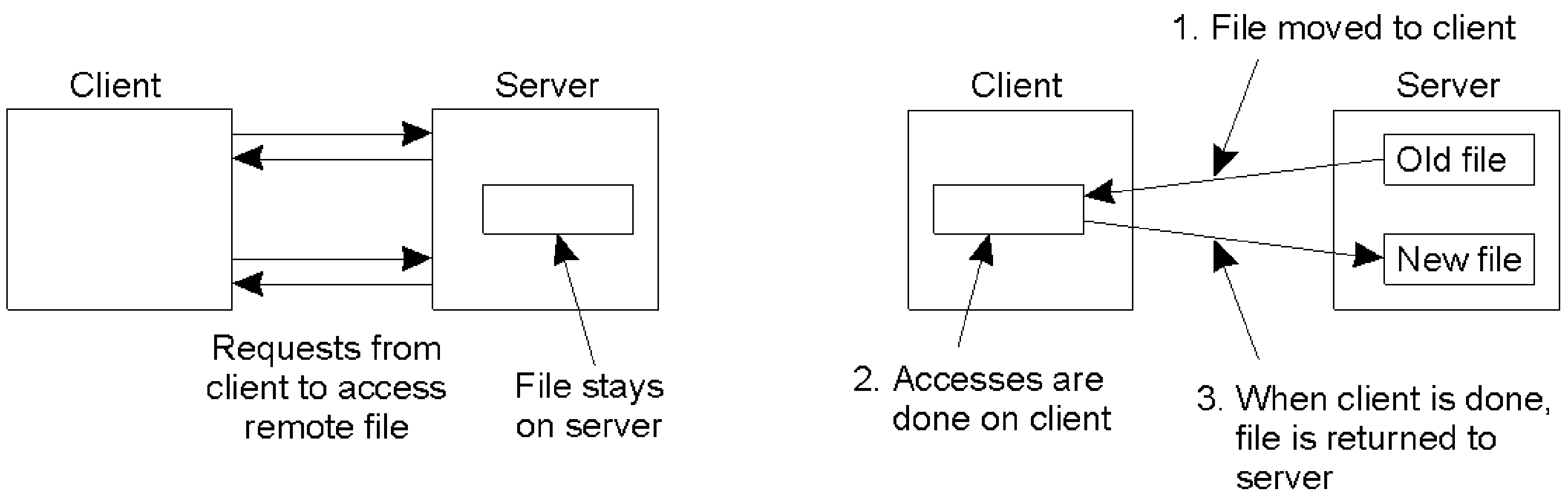
Schematic View of Virtual File System



Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

File Access Cross Different Computers



a) The remote access model.
b) The upload/download model

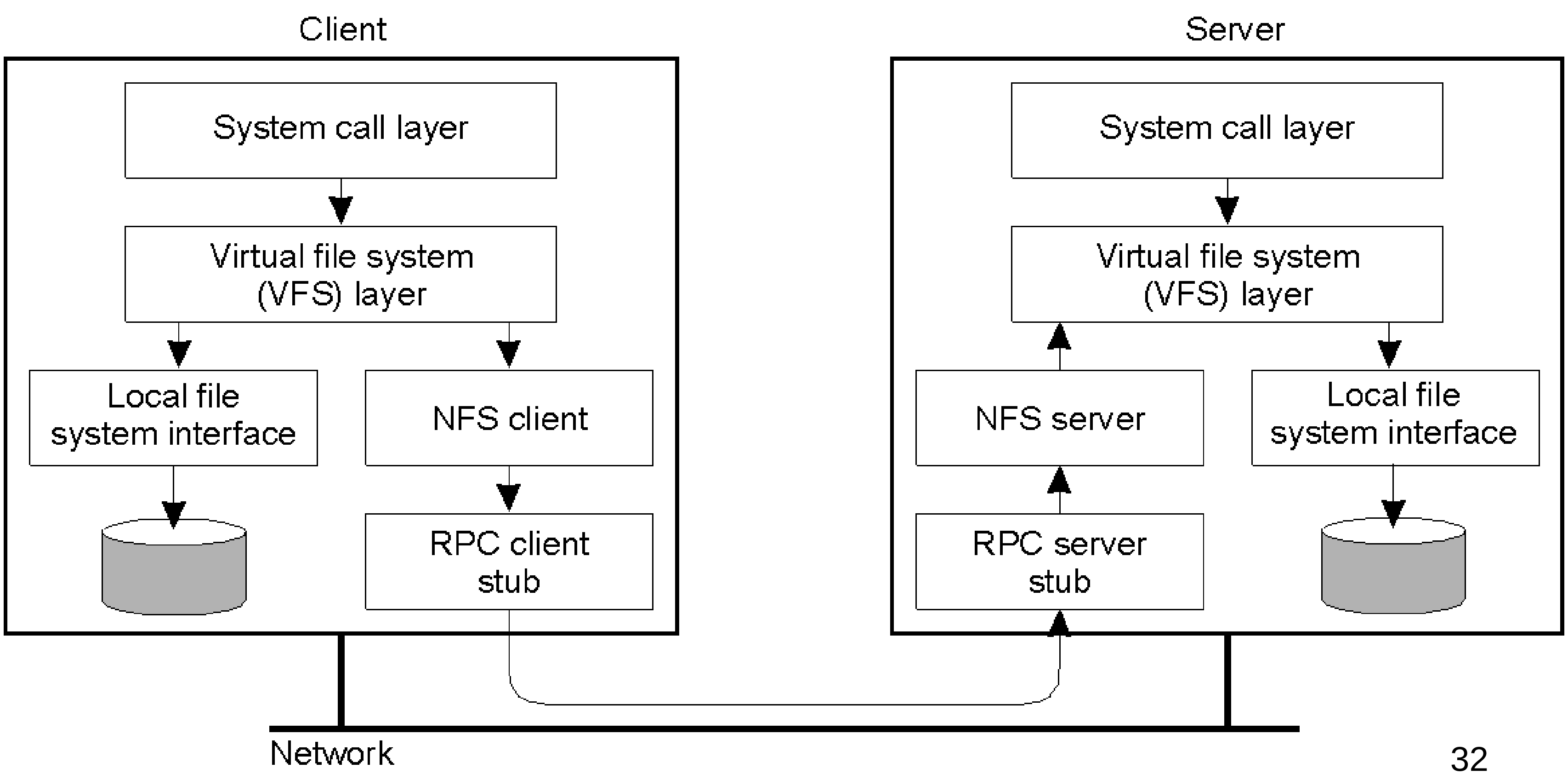
The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)

- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet)

NFS Architecture

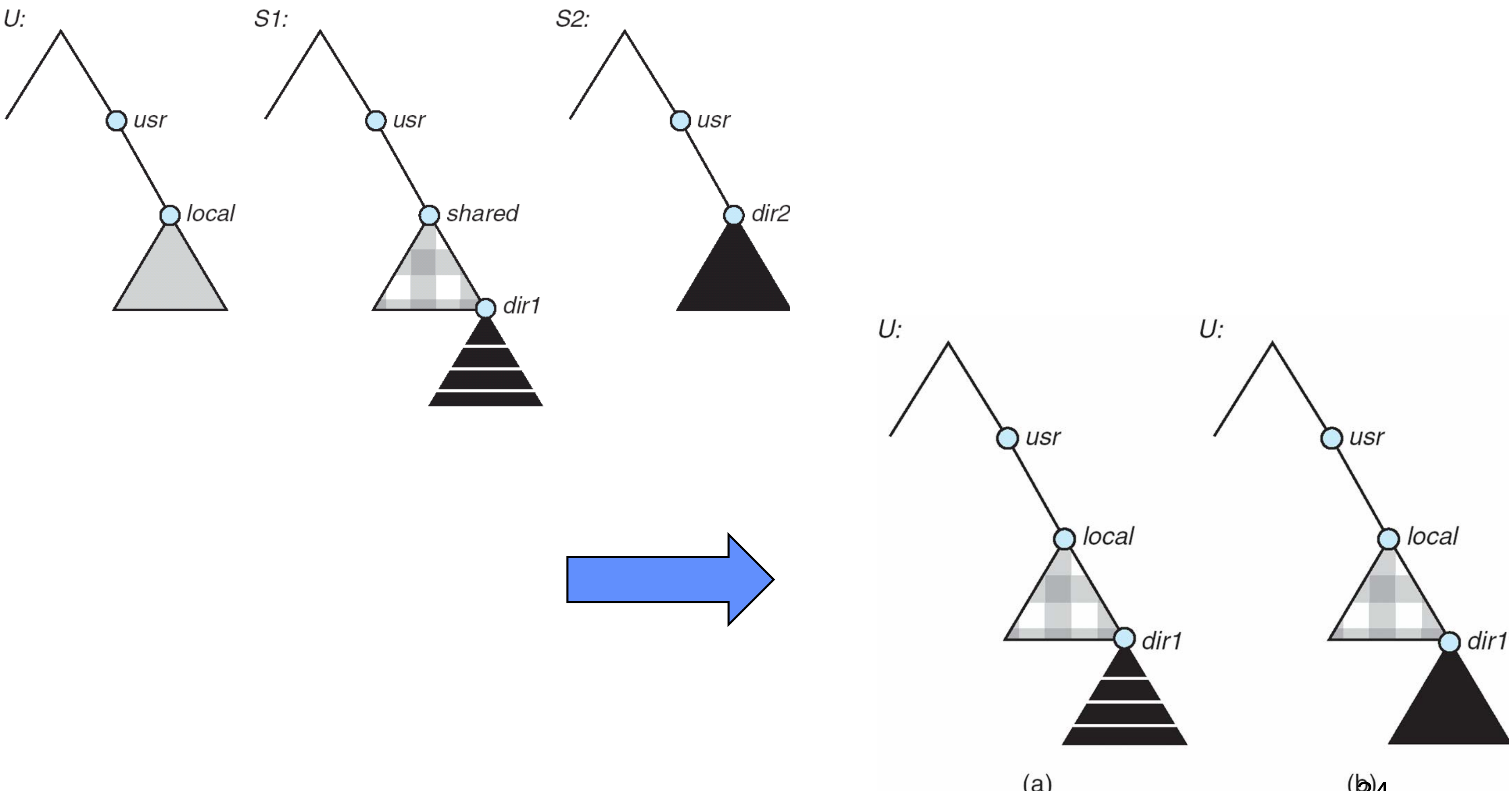
- The basic NFS architecture for UNIX systems.



