Operating Systems Notes - Chapter 12

July 27, 2025

Contents

1 Overview

1.1 Things to learn

- Explore OS I/O subsystem structure.
- Discuss I/O hardware principles and complexities.
- Explain I/O hardware and software performance aspects.

1.2 Introduction

- Computer's main jobs: I/O and computing.
- Often, I/O is primary, computing incidental (e.g., browsing, editing).
- OS role in I/O: manage and control I/O operations and devices.
- Topics covered:
  - I/O hardware basics: constraints on OS internal facilities.
  - OS I/O services and application I/O interface.
  - Bridging gap between hardware and application interfaces.
  - UNIX System V STREAMS mechanism: dynamic driver code pipelines.
  - I/O performance and OS design principles for improvement.

1.3 Overview

- Device control is major OS design concern.
- Wide variation in I/O device function/speed (mouse, hard disk, flash drive, tape robot) requires varied control methods.
- These methods form kernel's I/O subsystem, separating kernel from device management complexities.
- I/O device technology trends:
  - Increasing standardization of software/hardware interfaces: helps incorporate new device generations.
  - Increasingly broad variety of I/O devices: challenge to incorporate new, unlike devices.
- Challenge met by hardware/software techniques:
  - Basic I/O hardware elements (ports, buses, device controllers) accommodate diverse devices.
  - Kernel structured with **device-driver** modules to encapsulate device details.
  - Device drivers provide uniform device-access interface to I/O subsystem, similar to system calls for applications.

1.4 Section glossary

Term	Definition
device driver	OS component providing uniform access and managing I/O to various devices.

2 I/O hardware

2.1 Device Communication

- Computers operate many device types: storage (disks, tapes), transmission (network, Bluetooth), human-interface (screen, keyboard, mouse, audio).
- Specialized devices exist (e.g., jet steering).
- Device communicates via signals over cable/air.
- Connection point: **port** (e.g., serial port).
- Devices sharing wires: **bus** (e.g., PCI bus).
- Bus: set of wires, rigidly defined protocol for messages (electrical voltages, timings).
- **Daisy chain**: devices connected in string (A to B, B to C), usually operates as a bus.

2.2 Bus Structures

- Buses vary in signaling, speed, throughput, connection.
- Typical PC bus structure:
  - **PCIe bus**: connects processor-memory subsystem to fast devices.
  - **Expansion bus**: connects slow devices (keyboard, serial, USB ports).
  - **Serial-attached SCSI (SAS)** bus: connects disks to SAS controller.
- PCIe: flexible bus, data over "lanes".
  - Lane: two signaling pairs (receive/transmit), full-duplex byte stream.
  - Data packets: eight-bit byte format, simultaneous in both directions.
  - Physical links: 1, 2, 4, 8, 12, 16, or 32 lanes (e.g., x8).
  - Multiple "generations" (e.g., PCIe gen3 x8: 8 GB/s throughput).

2.3 Controllers

- **Controller**: electronics operating a port, bus, or device.
- Serial-port controller: simple, single chip.
- **Fibre channel (FC)** bus controller: complex, often separate circuit board (**host bus adapter (HBA)**).
- HBA: contains processor, microcode, private memory for FC protocol.
- Some devices have built-in controllers (e.g., disk drive board).
- Disk controller: implements disk-side protocol (SAS, SATA), microcode, processor for tasks (bad-sector mapping, prefetching, buffering, caching).

2.4 Memory-mapped I/O

- Processor communicates with controller via registers (data, control).
- Methods:
  - Special I/O instructions: transfer byte/word to I/O port address.
  - **Memory-mapped I/O**: device-control registers mapped into processor address space.
  - CPU uses standard data-transfer instructions to read/write registers at mapped locations.
- Past PCs: mixed I/O instructions and memory-mapped I/O.
- Graphics controller: I/O ports for control, large memory-mapped region for screen contents.
- Writing millions of bytes to graphics memory faster than millions of I/O instructions.
- Trend: systems moved to memory-mapped I/O for efficiency.
- Today: most I/O via device controllers using memory-mapped I/O.

2.5 I/O Device Control Registers

- Typically four registers: status, control, data-in, data-out.
- **Data-in register**: read by host for input.
- **Data-out register**: written by host to send output.
- **Status register**: bits read by host, indicate states (command complete, byte available, error).
- **Control register**: written by host to start command or change device mode (e.g., full/half-duplex, parity, word length, speed).
- Data registers: 1-4 bytes.

- Some controllers have FIFO chips: hold several bytes, expand capacity, buffer data bursts.

2.6 Polling

- Host-controller interaction: handshaking.
- Example (2 bits for coordination):
  1. Host reads ‘busy‘ bit until clear.
  2. Host sets ‘write‘ bit in ‘command‘ register, writes byte to ‘data-out‘ register.
  3. Host sets ‘command-ready‘ bit.
  4. Controller notices ‘command-ready‘ set, sets ‘busy‘ bit.
  5. Controller reads command, reads ‘data-out‘ byte, performs I/O.
  6. Controller clears ‘command-ready‘, clears ‘error‘ bit, clears ‘busy‘ bit.
- Loop repeated for each byte.
- Step 1: host is **busy-waiting** or **polling** (repeatedly reading ‘status‘ register).
- Polling efficient if controller/device fast.
- Inefficient if wait is long and other CPU tasks pending.
- Risk of data loss if host waits too long (buffer overflow).
- Polling efficient for basic operation (3 CPU cycles).
- Inefficient if rarely finds device ready.
- Alternative: hardware controller notifies CPU when ready via **interrupt**.

