

OPERATING SYSTEM

Chapter 6

I/O MANAGEMENT

Detailed Study Notes for Engineering Students

Topics Covered

I/O Hardware, Polling, Interrupts, DMA

Application I/O Interface (Block and Character Devices, Network Devices, Clocks and Timers, Blocking and Nonblocking I/O)

Kernel I/O Subsystem (Scheduling, Buffering, Caching, Spooling and Device Reservation, Error Handling)

Performance

15 MCQs with Answer Key | 10 Broad Questions

Contents

- 6.1** Introduction to I/O Management
- 6.2** I/O Hardware
- 6.3** Polling (Programmed I/O)
- 6.4** Interrupts
- 6.5** Direct Memory Access (DMA)
- 6.6** Application I/O Interface
- 6.7** Kernel I/O Subsystem
- 6.8** Life Cycle of an I/O Request
- 6.9** Performance
- 6.10** Chapter Summary and Quick Revision
- 6.11** Multiple Choice Questions (15)
- 6.12** Answer Key
- 6.13** Broad Questions (10)

6.1 Introduction to I/O Management

Computers exist to process data, and data must enter and leave the machine through **input/output (I/O) devices**. The two main jobs of an operating system are managing the CPU and memory, and managing the enormous variety of I/O devices attached to the computer. Devices differ in speed, data unit, direction of transfer, error behaviour and purpose, so the OS needs a clean, uniform way to control them. The part of the OS that does this work is called **I/O management**.

The design goal of I/O management is to hide the complexity of devices behind a small set of standard interfaces. Device-specific code is pushed into modules called **device drivers**, while the rest of the kernel and the applications see a uniform view of I/O. This is the same idea as abstraction used throughout operating system design.

Main Objectives of I/O Management

- **Device independence:** programs should work with any similar device without change (for example, a file can be written to a USB drive or a hard disk with the same call).
- **Uniform naming:** devices are named by simple strings or paths (for example, /dev/sda in UNIX).
- **Efficiency:** since I/O devices are very slow compared with the CPU, the OS must overlap I/O with computation.
- **Error handling:** errors should be handled as close to the hardware as possible and hidden from higher layers whenever they can be corrected.
- **Protection and sharing:** the OS must arbitrate between competing processes and prevent illegal access to devices.
- **Synchronous and asynchronous support:** both blocking and interrupt-driven styles should be possible.

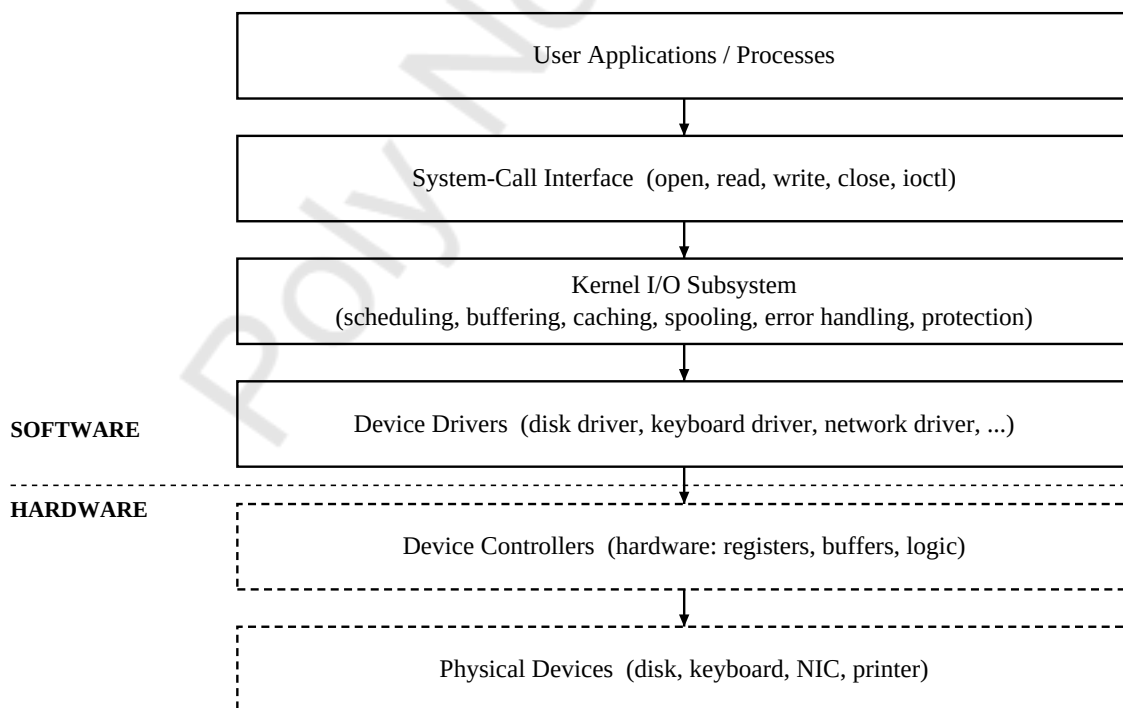


Fig. 6.1: Layered structure of the I/O system

6.2 I/O Hardware

Although I/O devices look very different, they connect to the computer in only a few ways. A device communicates with the system through a **port** (a connection point, such as a serial port), a **bus** (a shared set of wires with a defined protocol, such as PCIe or USB), or a **daisy chain**. Between the device and the bus sits an electronic unit called the **device controller** (or host adapter).