NFS Mount Protocol

- Establishes initial logical connection between server and client
- Client: Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
- Server:
  - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
  - 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
- The mount operation changes only the user's view and does not affect the server side

Mounting Examples



NFS Protocol

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

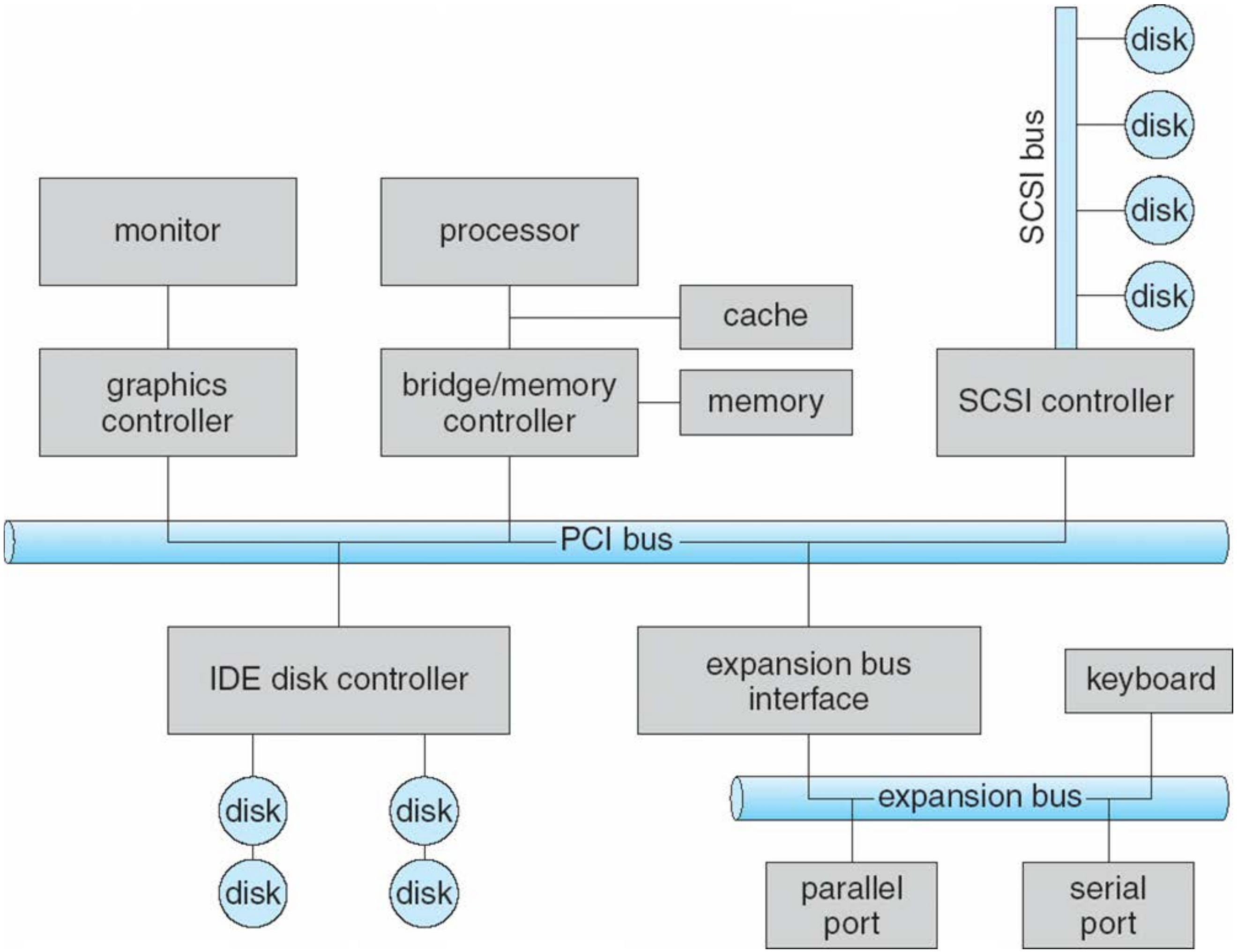
I/O System

Nov 30/Dec 2, 2016

Overview

- Two main jobs of a computer system
  - input/output (I/O)
  - computing (processing)
  - I/O is often the major job
- Role of OS for I/O
  - Manage and control I/O operations and I/O devices
- Roadmap
  - I/O Hardware
    - How do I/O devices connect and work with main computer components (CPU, memory)?
  - Kernel I/O Subsystem
    - What basic services are provided by the kernel to bridge the gap between I/O hardware interface and application interface

I/O Hardware



- Incredible variety of I/O devices
- I/O devices connect to CPU/Memory via
  - Port (e.g., series port)
  - Bus (daisy chain or shared direct access)

How CPU communicates with I/O devices

- Device controller
  - Each I/O device is associated with a controller
  - The controller could be a part of a port, a bus, or a device
- CPU (host) interacts with a device by reading/writing registers of the controller of the device
  - Data-in register: read by the host (CPU) to get input
  - Data-out register: written by the host to send output
  - Status register: read by the host; indicating following states
    - whether the current command has completed
    - whether a byte is available to be read from the data-in register
    - whether a device error has occurred
  - Control register: written by the host to start a command or to change the mode of a device (bit patterns are defined to represent commands)

How CPU communicates with I/O devices

- Ways to access registers
  - Using special I/O instructions
    - Specifying the transfer of a byte or word to/from an I/O port address
    - Example from the Intel architecture: `out/in 0x21,AL`
  - Memory-mapped I/O
    - Registers are mapped into the address space of the CPU
    - The CPU executes I/O requests using load/store instructions
  - Hybrid of the above two
    - Some devices are controlled through I/O instructions, others are through accessing memory-mapped region.
    - Example: PC's graphics controller has I/O ports for basic control operations but has a large memory-mapped region to hold screen contents.