2.7 Interrupts

- Basic mechanism:
  - CPU senses **interrupt-request line** after each instruction.
  - Controller asserts signal: CPU saves state, jumps to **interrupt-handler routine** at fixed address.
  - Handler determines cause, processes, restores state, executes ‘return from interrupt‘.
  - Device controller *raises* interrupt, CPU *catches* and *dispatches*, handler *clears*.
- Modern systems: hundreds of interrupts/second.
- Sophisticated interrupt handling features needed:
  1. Defer interrupt handling during critical processing.
  2. Efficient dispatch to proper handler (avoid polling all devices).
  3. Multilevel interrupts: distinguish high/low priority, respond urgently.
  4. Instruction to get OS attention directly (**traps**) for page faults, errors (division by zero).
- Features provided by CPU and **interrupt-controller hardware**.
- CPU has two interrupt request lines:
  - **Nonmaskable interrupt**: for unrecoverable errors (e.g., memory errors).
  - **Maskable**: can be turned off by CPU for critical sequences, used by device controllers.
- Interrupt mechanism accepts **address**: selects handler from **interrupt vector**.
- Interrupt vector: table of memory addresses of specialized handlers.
- Purpose: reduce need for single handler to search all sources.
- More devices than vector elements: **interrupt chaining**.
- **Interrupt chaining**: each vector element points to head of handler list; handlers called until one services request.
- Compromise: avoids huge table, improves dispatch efficiency.
- Intel Pentium event-vector table:
  - 0-31 (nonmaskable): error conditions, page faults, debugging.
  - 32-255 (maskable): device-generated interrupts.
- **Interrupt priority levels**: defer low-priority, allow high-priority preemption.
- OS interaction with interrupts:
  - Boot time: probes buses, installs handlers into vector.
  - During I/O: controllers raise interrupts (output complete, input available, failure).
  - Handles **exceptions**: division by zero, protected memory access, privileged instruction from user mode.
  - Events trigger urgent, self-contained routines.
- Interrupt handling split:

  - **First-level interrupt handler (FLIH)**: context switch, state storage, queuing.
  - **Second-level interrupt handler (SLIH)**: performs actual handling.
- Other uses:
  - Virtual memory paging: page fault raises interrupt, suspends process, handler fetches page, schedules another process.
  - System calls: library routines build data structure, execute **software interrupt** or **trap**.
  - Trap: saves user state, switches to kernel mode, dispatches to kernel routine.
  - Trap priority: low compared to device interrupts (less urgent).
  - Kernel flow control: disk read completion (high-priority handler records status, clears interrupt, starts next I/O, raises low-priority interrupt; low-priority handler copies data, calls scheduler).
- Threaded kernel architecture (e.g., Solaris): interrupt handlers as kernel threads, high scheduling priorities, preemption, concurrent execution on multiprocessor.
- Summary: interrupts handle asynchronous events, trap to supervisor mode. Use priority system. Device controllers, hardware faults, system calls raise interrupts. Efficient handling crucial for performance. Interrupt-driven I/O common, polling for high-throughput, sometimes combined.

2.8 Direct memory access

- For large transfers (e.g., disk), **programmed I/O (PIO)** (CPU watching status bits, feeding data byte-by-byte) is wasteful.
- Offload work to **direct-memory-access (DMA)** controller.
- Initiate DMA: host writes DMA command block to memory (source, destination, byte count).
- Command block can be complex: list of non-contiguous sources/destinations (**scatter-gather**).
- CPU writes command block address to DMA controller, continues other work.
- DMA controller operates memory bus directly, performs transfers without main CPU.
- Standard component in modern computers.
- Target address in kernel space is straightforward.
- User space target: risk of modification during transfer.
- Data to user space: second copy operation (**double buffering**) from kernel to user memory, inefficient.
- Trend: OS moved to memory-mapping for direct I/O between devices and user address space.
- Handshaking (DMA controller and device controller): **DMA-request** and **DMA-acknowledge** wires.
- Device places signal on DMA-request when word available.
- DMA controller seizes memory bus, places address, signals DMA-acknowledge.
- Device receives DMA-acknowledge, transfers word, removes DMA-request.
- Transfer finished: DMA controller interrupts CPU.
- DMA controller seizing bus: CPU momentarily prevented from main memory access (**cycle stealing**).
- Cycle stealing slows CPU, but DMA generally improves total system performance.
- Some architectures use physical addresses for DMA, others **direct virtual memory access (DVMA)**.
- DVMA: uses virtual addresses, translates to physical. Can transfer between memory-mapped devices without CPU/main memory.