6.2.1 Categories of I/O Devices

Category	Purpose	Examples
Storage devices	Long-term data storage	Hard disk, SSD, optical disc, tape
Transmission devices	Communication with other systems	Network card, Wi-Fi adapter, modem
Human-interface devices	Interaction with the user	Keyboard, mouse, display, printer, speaker
Specialised devices	Domain-specific control	Sensors, actuators, flight-control units

6.2.2 Typical Device Data Rates

Device	Approximate data rate
Keyboard	about 10 bytes per second
Mouse	about 100 bytes per second
Gigabit Ethernet	about 125 MB per second
Hard disk (SATA)	about 100 to 200 MB per second
NVMe SSD	several GB per second

The very large difference in data rates (ten bytes per second to gigabytes per second) is the main reason why different I/O techniques are needed.

6.2.3 Device Controller and Its Registers

The controller is a small processor with its own circuitry. The CPU talks to it by reading and writing the controller's **registers**. Most controllers have four registers:

Register	Function
Data-in register	Read by the host to get input from the device
Data-out register	Written by the host to send output to the device
Status register	Holds bits such as <i>busy</i> , <i>command-ready</i> , <i>error</i> , <i>data-ready</i> ; read by the host
Control register	Written by the host to start a command or change the device mode

6.2.4 Ways of Addressing Device Registers

- **Port-mapped (isolated) I/O:** special I/O instructions (for example, IN and OUT on x86) with a separate I/O address space.
- **Memory-mapped I/O:** registers or device memory are mapped into the normal address space and accessed with ordinary load/store instructions. It is common for graphics controllers, which have large memory to be filled quickly.

- **Hybrid:** many systems use both (for example, x86 graphics cards use memory-mapped regions and I/O ports for control).

Feature	Port-mapped I/O	Memory-mapped I/O
Address space	Separate from memory	Shared with memory
Instructions	Special (IN/OUT)	Normal (LOAD/STORE)
Address space used	Does not consume memory addresses	Consumes part of memory addresses
Programming	Needs assembly or special functions	Can use C pointers directly
Protection	Easy via privileged instructions	Done using memory protection (paging)

6.3 Polling (Programmed I/O)

Polling is a technique in which the CPU repeatedly checks the status of a device in a loop until the device is ready. It is also called **busy-waiting** or **programmed I/O**. The host and the controller coordinate with a **handshaking** protocol using two bits: the *busy* bit (set by the controller) and the *command-ready* bit (set by the host).

Handshaking Steps for Writing One Byte

1. The host repeatedly reads the busy bit until it becomes clear (this is the polling loop).
2. The host sets the *write* bit in the command register and writes a byte into the data-out register.
3. The host sets the *command-ready* bit.
4. When the controller notices command-ready, it sets the *busy* bit.
5. The controller reads the command register, sees the write command, reads the data-out register, and performs the I/O to the device.
6. The controller clears command-ready, clears the error bit (success), and clears the busy bit to show it has finished.

Advantages and Disadvantages

Advantages	Disadvantages
Simple to design and program	Wastes CPU cycles while waiting (busy-wait)
No interrupt overhead or context switch	Inefficient if device is slow or seldom ready
Efficient when device is fast and the wait is very short	CPU cannot do other useful work meanwhile
Predictable timing; good for dedicated embedded systems	Poor scalability with many devices

Remember: Polling takes three CPU instruction cycles (read status, logical-AND to extract the status bit, branch if not zero). This is cheap if the wait is short, but becomes expensive when the CPU must poll for a long time.

6.4 Interrupts

An **interrupt** is a signal to the CPU, sent by hardware or software, indicating that an event needs immediate attention. Instead of polling, the CPU can do useful work and the device *tells* the CPU when it needs service. The CPU hardware has an **interrupt-request line** that it senses after executing every instruction.

6.4.1 Basic Interrupt Mechanism

1. The device controller **raises** an interrupt by asserting a signal on the interrupt-request line.
2. The CPU **catches** the interrupt after the current instruction and saves the state (program counter and flags).
3. The CPU **dispatches** to the **interrupt handler** (Interrupt Service Routine, ISR), whose address is found using the interrupt vector.
4. The handler determines the cause, services the device and **clears** the interrupt.
5. The CPU **restores** the state and returns to the interrupted program.