Protocols for Interaction between the CPU (host) and I/O devices

- Polling
- Interrupts
- Direct memory access

Polling

- I/O device controller indicates its state through the busy bit in the status register
  - busy bit is set (i.e., it is "1") – the device is busy with working
  - busy bit is clear (i.e., it is "0") – the device is ready to accept the next command
- The host indicates its state through the command-ready bit in the command register of an I/O device controller
  - command-ready is set, if the host has written a command and waits for the device controller to execute it

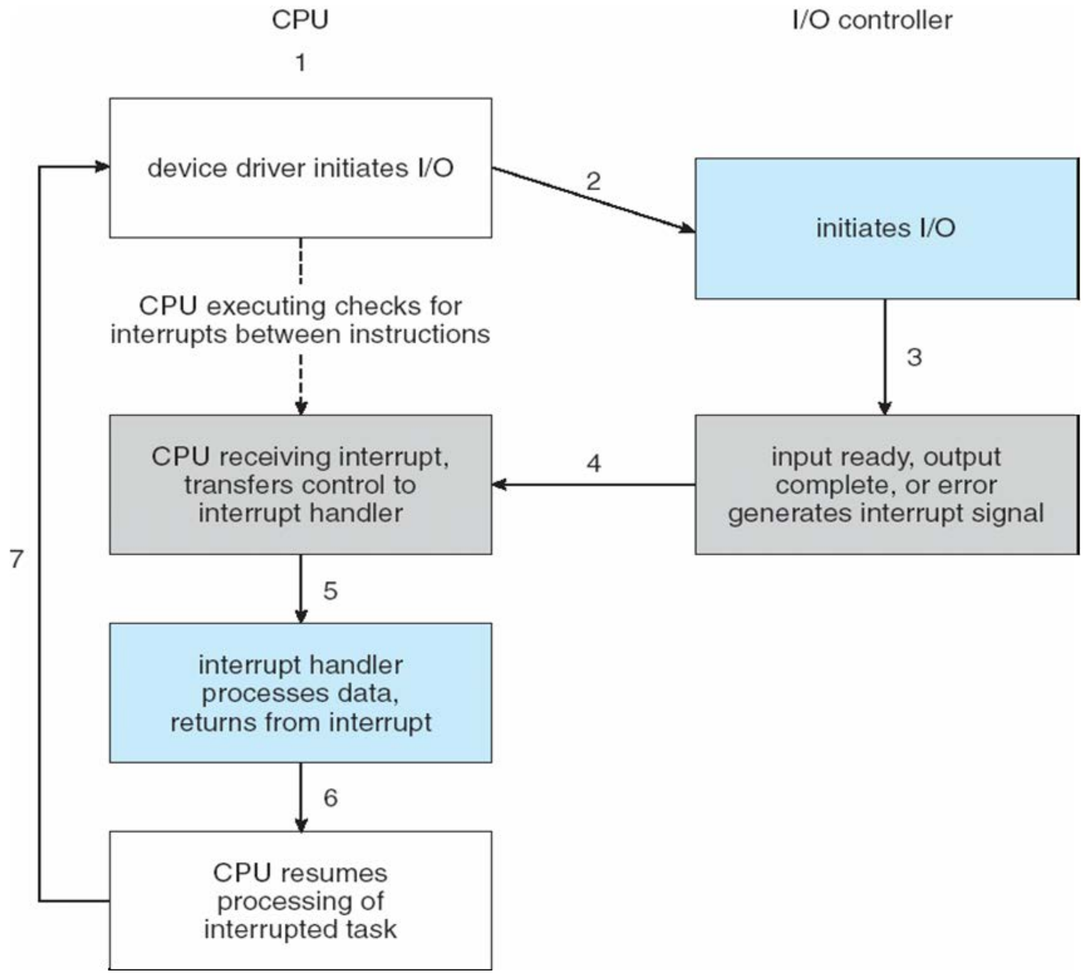
Polling: Write Example

- The host repeatedly reads the *busy bit* in the *status register* until it becomes clear
- The host sets the *write bit* in the *command register* and writes a byte into the data-out register
- The host sets the *command-ready bit* in the *command register.*
- When the controller notices that the *command-ready bit* is set, it sets the *busy bit* in the *status register.*
- The controller reads the *command register* and sees that *write command.* It reads the *data-out register* to get the byte and does the I/O to the device.
- The controller clears the *command-ready bit*, clears the *error bit* in the *status register* to indicate the device I/O succeeded, and clears the *busy bit* to indicate it is finished.

Interrupts

- Purpose: To avoid busy-waiting for the busy bit to become clear.
- CPU has an *Interrupt-request line* that can be triggered by I/O device when the I/O device has input ready, or has completed output (and thus ready for next output)
- *Interrupt handler* receives and handles interrupts