2.9 I/O hardware summary

- Key concepts:
  - Bus
  - Controller
  - I/O port and its registers
  - Handshaking (host and device controller)
  - Handshaking execution: polling or interrupts
  - Offloading large transfers to DMA controller
- Challenge for OS implementers: wide variety of devices, unique capabilities, control-bit definitions, protocols.
- Questions: how to attach new devices without OS rewrite? How to provide uniform I/O interface to applications?

2.10 Section glossary

Term	Definition
port	Connection point for devices to attach to computers.
PHY	Physical hardware component connecting to a network (OSI layer 1).
bus	Communication system connecting computer components (CPU, I/O devices) fo data/command transfer.
daisy chain	Devices connected in a string (A to B, B to C).
PCIe bus	Common computer I/O bus connecting CPU to I/O devices.
expansion bus	Computer bus for connecting slow devices (e.g., keyboards).
serial-attached SCSI (SAS)	Common type of I/O bus.
SAS	Common type of I/O bus.
controller	Special processor managing I/O devices.
fibre channel (FC)	Storage I/O bus used in data centers to connect computers to storage arrays.
host bus adapter (HBA)	Device controller installed in host bus port for device connection.
memory-mapped I/O	Device I/O method where device-control registers map into processor address space.
data-in register	Device I/O register for host to get input.
data-out register	Device I/O register for host to send output.
status register	Device I/O register indicating status.
control register	Device I/O register for host to place commands.
busy waiting	Thread/process continuously uses CPU while waiting; I/O loop reading status.
polling	I/O loop where I/O thread continuously reads status waiting for I/O completion.
interrupt	Hardware mechanism for device to notify CPU it needs attention.
interrupt-request line	Hardware connection to CPU for signaling interrupts.
interrupt-handler routine	OS routine called when interrupt signal received.
interrupt-controller hardware	Computer hardware components for interrupt management.
nonmaskable interrupt	Interrupt that cannot be delayed or blocked (e.g., unrecoverable memory error).
maskable	Describes an interrupt that can be delayed or blocked.
interrupt vector	OS data structure indexed by interrupt address, pointing to handlers.
interrupt chaining	Mechanism where interrupt vector element points to list of handlers.
interrupt priority level	Prioritization of interrupts for handling order.
exception	Software-generated interrupt by error or user program request for OS service.
first-level interrupt handler (FLIH)	Interrupt handler for reception and queuing of interrupts.
second-level interrupt handler (SLIH)	Interrupt handler that actually handles interrupts.
software interrupt	Software-generated interrupt; also called a trap.
trap	Software interrupt.
programmed I/O (PIO)	Data transfer method where CPU transfers data one byte at a time.
direct memory access (DMA)	Operation allowing device controllers to transfer large data directly to/from main memory
scatter-gather	I/O method specifying multiple sources/destinations in one command.
double buffering	Copying data twice (e.g., device to kernel, then kernel to process), or using two buffers.
cycle stealing	Device (e.g., DMA controller) using bus, temporarily preventing CPU access.
direct virtual memory access (DVMA)	DMA using virtual addresses as transfer sources/destinations.

3 Application I/O interface

3.1 Structuring Techniques and Interfaces

- Goal: treat I/O devices uniformly.
- Example: open file on disk without knowing disk type.
- Add new devices without OS disruption.
- Approach: abstraction, encapsulation, software layering.
- Abstract away device differences by identifying general kinds.
- Each kind accessed via standardized functions (**interface**).
- Differences encapsulated in kernel modules (**device drivers**).
- Device drivers: custom-tailored to specific devices, export standard interfaces.
- Purpose of device-driver layer: hide differences among device controllers from kernel I/O subsystem.
- I/O system calls encapsulate device behavior in generic classes, hiding hardware differences from applications.
- Benefits: simplifies OS development, allows hardware manufacturers to design compatible devices or write drivers for popular OS.
- Device may ship with multiple drivers (Windows, Linux, Mac).

3.2 Device Characteristics

- Devices vary on many dimensions:
  - **Character-stream or block**:
    - Character-stream: transfers bytes one by one.
    - Block: transfers a block of bytes as a unit.
  - **Sequential or random access**:
    - Sequential: fixed order determined by device.
    - Random-access: user seeks to any storage location.
  - **Synchronous or asynchronous**:
    - Synchronous: predictable response times, coordinated.
    - Asynchronous: irregular/unpredictable response times, not coordinated.
  - **Sharable or dedicated**:
    - Sharable: used concurrently by several processes/threads.
    - Dedicated: cannot be used concurrently.
  - **Speed of operation**: few bytes/sec to GB/sec.
  - **Read-write, read only, write once**:
    - Read-write: both input/output.
    - Read only: only one data transfer direction.
    - Write once: written once, then read-only.

3.3 Application Access Conventions

- OS hides many differences, groups devices into conventional types.
- Major access conventions:
  - Block I/O
  - Character-stream I/O
  - Memory-mapped file access
  - Network sockets
- Special system calls for devices like time-of-day clock, timer, graphical display, video, audio.
- Most OS have an **escape** or **back door**: transparently passes arbitrary commands to device driver.
- UNIX: ‘ioctl()‘ system call.
  - Enables application to access any driver functionality without new system call.
  - Three arguments: device identifier (major/minor numbers), command integer, pointer to data structure.
  - Major number: device type, routes I/O requests to driver.
  - Minor number: device instance.