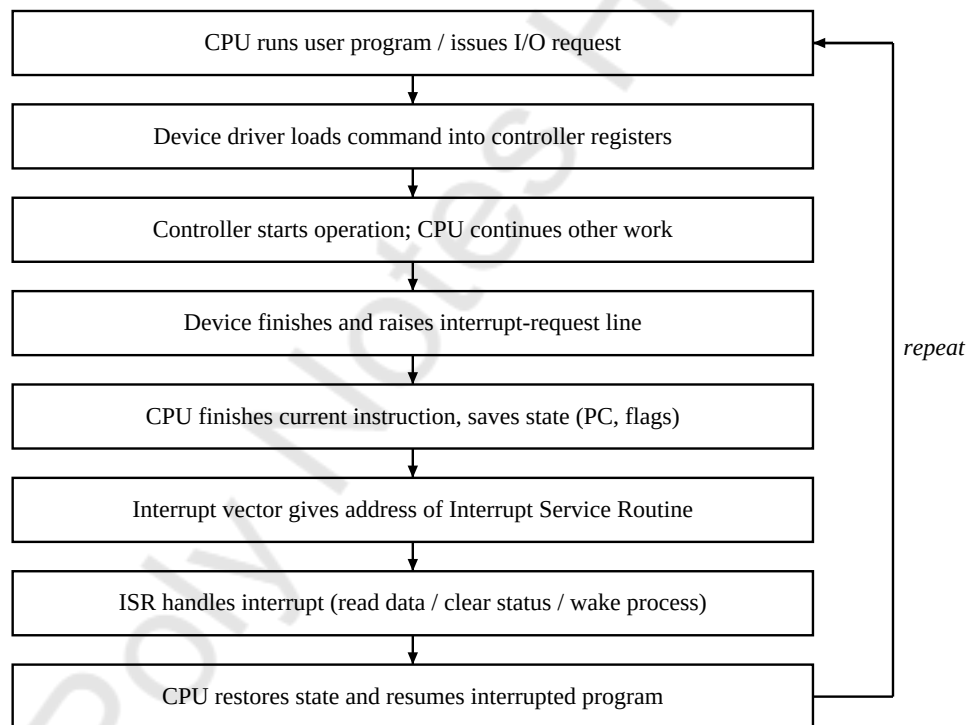


Fig. 6.2: Interrupt-driven I/O cycle

6.4.2 Important Terms

Term	Meaning
Interrupt vector	A table of addresses of interrupt handlers, indexed by the interrupt number; avoids searching for the right handler.
Interrupt chaining	When more devices than vector entries exist, each entry points to a list of handlers which are called one by one until one services the interrupt.
Maskable interrupt	Can be disabled by the CPU before critical instruction sequences (the interrupt-request line is called INTR).
Non-maskable interrupt (NMI)	Cannot be disabled; reserved for serious events such as unrecoverable memory errors.
Interrupt priority levels	Allow the CPU to defer low-priority interrupts without masking all of them and let a high-priority interrupt pre-empt a low-priority handler.
Exception	Event raised by CPU itself, such as divide-by-zero, invalid memory access or page fault.
Trap (software interrupt)	Instruction that causes a switch to kernel mode; used for system calls.

6.4.3 Polling versus Interrupt-driven I/O

Basis	Polling	Interrupt-driven I/O
Who starts the transfer service	CPU keeps checking device	Device notifies the CPU
CPU utilisation	Low (busy-wait)	High (CPU runs other tasks)
Overhead	Wasted cycles in loop	Context switch and ISR overhead
Response time	Depends on polling interval	Fast, immediate on event
Best for	Fast devices with short, frequent waits	Slow or unpredictable devices
Complexity	Simple	More complex hardware and software

6.5 Direct Memory Access (DMA)

With interrupt-driven **programmed I/O**, the CPU itself moves every byte between the controller registers and memory. For large transfers (for example a disk block) this wastes huge CPU time. **Direct Memory Access (DMA)** offloads this job to a special-purpose processor, the **DMA controller**, which transfers an entire block directly between device and main memory without CPU involvement. The CPU is interrupted only *once* per block, not once per byte.

Steps in a DMA Transfer (Disk Read)

1. The device driver is told to transfer disk data to a memory buffer at address X.
2. The driver tells the disk controller to transfer C bytes from disk to the buffer at X.
3. The CPU writes a **DMA command block** (source address, destination address, byte count) into memory and gives its address to the DMA controller, then continues other work.
4. The disk controller starts the transfer and sends each byte to the DMA controller.

5. The DMA controller seizes the memory bus, places the address and transfers the bytes directly into memory, decrementing the count C. Handshaking uses **DMA-request** and **DMA-acknowledge** lines between the DMA controller and the device controller.
6. When C reaches zero, the DMA controller interrupts the CPU to signal that the transfer is complete.