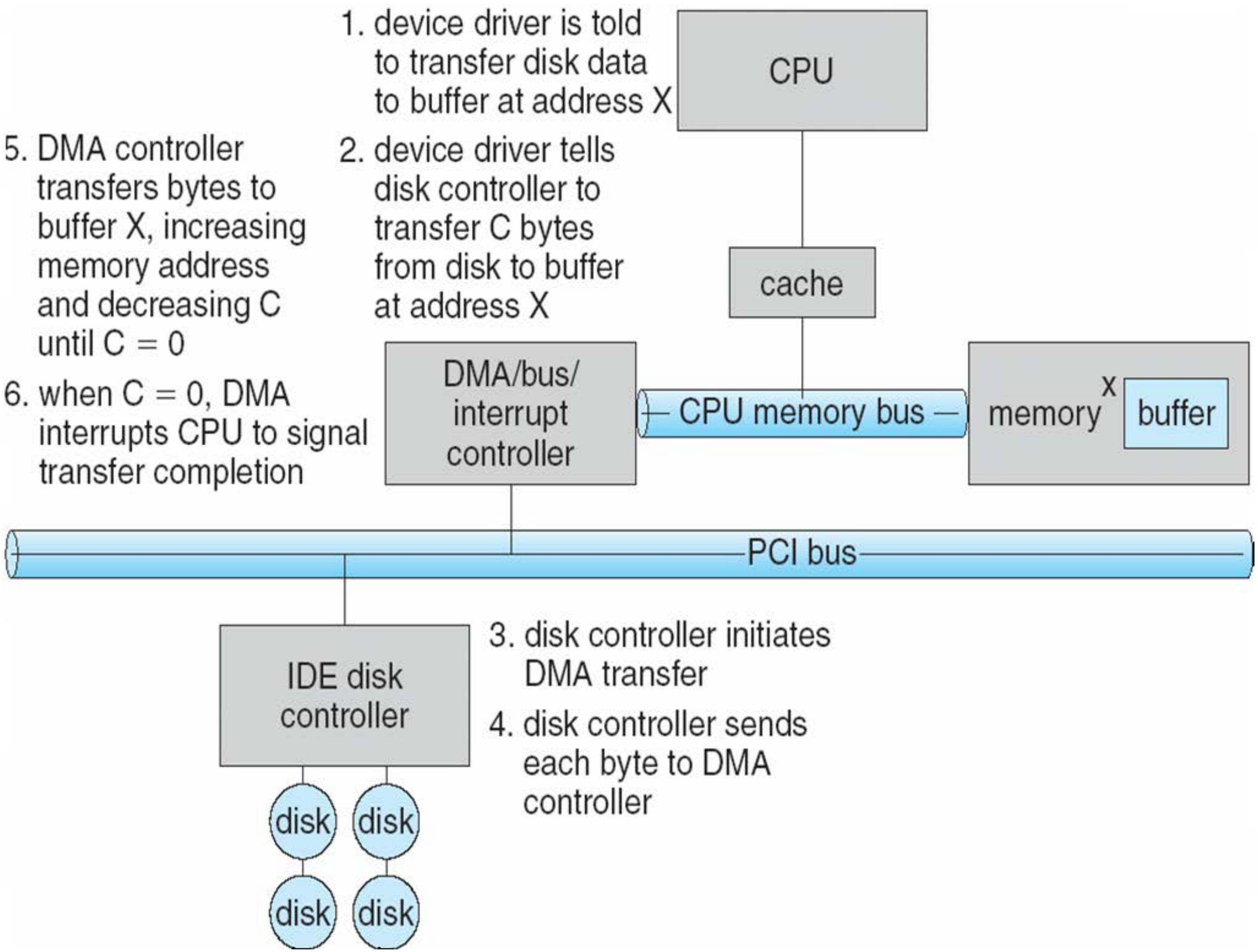
Interrupt-Driven I/O Cycle



Direct Memory Access

- Used to avoid **programmed I/O** (i.e., CPU-involved I/O) for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory

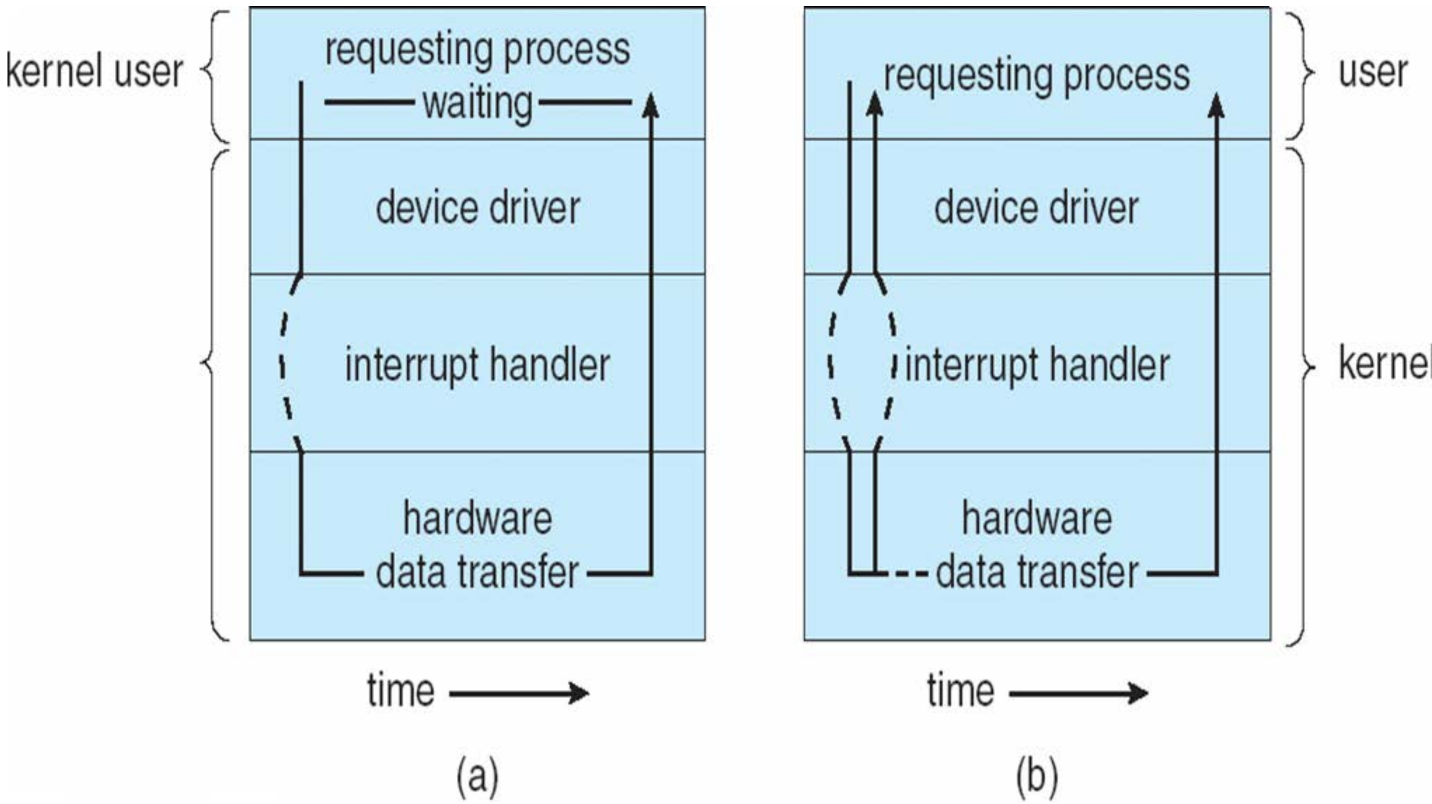
DMA Transfer: Example



Blocking and Nonblocking I/O

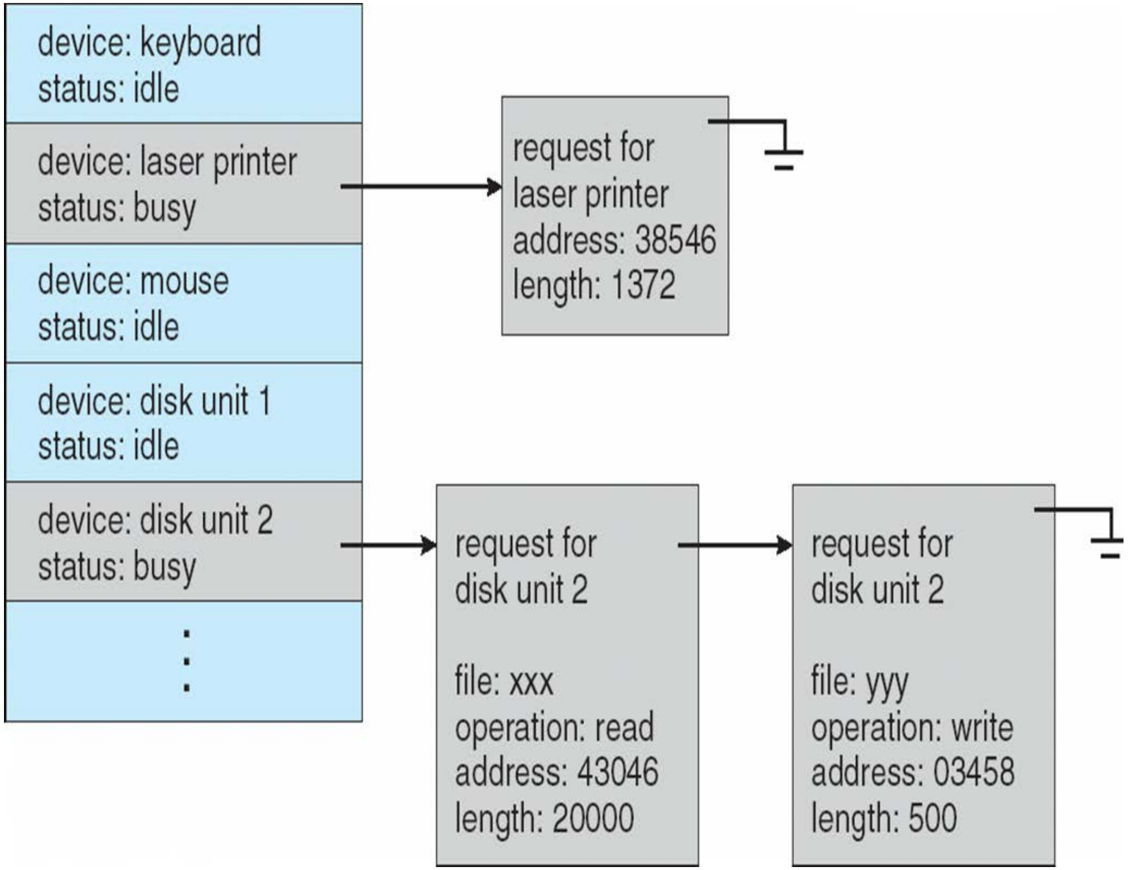
- **Blocking** - process suspended until I/O completed
  - Easy to use and understand
- **Nonblocking** - I/O call returns as much as available
  - Returns quickly with count of bytes read or written
- **Asynchronous** - process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed

Blocking (Synchronous) and Asynchronous



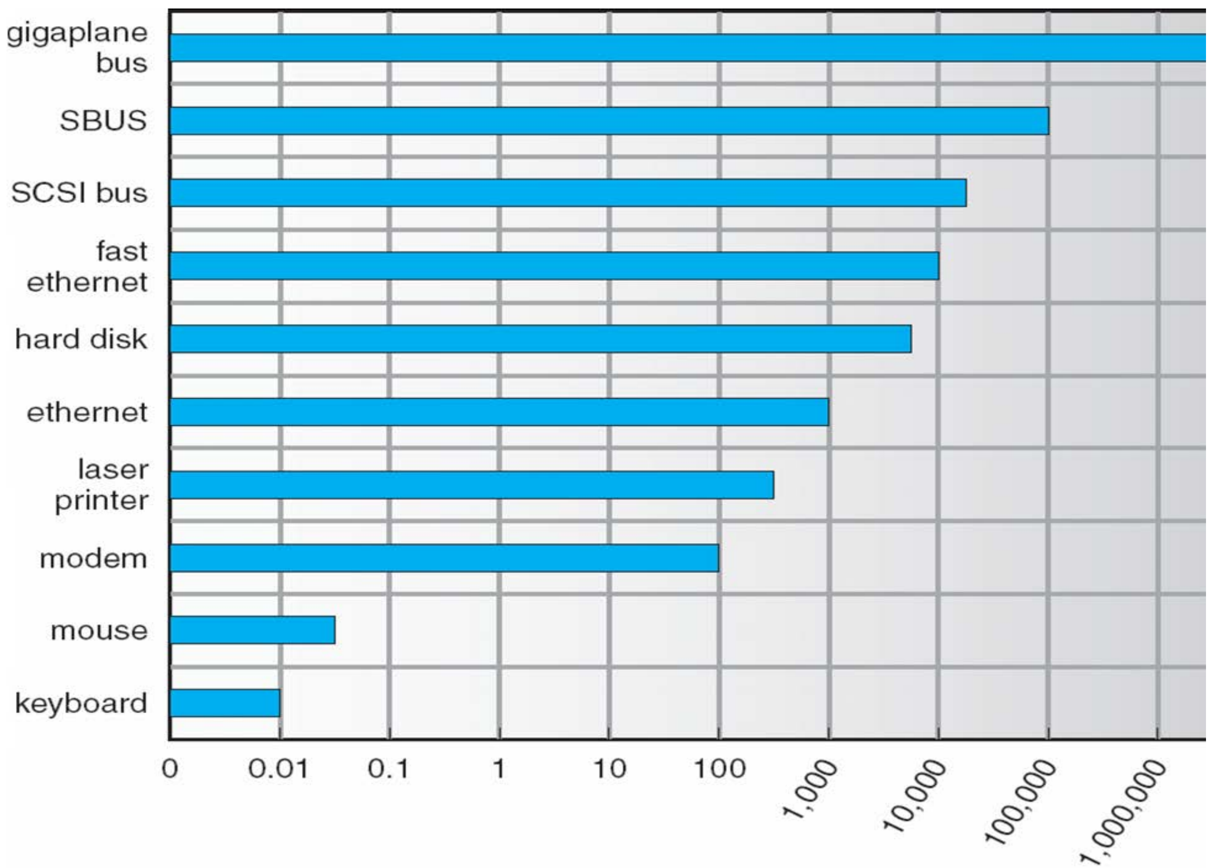
Kernel I/O Subsystem: Scheduling

- Managing request queues for each I/O device
  - When there is a set of pending requests
- I/O request *ordering* in per-device queue
  - Example: Scheduling disk head movement



Kernel I/O Subsystem: Buffering

- Buffering - store data in memory while transferring between devices
  - To cope with device speed mismatch

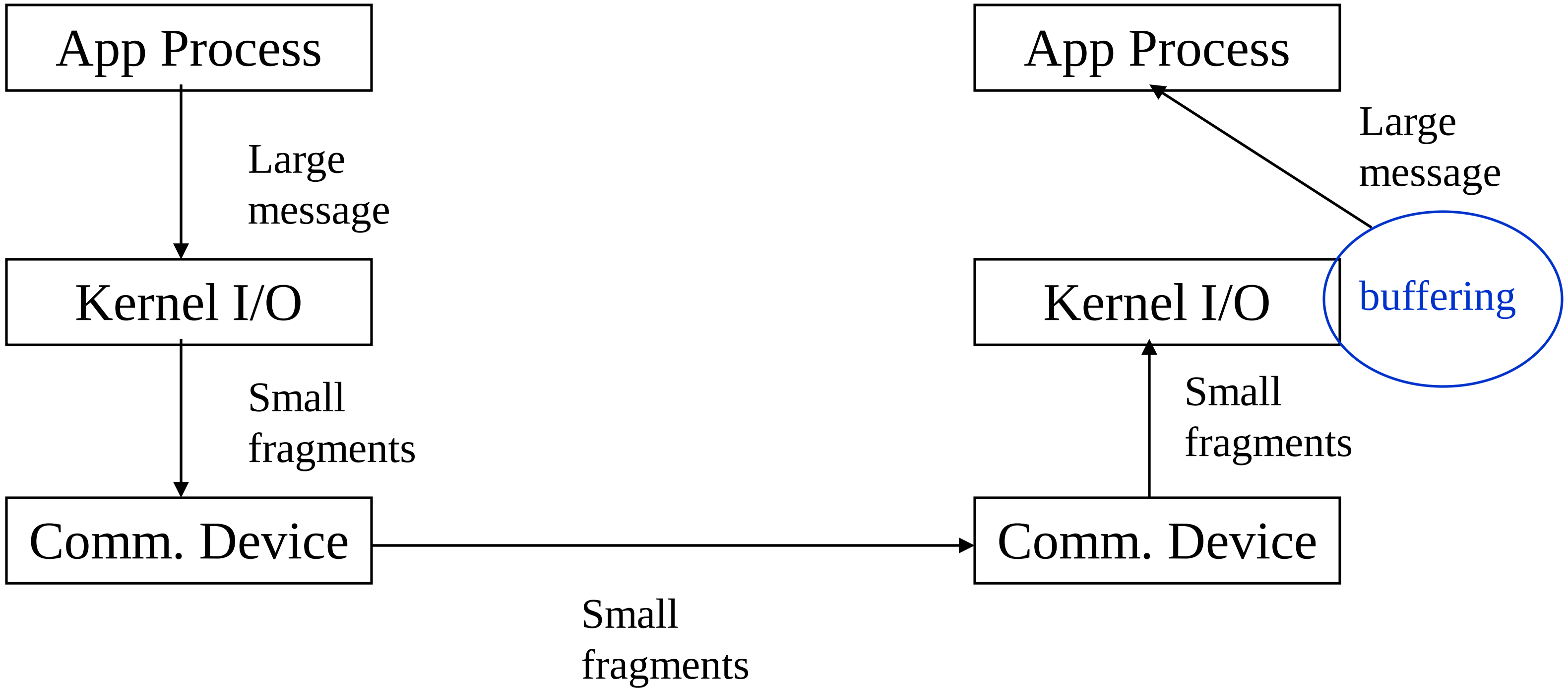


Example: store data received from network interface to hard disk

- received data (*byte by byte*) is buffered
- buffered data are sent to the disk *block by block*