3.4 Block and Character Devices

- **Block-device interface**: for disk drives and block-oriented devices.
- Commands: ‘read()‘, ‘write()‘, ‘seek()‘ (for random-access).
- Applications usually access via file-system interface.
- OS/special applications (e.g., DBMS) may prefer **raw I/O**: direct access as linear array of blocks, no file system.
- Raw I/O avoids extra buffering and redundant locking.
- Compromise: OS allows mode on file that disables buffering/locking (UNIX: **direct I/O**).
- Memory-mapped file access: layered on block-device drivers.
- Access disk storage via byte array in main memory.
- System call maps file to memory, returns virtual address.
- Data transfers only when needed (demand-paged virtual memory access), efficient.
- Convenient for programmers: simple read/write to memory.
- Used for kernel services (e.g., executing program, swap space access).
- **Character-stream interface**: for keyboards, mice, modems, printers, audio boards.
- Basic system calls: ‘get()‘, ‘put()‘ one character.
- Libraries built on top: line-at-a-time access, buffering, editing.
- Convenient for spontaneous input/linear output streams.

3.5 Network Devices

- Network I/O interface differs from disk I/O (‘read()‘-‘write()‘-‘seek()‘).
- Common interface: network **socket** (UNIX, Windows).
- Socket interface system calls:
  - Create socket.
  - Connect local socket to remote address.
  - Listen for remote application connection.
  - Send/receive packets.
  - ‘select()‘: manages set of sockets, returns info on ready sockets (packet waiting, room to send).
- ‘select()‘ eliminates polling/busy waiting for network I/O.
- Facilitates distributed applications using any network hardware/protocol.
- Other IPC/network communication approaches: half-duplex pipes, full-duplex FIFOs, full-duplex STREAMS, message queues, sockets (UNIX).

3.6 Clocks and Timers

- Hardware clocks/timers provide: current time, elapsed time, set timer for operation X at time T.
- Used heavily by OS and time-sensitive applications.
- System calls not standardized.
- **Programmable interval timer**: hardware to measure elapsed time, trigger operations.
- Set to wait, then generate interrupt (once or periodically).
- Used by scheduler (preempt process), disk I/O (flush dirty cache), network (cancel slow operations).
- OS provides user process interface for timers.
- Supports more timer requests than hardware channels by simulating virtual clocks.
- Kernel maintains sorted list of wanted interrupts, sets timer for earliest.
- Hardware clock: high-frequency counter.
- Can be read from device register (high-resolution clock), offers accurate time intervals.
- **High-performance event timer (HPET)**: modern PCs, 10-megahertz range.
- Comparators trigger interrupts when value matches HPET.
- Precision limited by timer resolution + virtual clock overhead.
- System clock drift corrected by protocols (e.g., **network time protocol (NTP)**).

3.7 Nonblocking and Asynchronous I/O

- **Blocking** system call: calling thread suspended, moved to wait queue. Resumes after completion.
- Physical I/O actions: generally asynchronous (varying/unpredictable time).
- OS provides blocking calls for application interface (easier to write).
- **Nonblocking** I/O needed for some user processes (e.g., UI, video app).
- Overlap execution with I/O: multithreaded application (some threads block, others execute).
- Nonblocking I/O system calls: return quickly, indicate bytes transferred, don't halt thread.
- Asynchronous system call: returns immediately, doesn't wait for I/O. Thread continues.
- I/O completion communicated via variable setting, signal/software interrupt, or callback.
- Difference: nonblocking 'read()' returns available data immediately (full, fewer, none); asynchronous 'read()' requests full transfer to complete later.
- Asynchronous activities throughout modern OS (secondary storage, network I/O).
- OS buffers I/O, returns to application, completes request later (optimizes performance).
- Limit on buffer time (e.g., UNIX flushes every 30 seconds).
- Applications can request buffer flush.
- Data consistency: kernel reads from buffers before I/O, ensures data returned to reader.
- Multiple threads to same file may need locking protocols for consistency.
- I/O system calls can indicate synchronous execution for immediate requests.
- 'select()' for network sockets: nonblocking behavior, specifies max waiting time (0 for polling).
- 'select()' overhead: only checks if I/O possible, needs 'read()'/'write()' after.
- Mach: blocking multiple-read call, returns on first completion.

3.8 Vectored I/O

- **Vectored I/O**: one system call performs multiple I/O operations involving multiple locations.
- UNIX 'readv': accepts vector of multiple buffers, reads to/writes from vector.
- Also called **scatter-gather**.
- Benefits:
  - Avoids context-switching and system-call overhead.
  - No need to transfer data to larger buffer first.
  - Some versions provide atomicity (all I/O done without interruption, avoids data corruption).
- Programmers use scatter-gather for increased throughput, decreased overhead.