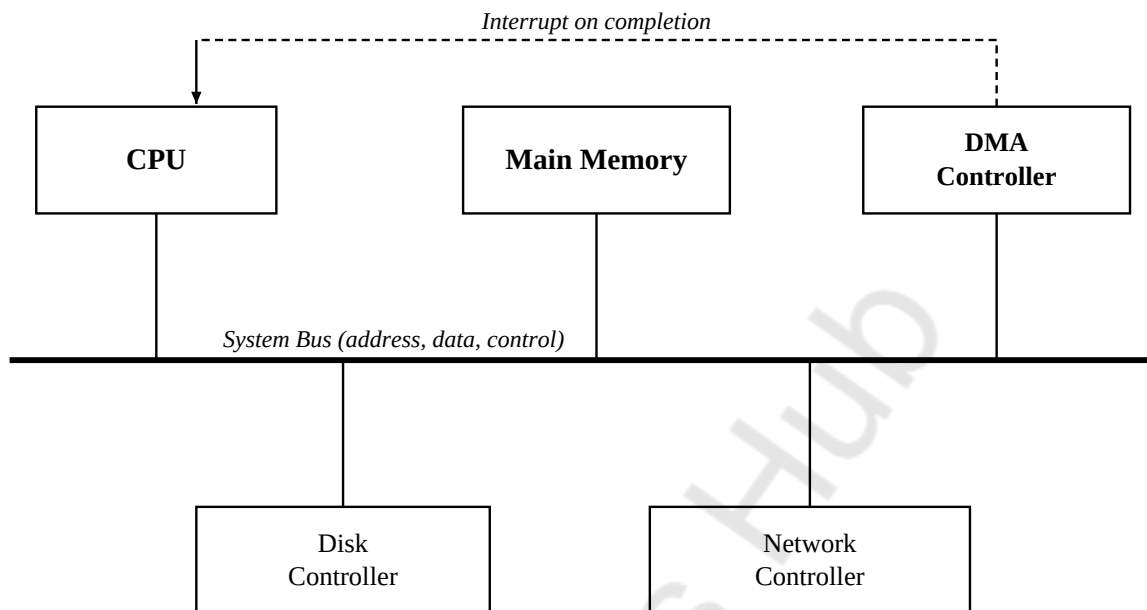


Fig. 6.3: DMA transfer and the system bus

6.5.1 DMA Concepts

- **Cycle stealing:** when the DMA controller holds the memory bus, the CPU is briefly blocked from main memory (although it can still use its cache). This slight slowing down is known as cycle stealing, yet it is still far more efficient than the CPU copying data.
- **Burst (block) mode:** the DMA controller keeps the bus for the entire block transfer.
- **Direct Virtual Memory Access (DVMA):** some systems let DMA use virtual addresses and translate them, so transfers can happen between two memory-mapped devices without the CPU.
- **Bus mastering:** modern controllers can themselves take control of the bus to transfer data.

6.5.2 Comparison of I/O Techniques

Basis	Polling (Programmed I/O)	Interrupt-driven I/O	DMA
CPU involvement	Complete, all the time	During every data unit	Only at start and end
Data path	Device - CPU - Memory	Device - CPU - Memory	Device - Memory directly
Interrupts	None	One per byte/word	One per block
Suitable for	Fast, simple devices	Slow devices (keyboard, mouse)	Bulk transfer (disk, network, audio)
Hardware cost	Lowest	Moderate	Highest (DMA controller)

6.6 Application I/O Interface

The OS must treat devices uniformly, even though they differ widely. It does this by classifying devices into a few **general kinds** and defining a standard set of system calls for each kind. The differences are encapsulated in kernel modules called **device drivers**, which present one of the standard interfaces to the rest of the kernel. A new device can then be added by simply installing its driver. Some devices (for example, those with unusual functions) are handled through an **escape (back-door) call** such as `ioctl()` in UNIX.

6.6.1 Characteristics of I/O Devices

Aspect	Variation	Example
Data-transfer mode	Character or block	Terminal (character), disk (block)
Access method	Sequential or random	Modem (sequential), CD-ROM (random)
Transfer schedule	Synchronous or asynchronous	Tape (synchronous), keyboard (asynchronous)
Sharing	Dedicated or sharable	Tape (dedicated), keyboard (sharable)
Device speed	Latency, seek time, transfer rate	Varies from bytes/s to GB/s
I/O direction	Read only, write only, read-write	CD-ROM, graphics controller, disk

6.6.2 Block and Character Devices

Block Devices

A block device stores data in fixed-size blocks, each with its own address. The interface includes commands such as `read()`, `write()` and `seek()` for random access. Applications normally access them through the file system, but some programs (such as databases) want direct control and use **raw I/O**, which gives a plain array of blocks with no buffering or locking. **Direct I/O** disables buffering and locking but keeps file-system naming. **Memory-mapped file access** maps a file into virtual memory, and the data is moved by demand paging.

Character (Stream) Devices

A character device transfers a stream of bytes one at a time, in the order received or sent. Basic calls are `get()` and `put()`. Examples are keyboards, mice, serial ports and printers. Libraries layered on top often provide line-at-a-time access with buffering and editing (for example, backspace handling on a keyboard).

Basis	Block device	Character device
Unit of transfer	Fixed-size block	Single character (byte)
Access	Random (seek possible)	Sequential only
Typical calls	read, write, seek	get, put
Buffering	Usually buffered/cached	Usually unbuffered or line-buffered
Examples	HDD, SSD, USB drive, CD-ROM	Keyboard, mouse, serial port, printer

6.6.3 Network Devices

Network I/O differs enough from disk I/O that most OSes give a different interface, usually the **socket interface**. A socket acts like a connector: an application creates a socket, connects it to a remote address, and then sends and receives packets through it. The interface also provides a **select()** function that manages a set of sockets and reports which have data waiting or space to send, avoiding polling and busy-waiting. Other approaches include pipes, FIFOs, streams and message queues.