Kernel I/O Subsystem: Buffering

- Buffering - store data in memory while transferring between devices
  - To cope with device transfer size mismatch



Kernel I/O Subsystem: Buffering

- Buffering - store data in memory while transferring between devices
  - To maintain "*copy semantics*"
- Copy semantics
  - Example:
    - system call " write (file-handle, buff)" , where buff points to a buffer space in a user process
    - before the content in the user space buffer is written to the file, if the content is changed by the application process, "copy semantics" is violated
  - Solution: copy the content of buffer to a buffer in the kernel space

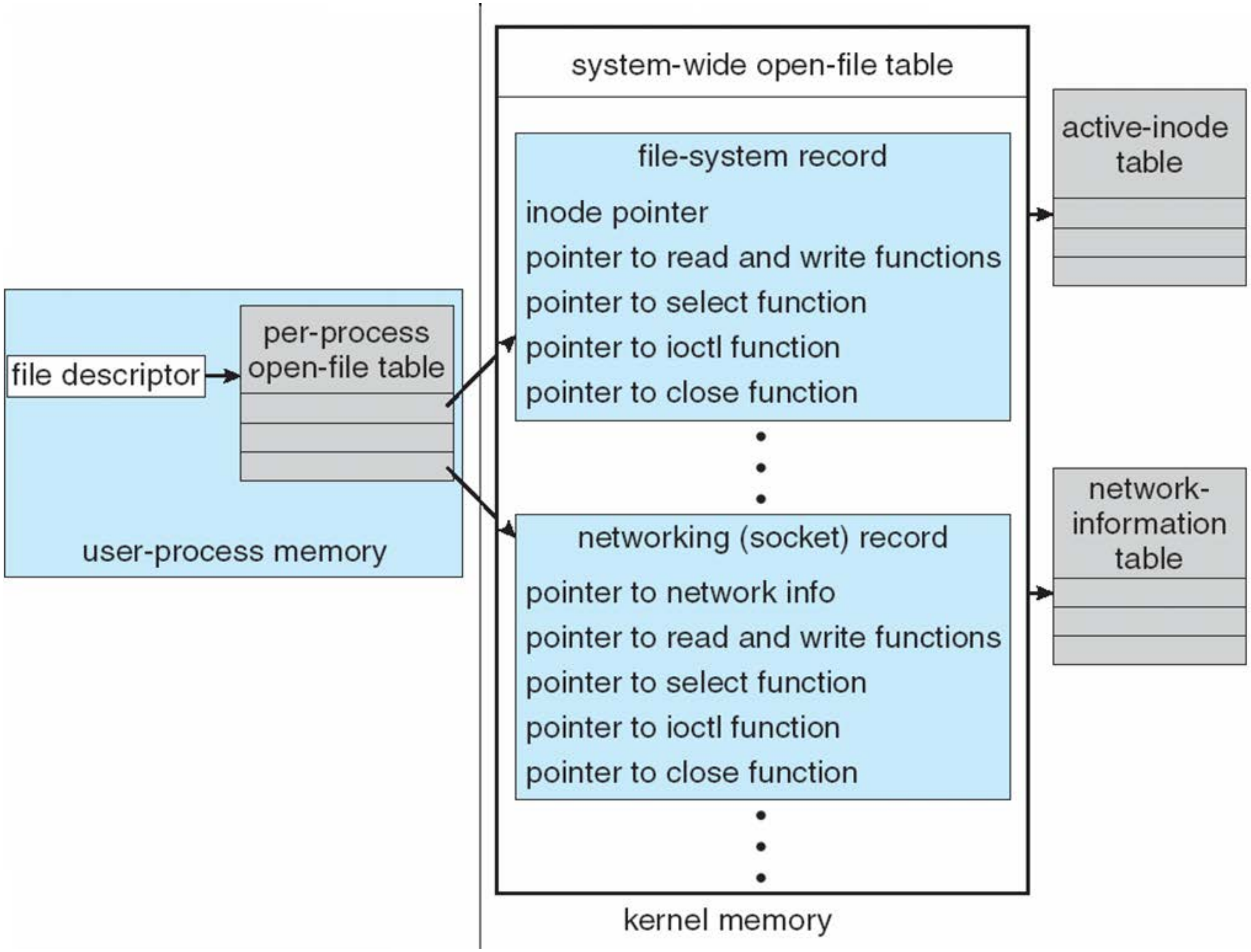
Kernel I/O Subsystem

- **Caching** - fast memory holding copy of data
  - Always just a copy
  - Key to performance
- **Spooling** - hold output for a device
  - If device can serve only one request at a time
  - i.e., printing
- **Device Reservation** - provides exclusive access to a device
  - System calls for allocation and deallocation
  - Watch out for deadlock

Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- Some use object-oriented methods and message passing to implement I/O

UNIX I/O Kernel Structure



I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
    - Memory-mapped and I/O port memory locations must be protected too

Error Handling

- OS can recover from disk read, device unavailable, transient write failures
- Most return an error number or code when I/O request fails
- System error logs hold problem reports