3.9 Section glossary

Term	Definition
escape	Method of passing arbitrary commands when interface lacks standard method.
back door	Method of passing arbitrary commands when interface lacks standard method.
block-device interface	Interface for I/O to block devices.
raw I/O	Direct access to secondary storage as array of blocks, no file system.
direct I/O	Block I/O bypassing OS block features (buffering, locking).
character-stream interface	Interface for I/O to character devices (e.g., keyboards).
socket	Endpoint for communication; interface for network I/O.
programmable interval timer	Hardware timer provided by many CPUs.
high-performance event timer (HPET)	Hardware timer provided by some CPUs.
network time protocol (NTP)	Network protocol for synchronizing system clocks.
blocking	I/O request that suspends calling thread until I/O completes.
nonblocking	I/O request that returns immediately with available data, allowing thread to continue.
Vectored I/O	One system call performs multiple I/O operations involving multiple locations.
scatter-gather	I/O method specifying multiple sources/destinations in one command structure.

4 Kernel I/O subsystem

4.1 I/O Services

- Kernels provide many I/O-related services.
- Services provided by kernel's I/O subsystem: scheduling, buffering, caching, spooling, device reservation, error handling.
- Builds on hardware and device-driver infrastructure.
- I/O subsystem responsible for protecting itself from errant processes and malicious users.

4.2 I/O Scheduling

- Scheduling I/O requests: determine good execution order.
- Application system call order rarely best.
- Benefits: improve overall system performance, fair device access, reduce average waiting time.
- Example: disk arm near beginning, 3 apps request blocks (end, beginning, middle). OS can serve 2, 3, 1 to reduce arm travel.
- Implementation: maintain wait queue for each device.
- Blocking I/O system call: request placed on device queue.
- I/O scheduler rearranges queue for efficiency, average response time.
- OS may prioritize delay-sensitive requests (e.g., virtual memory subsystem over applications).
- Asynchronous I/O: kernel tracks many requests.
- Uses **device-status table**: entry for each I/O device.
- Table entry: device type, address, state (not functioning, idle, busy).
- If busy, request type and parameters stored in table entry.
- Scheduling I/O operations improves computer efficiency.
- Also uses storage space in main memory/storage hierarchy via buffering, caching, spooling.

4.3 Buffering

- **Buffer**: memory area storing data transferred between devices or device/application.
- Reasons for buffering:
  1. Cope with speed mismatch (producer/consumer).
  2. Example: network (slow) to SSD (fast). Buffer accumulates network bytes, then writes to SSD in single operation.
  3. **Double buffering**: two buffers used to decouple producer/consumer, relax timing. Network fills buffer 1, drive writes buffer 1; network fills buffer 2, drive writes buffer 2.
  4. Provide adaptations for different data-transfer sizes (e.g., network fragmentation/reassembly).
  5. Support **copy semantics** for application I/O.
  6. Copy semantics: data written to disk is version at time of system call, independent of subsequent application buffer changes.
  7. OS guarantees copy semantics: 'write()' copies application data to kernel buffer before returning. Disk write from kernel buffer.
  8. Copying data between kernel/application space common despite overhead, due to clean semantics.
  9. More efficient: virtual memory mapping and copy-on-write page protection.

4.4 Caching

- **Cache**: region of fast memory holding copies of data.
- Access to cached copy more efficient than original.
- Example: process instructions on disk, cached in physical memory, copied to CPU caches.
- Difference from buffer: buffer may hold only copy, cache holds copy of item residing elsewhere.
- Caching and buffering distinct, but memory region can serve both.
- OS uses main memory buffers for disk data (copy semantics, efficient scheduling).
- These buffers also act as cache: improve I/O efficiency for shared files or rapid write/reread.
- Kernel checks buffer cache for file I/O request: if available, avoids/defers physical disk I/O.
- Disk writes accumulated in buffer cache for seconds: allows efficient write schedules.

4.5 Spooling and Device Reservation

- **Spool**: buffer holding output for device (e.g., printer) that cannot accept interleaved data streams.
- Printer serves one job at a time, but multiple apps may print concurrently.
- OS intercepts printer output: each application's output spooled to separate secondary storage file.
- Application finishes: spooling system queues spool file for printer output.
- Spooling system copies queued files to printer one at a time.
- Managed by system daemon process or in-kernel thread.
- OS provides control interface: display queue, remove jobs, suspend printing.
- Some devices (tape drives, printers) cannot multiplex I/O requests.
- Spooling coordinates concurrent output.
- Explicit coordination facilities: OS supports exclusive device access.
- VMS: process allocates idle device, deallocates when done.
- Other OS: limit one open file handle to such device.
- Many OS provide functions for processes to coordinate exclusive access (e.g., Windows 'wait' for device object, 'OpenFile()' parameter for access types).
- Applications responsible for avoiding deadlock.

4.6 Error Handling

- Protected memory OS guards against hardware/application errors, prevents system failure from minor malfunctions.
- Devices/I/O transfers can fail: transient (network overloaded) or permanent (defective controller).
- OS compensates for transient failures: disk 'read()' retry, network 'send()' resend.
- Permanent failure of important component: OS unlikely to recover.
- I/O system call returns success/failure bit.
- UNIX: 'errno' integer variable returns error code (hundreds of values: argument out of range, bad pointer, file not open).
- Some hardware provides detailed error info, but many OS don't convey to application.
- SCSI protocol error reporting:
  - **Sense key**: general nature of failure (hardware error, illegal request).
  - Additional sense code: category of failure (bad command parameter, self-test failure).
  - Additional sense-code qualifier: more detail (which parameter, which subsystem failed).
- Many SCSI devices maintain internal error-log pages (seldom requested).