6.6.4 Clocks and Timers

Hardware clocks and timers provide three basic functions: (1) give the **current time**, (2) give the **elapsed time**, and (3) set a **timer** to trigger operation X at time T. The hardware used for this is called a **programmable interval timer**; it can be set to wait for a given time and then generate an interrupt, either once (one-shot) or repeatedly (periodic).

- **Uses:** CPU scheduler time slices, flushing the disk cache periodically, cancelling network operations that take too long (time-outs), and process alarms.
- The kernel can simulate many virtual timers using a single hardware timer by keeping a list of requests sorted by wake-up time.
- The resolution of the hardware clock is limited (typically tens of milliseconds in older systems, much finer today) and a **high-performance event timer** may be used where precision is needed.

6.6.5 Blocking, Nonblocking and Asynchronous I/O

Type	Behaviour of the system call	Notes
Blocking I/O	The calling process is suspended (moved to the wait queue) until the I/O completes.	Easy to understand and write; most common. Process wakes when I/O completes.
Nonblocking I/O	The call returns immediately with whatever data is available (maybe none or partial).	Useful for user interfaces and video applications; usually implemented with multithreading or select().
Asynchronous I/O	The call returns immediately and the I/O runs while the process continues; completion is later signalled (flag, signal, callback).	Differs from nonblocking: the full request is performed eventually; the process is notified when the whole transfer is done.

Vectored I/O (scatter-gather): some interfaces such as *readv()* and *writelv()* let one system call perform multiple I/O operations on several buffers, which cuts system-call and context-switch overhead and guarantees atomicity.

6.7 Kernel I/O Subsystem

The kernel provides many services related to I/O. Several of them (scheduling, buffering, caching, spooling, device reservation and error handling) are built on the hardware and device-driver infrastructure. This set of services is called the **kernel I/O subsystem**. Another job of this subsystem is **protection**: all I/O instructions are privileged, so a user process must request I/O through system calls, which lets the OS check validity and prevent illegal access.

6.7.1 I/O Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. Order chosen by applications is rarely the best. Scheduling can improve overall system performance, share device access fairly among processes, and reduce average waiting time. It is implemented by maintaining a **wait queue** for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device and the I/O scheduler rearranges the queue (for example, by seek-time for disk).

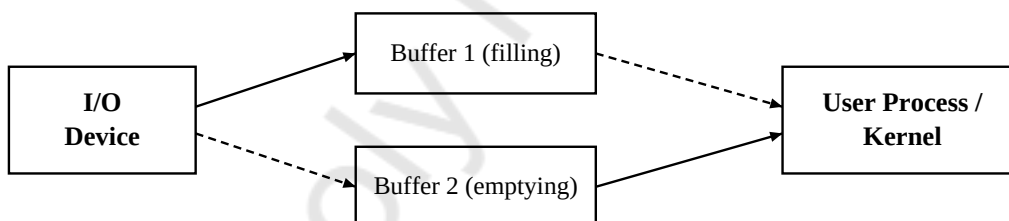
- The kernel keeps a **device-status table**, with an entry for each device showing its type, address and state (idle, busy, not functioning). If the device is busy, the entry also holds the list of waiting requests.
- Scheduling may favour latency-sensitive requests, such as those from virtual memory, over bulk requests.
- Some kernels try to provide **fairness** so no one application receives poor service, or give priority service for delay-sensitive requests.

6.7.2 Buffering

A **buffer** is a memory area that stores data while it is being transferred between two devices or between a device and an application.

Reasons for Buffering

- **Speed mismatch** between producer and consumer (for example, a modem and a disk): data from the slow device is collected in a buffer, and when the buffer is full it is written to the disk in one operation. **Double buffering** allows the next buffer to be filled while the first is being written.
- **Different data-transfer sizes**: common in computer networking, where messages are fragmented and reassembled using buffers at both ends.
- **Maintain copy semantics**: when an application calls write(), the OS copies the application data into a kernel buffer before returning, so later changes to the application buffer do not change what is written to disk.



Roles of the two buffers swap each time (double buffering)

Fig. 6.4: Double buffering between a device and a process

6.7.3 Caching

A **cache** is a region of fast memory that holds *copies* of data. Access to the cached copy is more efficient than access to the original. For example, instructions of the running process are stored on disk, cached in physical memory, and again copied into the CPU's secondary and primary caches.