4.7 I/O Protection

- Errors related to protection.
- User process may disrupt system by illegal I/O instructions.
- Prevention: all I/O instructions are privileged.
- Users cannot issue I/O instructions directly; must use OS.
- User program executes system call for OS to perform I/O.
- OS (monitor mode) checks request validity, performs I/O, returns to user.
- Memory-mapped and I/O port memory locations protected from user access by memory-protection system.
- Kernel cannot deny all user access (e.g., graphics games need direct access to memory-mapped graphics memory).
- Kernel might provide locking mechanism to allocate graphics memory section to one process at a time.

4.8 Kernel Data Structures

- Kernel keeps state info on I/O components via in-kernel data structures.
- Example: open-file table structure.
- Tracks network connections, character-device communications, other I/O activities.
- UNIX file-system access to various entities (user files, raw devices, process address spaces).
- 'read()' semantics differ for each entity.
- UNIX encapsulates differences using object-oriented technique.
- Open-file record: contains dispatch table with pointers to appropriate routines based on file type.
- Some OS use message-passing for I/O (e.g., Windows).
- I/O request converted to message, sent through kernel to I/O manager, then device driver.

- Message contents may change.
- Message-passing overhead vs. procedural techniques (shared data structures).
- Benefits: simplifies I/O system structure/design, adds flexibility.

4.9 Power Management

- Data centers: power costs, greenhouse gas emissions, heat generation (cooling uses twice as much electricity as powering equipment).
- OS role in power use:
  - Cloud computing: adjust processing loads, evacuate user processes, idle/power off systems.
  - Analyze load, power off components (CPUs, external I/O devices) if low and hardware-enabled.
  - CPU cores suspended/resumed based on load (state saved/restored).
  - Servers: disabling unneeded cores decreases electricity/cooling needs.
- Mobile computing: power management high priority (maximize battery life).
- Android power management features:
  1. Power collapse: deep sleep state, marginally more power than off, responds to external stimuli, quick wake-up.
  2. Achieved by powering off individual components (screen, speakers, I/O subsystem), CPU in lowest sleep state.
  3. Idle Android phone uses little power, wakes quickly for calls.
  4. Component-level power management: infrastructure understands component relationships and usage.
  5. Android builds device tree (physical-device topology).
  6. Each component associated with device driver, tracks usage (e.g., I/O pending to flash, open audio reference).
  7. If component unused, turned off. If all components on bus unused, bus off. If all components in device tree unused, system enters power collapse.
  8. Wakelocks: applications temporarily prevent power collapse.
  9. Applications acquire/release wakelocks. Kernel prevents power collapse while wakelock held.
  10. Example: Android Market holds wakelock during app update.
- Power management based on device management.
- Boot time: firmware analyzes hardware, creates device tree in RAM.
- Kernel uses device tree to load drivers, manage devices.
- Additional activities: hot-plug (add/subtract devices), device state management, power management.
- Modern computers use **advanced configuration and power interface (ACPI)** firmware.
- ACPI: industry standard, provides callable routines for kernel (device state discovery/management, error management, power management).
- Example: kernel calls device driver, which calls ACPI routines, which talk to device.

4.10 Kernel I/O Subsystem Summary

- I/O subsystem coordinates extensive services for applications and kernel.
- Supervises:
  - Management of name space for files/devices.
  - Access control to files/devices.
  - Operation control (e.g., modem cannot ‘seek()‘).
  - File-system space allocation.
  - Device allocation.
  - Buffering, caching, spooling.
  - I/O scheduling.
  - Device-status monitoring, error handling, failure recovery.
  - Device-driver configuration and initialization.
  - Power management of I/O devices.
- Upper levels of I/O subsystem access devices via uniform interface from device drivers.

4.11 Section glossary

Term	Definition
device-status table	Kernel data structure tracking status and queues of operations for devices.
buffer	Memory area storing data being transferred.
double buffering	Copying data twice or using two buffers to decouple producers/consumers.
copy semantics	Meaning assigned to data copying (e.g., if data can be modified after write request).
cache	Temporary copy of data in fast memory to improve performance.
spool	Buffer holding output for device that cannot accept interleaved data streams.
sense key	SCSI protocol info in status register indicating error.
advanced configuration and power interface (ACPI)	Firmware managing hardware aspects (power, device info).

5 Transforming I/O requests to hardware operations

5.1 Connecting Application Requests to Hardware

- How OS connects application request to network wires or disk sector.
- Example: reading file from disk.
- Application refers to data by file name.
- File system maps file name through directories to space allocation.
- MS-DOS for FAT: name maps to number, indicates entry in file-access table, tells which disk blocks allocated.
- UNIX: name maps to inode number, inode contains space-allocation info.
- How is connection made from file name to disk controller (hardware port address or memory-mapped registers)?