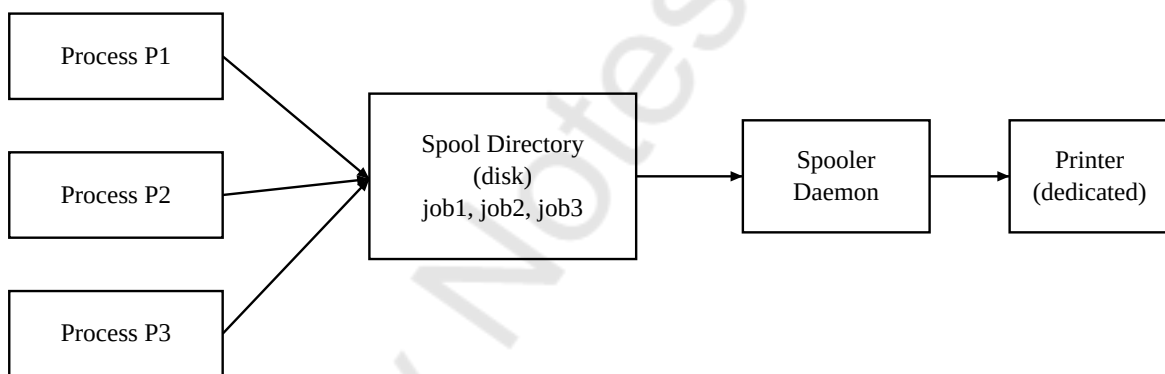
Basis	Buffer	Cache
Contents	May hold the only existing copy of data	Holds a copy of data that exists elsewhere

Basis	Buffer	Cache
Main purpose	Handle speed or size mismatch, copy semantics	Improve access speed
Location	Main memory area	Fast memory (RAM, CPU cache)
Example	Print data waiting for the printer	Disk blocks kept in the buffer cache

Sometimes a region of memory can be used for both purposes. For instance, to preserve copy semantics and enable efficient scheduling of disk I/O, the OS uses buffers in main memory to hold disk data; these buffers are also used as a cache to improve I/O efficiency for files shared by applications or being written and re-read rapidly.

6.7.4 Spooling and Device Reservation

A **spool** (Simultaneous Peripheral Operations On-Line) is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although a printer can serve only one job at a time, several applications may wish to print at the same moment. The OS solves this by **intercepting** all output to the printer. Each application's output is spooled to a separate disk file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer, and a **spooler daemon** (system process) copies queued files to the printer one at a time.



Jobs queue on disk; printer used by one job at a time

Fig. 6.5: Spooling of print jobs

Device reservation: some devices, such as tape drives and printers, cannot usefully multiplex requests from many applications. An alternative to spooling is to let one process **coordinate exclusive access**. Some operating systems provide support for exclusive device access by enabling a process to allocate an idle device and deallocate it when no longer needed (for example, the *open()* call can fail if the device is already in use). The danger is **deadlock** when processes hold one device while waiting for another, so the OS or applications must avoid it.

6.7.5 Error Handling

An OS that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical glitch. Devices and I/O transfers can fail in many ways, either for **transient** reasons (such as network overload) or **permanent** reasons (such as a disk-controller defect).

- **Transient failures:** OS can often recover by **retrying** the operation (for example, a failed disk read() is retried, or a failed network send() causes resend()).

- **Permanent failures:** recovery is usually not possible; the error is reported to the caller.
- System calls return an **error bit** or **status code**, such as *errno* in UNIX, indicating success or failure. Some hardware (for example, SCSI devices) reports detailed **sense keys** and additional sense codes.
- Many systems keep an **error log** of problems for diagnosis.
- A process that tries illegal I/O (for instance, access to a privileged instruction) is stopped by hardware protection and the OS signals the error.

6.7.6 I/O Protection and Kernel Data Structures

- **I/O protection:** all I/O instructions are defined as **privileged**, so users cannot issue them directly; memory-mapped and I/O-port memory locations are protected from user access by the memory-protection system. User programs use system calls to ask the kernel to perform I/O.
- **Kernel data structures:** the kernel keeps state about I/O components, such as the open-file table, network connections and character-device state. Many complicated structures track buffers, memory allocation and dirty blocks. Some OSes use **object-oriented methods** and message passing for I/O (for example, Windows uses message passing; the I/O request is converted to a message sent through the kernel to the I/O manager and then to the device driver).

6.7.7 Summary of Kernel I/O Services

Service	What it does	Problem solved
Scheduling	Orders requests in a per-device queue	Fairness, lower waiting time, better throughput
Buffering	Holds data during transfer	Speed/size mismatch, copy semantics
Caching	Keeps fast copy of data	Faster repeated access
Spooling	Queues output for a non-shareable device	Interleaved output of multiple jobs
Device reservation	Gives exclusive access to one process	Sharing of non-multiplexable devices
Error handling	Retries, reports errors, logs	Reliability
Protection	Privileged I/O, system calls	Security and integrity

6.8 Life Cycle of an I/O Request

Consider a process that issues a blocking *read()* on a file descriptor of an already opened file. The OS goes through the following steps to convert the request into hardware operations:

1. The process issues a blocking *read()* system call.
2. The kernel I/O subsystem checks the parameters. If the data is already available in the buffer cache, it is returned and the request is complete.
3. Otherwise, a physical I/O must be performed: the process is removed from the run queue and placed in the wait queue for the device, and the I/O request is scheduled.
4. The I/O subsystem sends the request to the device driver, which allocates kernel buffer space and schedules the I/O.
5. The driver sends commands to the device controller by writing into the control registers.
6. The controller operates the device hardware to perform the data transfer (often using DMA).

7. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. DMA controller generates an interrupt when the transfer completes.
8. The correct interrupt handler receives the interrupt via the interrupt-vector table, stores any necessary data, signals the device driver and returns from the interrupt.
9. The device driver receives the signal, determines which I/O request completed, determines the request status and signals the kernel I/O subsystem that the request is complete.
10. The kernel transfers data or return codes to the address space of the requesting process, and moves the process from the wait queue back to the ready queue. The process is unblocked and resumes execution upon scheduling.

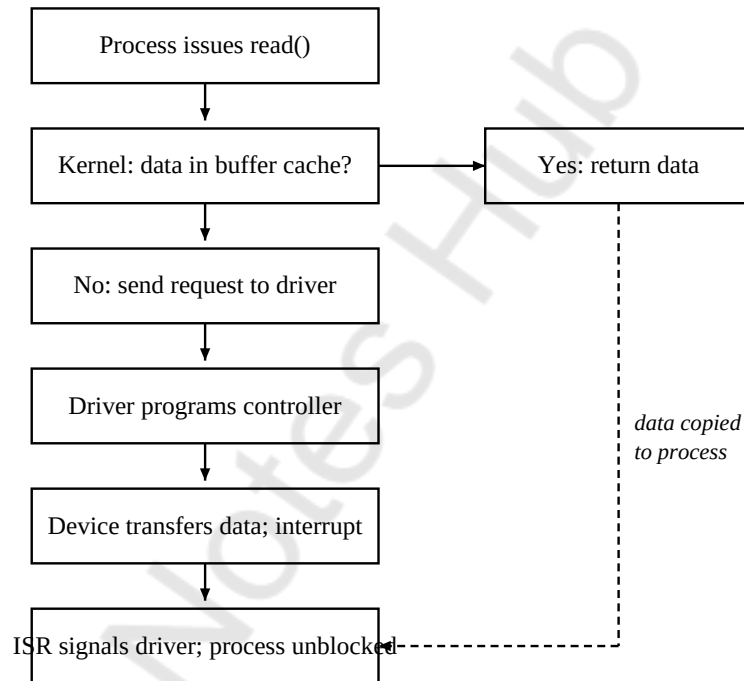


Fig. 6.6: Flow of a blocking read() request

6.9 Performance

I/O is a major factor in system performance. It places heavy demands on the CPU to execute device-driver code and to schedule processes fairly and efficiently as they block and unblock. The resulting **context switches** stress the CPU and its hardware caches. I/O also exposes inefficiencies in the interrupt-handling mechanisms of the kernel, and loads down the memory bus during data copies between controllers and physical memory and between kernel buffers and application space.

6.9.1 Principles to Improve I/O Efficiency

- **Reduce the number of context switches** (for example, by using polling for fast devices or by handling more work in a single interrupt).
- **Reduce the number of times data must be copied** in memory while passing between device and application (copy cost is high; use techniques like zero-copy and shared memory).
- **Reduce the frequency of interrupts** by using large transfers, smart controllers and polling (if busy-waiting can be minimised).

- **Increase concurrency** by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
- **Move processing primitives into hardware** to allow their operation in device controllers to be concurrent with CPU and bus operation.
- **Balance CPU, memory subsystem, bus and I/O performance**, because an overload in any one area causes idleness in others.

6.9.2 Where to Implement I/O Functionality

Designers must decide at which level a new I/O function should be implemented. Experiments usually start in the application level and then move toward the kernel and hardware as performance demands rise:

Level	Advantages	Disadvantages
Application code	Flexible, easy to experiment, bugs do not crash the system, no new kernel needed	Overhead of context switch, cannot use kernel data structures and services
Kernel	Efficient, uses kernel services, quick access to internal data	Harder to debug, a bug can crash the whole system, kernel changes needed
Device driver	Good performance, can be loaded as a module	Bugs in the driver can crash the OS; needs driver for each device
Device controller (hardware)	Fastest, operates concurrently with the CPU	Costly, inflexible, difficult to improve or fix bugs

6.9.3 Choice among I/O Methods

Situation	Better technique
Device is fast and wait is shorter than a context switch	Polling
Device is slow or events are rare and unpredictable	Interrupt-driven I/O
Large block transfers (disk, network, audio)	DMA
Very high network traffic	Mix of interrupt and polling (interrupt coalescing / adaptive polling)

6.10 Chapter Summary and Quick Revision

- The I/O subsystem hides device complexity through drivers, controllers and uniform interfaces.
- A controller has data-in, data-out, status and control registers; accessed by port-mapped or memory-mapped I/O.
- **Polling** = CPU repeatedly checks busy bit; simple but wastes cycles.
- **Interrupt** = device signals the CPU; handler found through the interrupt vector; priorities and masking possible.
- **DMA** = controller moves a block between device and memory; CPU interrupted once per block; cycle stealing.
- Device classes: block (random access, blocks), character (stream), network (sockets, select()), clocks/timers (current time, elapsed time, trigger).
- Blocking: process waits. Nonblocking: returns immediately with available data. Asynchronous: request runs while process continues, notified at completion.