5.2 Device Naming and Lookup

- MS-DOS for FAT: first part of file name (before colon) identifies hardware device (e.g., ‘C:‘ for primary hard disk).
- ‘C:‘ mapped to specific port address via device table.
- Device name space separate from file-system name space (due to colon separator).
- Easy to associate extra functionality (e.g., spooling for printer files).
- UNIX: device name space incorporated in regular file-system name space.
- No clear separation of device portion in path name.
- UNIX uses **mount table**: associates path name prefixes with specific device names.
- Resolve path name: lookup in mount table for longest matching prefix.
- Mount table entry gives device name (also in file-system name space).
- Lookup device name: finds ‘¡major, minor¿‘ device number, not inode.
- Major device number: identifies device driver to handle I/O.
- Minor device number: passed to device driver to index into device table.
- Device-table entry: gives port address or memory-mapped address of device controller.
- Modern OS flexibility: multiple stages of lookup tables.
- Mechanisms passing requests between applications/drivers are general.
- New devices/drivers can be introduced without kernel recompilation.
- Some OS load device drivers on demand.
- Boot time: system probes buses, loads necessary drivers (immediately or on first request).
- Devices added after boot: detected by error, kernel inspects, loads driver dynamically.
- Dynamic loading/unloading: more complex kernel algorithms, device-structure locking, error handling.

5.3 Life Cycle of a Blocking Read Request

1. Process issues blocking ‘read()‘ system call to file descriptor of opened file.
2. Kernel system-call code checks parameters. If data in buffer cache, data returned, I/O completed.
3. Else, physical I/O performed. Process removed from run queue, placed on wait queue for device. I/O request scheduled.
4. I/O subsystem sends request to device driver (subroutine call or in-kernel message).
5. Device driver allocates kernel buffer space, schedules I/O. Sends commands to device controller by writing to device-control registers.
6. Device controller operates device hardware for data transfer.
7. Driver may poll for status/data, or set up DMA transfer to kernel memory. DMA controller generates interrupt on transfer completion.
8. Correct interrupt handler receives interrupt via interrupt-vector table, stores data, signals device driver, returns.
9. Device driver receives signal, determines completed I/O request, status, signals kernel I/O subsystem.
10. Kernel transfers data/return codes to requesting process's address space. Moves process from wait queue to ready queue.
11. Moving process to ready queue unblocks it. Scheduler assigns CPU, process resumes at system call completion.

5.4 Section glossary

Term	Definition
mount table	In-memory data structure with info about each mounted volume, tracks file systems and access.

6 Streams

6.1 STREAMS Mechanism

- UNIX System V (and subsequent releases) has **STREAMS** mechanism.
- Enables dynamic assembly of driver code pipelines.
- Stream: full-duplex connection between device driver and user-level process.
- Components:
  - **Stream head**: interfaces with user process.
  - **Driver end**: controls the device.
  - Zero or more **stream modules**: between stream head and driver end.
- Each component: pair of queues (read queue, write queue).
- Data transfer: message passing between queues.

6.2 Modules and Flow Control

- Modules provide STREAMS processing functionality.
- Modules are *pushed* onto a stream using 'ioctl()' system call.
- Example: open USB device (keyboard) via stream, push module for input editing.
- Message exchange between queues in adjacent modules.
- To prevent queue overflow: queue may support **flow control**.
- Without flow control: queue accepts all messages, immediately sends to adjacent queue without buffering.
- With flow control: buffers messages, does not accept messages without sufficient buffer space.
- Involves control message exchanges between adjacent module queues.

6.3 User Process Interaction

- User process writes data to device: 'write()' or 'putmsg()' system call.
- 'write()': writes raw data to stream.
- 'putmsg()': allows user to specify a message.
- Stream head copies data into message, delivers to next module's queue.
- Copying continues to driver end, then device.
- User process reads data from stream head: 'read()' or 'getmsg()' system call.
- 'read()': stream head gets message, returns ordinary data (unstructured byte stream).
- 'getmsg()': message returned to process.
- STREAMS I/O is asynchronous (or nonblocking) except when communicating with stream head.
- Writing to stream: user process blocks (if next queue uses flow control) until room to copy message.
- Reading from stream: user process blocks until data available.

6.4 Driver End and Benefits

- Driver end (like stream head/modules) has read/write queue.
- Driver end must respond to interrupts (e.g., frame ready from network).
- Unlike stream head (may block), driver end must handle all incoming data.
- Drivers must support flow control.
- If device buffer full: device typically drops incoming messages (e.g., network card).
- Benefit of STREAMS: framework for modular, incremental device drivers and network protocols.
- Modules reusable by different streams/devices (e.g., networking module for Ethernet and 802.11 wireless).
- Supports message boundaries and control info between modules (not just unstructured byte stream).
- Most UNIX variants support STREAMS; preferred for protocols/device drivers.
- Example: System V UNIX and Solaris implement socket mechanism using STREAMS.

6.5 Section glossary

Term	Definition
STREAMS	UNIX I/O feature allowing dynamic assembly of driver code pipelines.
stream head	Interface between STREAMS and user processes.
driver end	Interface between STREAMS and the controlled device.
stream modules	Modules of functionality loadable into a STREAM.
flow control	Method to pause a sender of I/O; limits data flow rate.