- Kernel I/O subsystem: scheduling, buffering, caching, spooling, device reservation, error handling, protection.
- Buffer: temporary storage possibly the only copy. Cache: faster copy of existing data. Spool: buffer for output to non-interleavable device.
- Performance improves by fewer context switches, fewer copies, fewer interrupts, more concurrency and balanced hardware.

Poly Notes Hub

6.11 Multiple Choice Questions (15)

Choose the correct option for each question. The answer key is given in Section 6.12.

Q1. Which register of a device controller is written by the host to start a command?

- (a) Data-in register
- (b) Data-out register
- (c) Status register
- (d) Control register

Q2. Repeatedly checking the busy bit of a device by the CPU is known as:

- (a) DMA
- (b) Polling
- (c) Spooling
- (d) Caching

Q3. Which of the following is NOT a feature of interrupt-driven I/O?

- (a) Device notifies the CPU
- (b) Interrupt vector is used
- (c) CPU busy-waits in a loop
- (d) Interrupt service routine is executed

Q4. The table that contains the addresses of interrupt handlers is called the:

- (a) Page table
- (b) Interrupt vector
- (c) Device-status table
- (d) File-allocation table

Q5. An interrupt that cannot be disabled by the CPU is called:

- (a) Maskable interrupt
- (b) Software interrupt
- (c) Non-maskable interrupt
- (d) Vectored interrupt

Q6. In DMA, the CPU is interrupted:

- (a) After every byte transferred
- (b) After every word transferred
- (c) Once after the whole block is transferred
- (d) Never

Q7. The temporary stealing of memory bus cycles by the DMA controller from the CPU is called:

- (a) Cycle stealing
- (b) Thrashing
- (c) Bus arbitration
- (d) Swapping

Q8. Which system calls are commonly used with character devices?

- (a) read, write, seek
- (b) get, put
- (c) socket, bind
- (d) mmap, munmap

Q9. The function used to monitor a set of network sockets for readiness is:

- (a) ioctl()
- (b) select()
- (c) fork()
- (d) readv()

Q10. Which of the following is a function of a programmable interval timer?

- (a) Allocating memory
- (b) Generating an interrupt after a specified time
- (c) Scheduling disk requests
- (d) Compressing files

Q11. In which type of I/O does the system call return immediately with whatever data is available?

- (a) Blocking I/O
- (b) Nonblocking I/O
- (c) Synchronous I/O
- (d) Spooled I/O

Q12. Double buffering is used mainly to:

- (a) Increase disk capacity
- (b) Decouple producer and consumer by overlapping filling and emptying of buffers
- (c) Encrypt data
- (d) Reduce number of devices

Q13. A region of fast memory holding a copy of data that is stored elsewhere is called a:

- (a) Buffer
- (b) Spool
- (c) Cache
- (d) Register

Q14. Which technique is used to handle simultaneous print requests from many processes?

- (a) Polling
- (b) Spooling
- (c) Paging
- (d) Swapping

Q15. Which of the following will help to improve I/O performance?

- (a) Increase number of context switches
- (b) Increase copying of data
- (c) Reduce the frequency of interrupts and data copies
- (d) Disable DMA

6.12 Answer Key

Q. No.	Answer	Short Reason
1	(d)	Control register is written by host to start a command.
2	(b)	Polling (busy-waiting).
3	(c)	Busy-waiting belongs to polling, not interrupts.
4	(b)	Interrupt vector holds handler addresses.
5	(c)	NMI is reserved for critical errors.
6	(c)	DMA interrupts once per block.
7	(a)	Cycle stealing.
8	(b)	get() and put() for character devices.
9	(b)	select() waits on multiple sockets.
10	(b)	Timer generates interrupt after a set interval.
11	(b)	Nonblocking I/O returns at once with available data.
12	(b)	Fill one buffer while the other is emptied.
13	(c)	Cache holds a copy of data.
14	(b)	Spooling queues print jobs on disk.
15	(c)	Fewer interrupts and copies reduce overhead.

6.13 Broad Questions (10)

1. Explain the I/O hardware of a computer system. Describe the registers of a device controller and the methods of addressing them (port-mapped and memory-mapped I/O).
2. What is polling? Explain the handshaking protocol between host and controller with the help of the busy and command-ready bits. State its advantages and disadvantages.
3. Explain the interrupt mechanism in detail. Describe interrupt vector, interrupt chaining, maskable and non-maskable interrupts, and interrupt priorities with a neat diagram.
4. What is Direct Memory Access? Explain the steps of a DMA transfer with a neat diagram. Compare polling, interrupt-driven I/O and DMA.
5. Describe the application I/O interface. Explain the differences between block devices and character devices with examples.
6. Explain network devices and the socket interface. Discuss the role of clocks and programmable interval timers in an operating system.
7. Differentiate between blocking, nonblocking and asynchronous I/O with examples. What is vectored I/O?
8. Explain the kernel I/O subsystem. Discuss I/O scheduling and the device-status table.
9. What is buffering? Why is it needed? Distinguish between buffer and cache. Explain spooling and device reservation with a diagram, and discuss error handling in I/O.
10. Describe the life cycle of an I/O request. Discuss the factors affecting I/O performance and the principles used to improve it.