7 Performance

7.1 I/O and System Performance

- I/O: major factor in system performance.
- Heavy demands on CPU: execute device-driver code, schedule processes (block/unblock).
- Context switches: stress CPU and hardware caches.
- Exposes inefficiencies in kernel's interrupt-handling mechanisms.
- Loads memory bus: data copies between controllers/physical memory, and kernel buffers/application space.
- Graceful coping with demands: major concern for computer architects.
- Interrupt handling: relatively expensive (state change, execute handler, restore state).
- Programmed I/O (PIO) can be more efficient than interrupt-driven I/O if busy waiting minimized.
- I/O completion unblocks process: leads to full context switch overhead.
- Network traffic: high context-switch rate.
- Example: remote login character.
  - Local machine: character typed → keyboard interrupt → interrupt handler → device driver → kernel → user process.
  - User process issues network I/O system call → local kernel → network layers (packet construction) → network device driver.
  - Network device driver transfers packet to controller → sends character, generates interrupt.
  - Interrupt back up through kernel → network I/O system call completes.
  - Remote system: network hardware receives packet → interrupt generated.
  - Character unpacked from protocols → appropriate network daemon.
  - Network daemon identifies session → passes packet to subdaemon.
  - Throughout: context switches and state switches.
  - Receiver echoes character: doubles work.

7.2 I/O Efficiency Improvement Principles

- Some systems use separate **front-end processors** for terminal I/O to reduce main CPU interrupt burden.
- **Terminal concentrator**: multiplexes traffic from hundreds of remote terminals into one port.
- **I/O channel**: dedicated, special-purpose CPU in mainframes/high-end systems.
- Channel job: offload I/O work from main CPU, keep data flowing smoothly.
- Channels process more general/sophisticated programs, tuned for workloads.
- Principles to improve I/O efficiency:
  - Reduce number of context switches.
  - Reduce data copies in memory (between device/application).
  - Reduce interrupt frequency: use large transfers, smart controllers, polling (if busy waiting minimal).
  - Increase concurrency: use DMA-knowledgeable controllers/channels to offload data copying from CPU.
  - Move processing primitives into hardware: concurrent operation with CPU/bus.
  - Balance CPU, memory subsystem, bus, I/O performance: overload in one area causes idleness in others.
- I/O device complexity varies (mouse simple, Windows disk driver complex).
- Windows disk driver: manages individual disks, implements RAID arrays, converts requests to disk I/O, error handling, data recovery, optimizes performance.

7.3 I/O Functionality Implementation Location

- Where to implement I/O functionality: device hardware, device driver, or application software?
- Progression observed:
  1. Initially: experimental I/O algorithms at application level.
     - Flexible, bugs unlikely to crash system.
     - No reboot/reload drivers after code changes.
     - Inefficient: context switch overhead, no internal kernel data/functionality (messaging, threading, locking).
     - Example: FUSE system interface allows user-mode file systems.
  2. When proven: reimplement in kernel.
     - Improves performance.
     - More challenging development (large, complex kernel).

- Must be thoroughly debugged (avoid data corruption, system crashes).

3. Highest performance: specialized implementation in hardware (device or controller).
   - Disadvantages: difficulty/expense of improvements/bug fixes.
   - Increased development time (months vs. days).
   - Decreased flexibility (e.g., hardware RAID controller may not allow kernel to influence I/O order/location).

- I/O devices increasing in speed (NVM devices nearing DRAM speed).
- Increases pressure on I/O subsystems and OS algorithms to leverage read/write speeds.
- I/O performance of storage and network latency: CPU, caches, DRAM, NVM, PCIe, SSD, SAA, HDD.

7.4 Section glossary

Term	Definition
front-end processors	Small computers performing tasks in overall system; manage I/O, offload CPU.
terminal concentrator	Type of front-end processor for terminals.
I/O channel	Dedicated, special-purpose CPU in large systems for I/O or offloading main CPU.

8 Summary

8.1 Key Points

- Basic I/O hardware elements: buses, device controllers, devices.
- Data movement: CPU (programmed I/O) or DMA controller.
- Device driver: kernel module controlling a device.
- System-call interface handles basic hardware categories: block devices, character-stream devices, memory-mapped files, network sockets, programmed interval timers.
- System calls usually block processes, but nonblocking/asynchronous calls used by kernel/applications that must not sleep.
- Kernel's I/O subsystem provides services: I/O scheduling, buffering, caching, spooling, device reservation, error handling.
- Name translation: connects hardware devices to symbolic file names.
- Involves multiple mapping levels: character-string names $\rightarrow$ device drivers/addresses $\rightarrow$ physical addresses (I/O ports/bus controllers).
- Mapping can be within file-system name space (UNIX) or separate device name space (MS-DOS).
- STREAMS: UNIX mechanism for dynamic assembly of driver code pipelines.
- Drivers can be stacked, data passes sequentially and bidirectionally.
- I/O system calls are costly:
  - Context switching (kernel protection boundary).
  - Signal/interrupt handling.
  - CPU/memory load for data copying (kernel buffers $\leftrightarrow$ application space).