

I/O Systems



**Politecnico
di Torino**

Department of Control and
Computer Engineering



System Programming - Sarah Azimi

CAD & Reliability Group
DAUIN- Politecnico di Torino

Chapter 13: I/O Systems

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- STREAMS
- Performance

Objectives

- Explore the structure of an operating system's I/O subsystem
- Discuss the principles of I/O hardware and its complexity
- Provide details of the performance aspects of I/O hardware and software

Overview

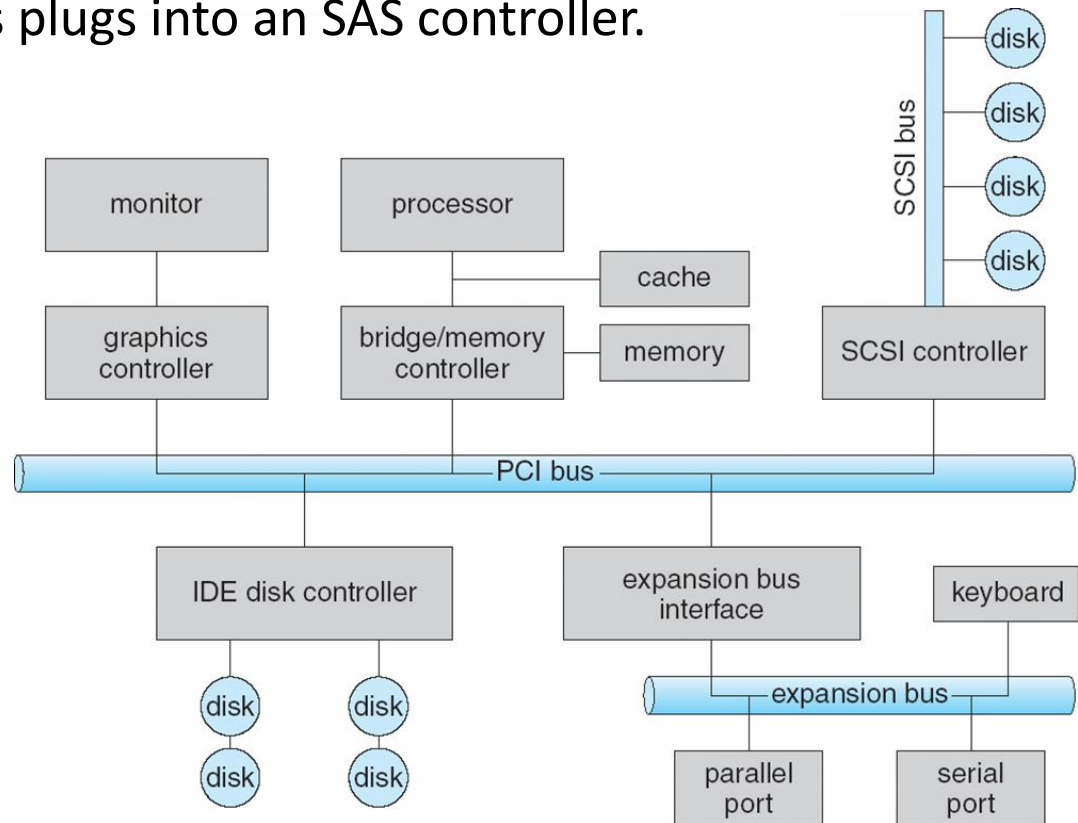
- I/O management is a major concern of operating system design and operation
 - Important aspect of computer operation
 - I/O devices vary greatly therefore, various methods are required to control them
 - Performance management
 - New types of devices frequent
- Ports, busses, device controllers connect to various devices
- To encapsulate the details and oddities of different devices, the kernel of an operating system is structured to use **device driver** modules.
 - Device drivers present uniform device-access interface to I/O subsystem

I/O Hardware

- Incredible variety of I/O devices
 - Storage
 - Transmission
 - Human-interface
- A device communicate with a computer system by sending signals.
 - **Port** – connection point for device for example serial port
 - **Bus** used in most computers today, such as PCI, a set of wires and rigidly defined protocol that specify a set of messages.
 - Bus - daisy chain or shared direct access
 - PCIe bus, connects the processor-memory subsystem to fast devices.
 - expansion bus connects relatively slow devices such as keyboard and USB ports.
 - **Controller (host adapter)** – is a collection of electronics that operate a port, a bus, a device that controls the signals.
 - Sometimes integrated
 - Sometimes separate circuit board (host bus adapter - HBA) that connects to the bus in the computer.
 - It contains processor, microcode, private memory, bus controller, etc
 - Some talk to per-device controller with bus controller, microcode, memory, etc

A Typical PC Bus Structure

- The **PCIe bus**, the common PC system bus, connects the processor-memory subsystem to fast devices and an **expansion bus** connects relatively slow devices such as keyboard.
- In the lower left portion of the figure, four disks are connected together on a **serial-attached SCSI (SAS)** bus plugs into an SAS controller.



I/O Hardware

- How does the processor give commands and data to a controller to accomplish an I/O transfer?
 - The controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers.
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
 - Data-in register: used by the host to get input
 - data-out register: written by the host to send output
 - status register: contains bits that can be read by the host which indicates states, such as whether the current command had been completed.
 - Control register: written by the host to start a command or to change the mode of a device
 - Typically 1-4 bytes, or FIFO buffer

I/O Hardware

- Devices have addresses, used by
 1. Direct I/O instructions that specify the transfer of a byte or a word to an I/O port address.
 2. **Memory-mapped I/O**
 - Device control registers are mapped into the address space of the processor.
 - The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers at their mapped locations in physical memory.

Device I/O Port Locations on PCs (partial)

- The usual I/O port addresses for PCs:

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Polling

- Assuming that 2 bits are used for coordinating the relationship between controller and host.
- For each byte of I/O
 1. The host repeatedly reads the busy bit until that bit becomes clear, Read busy bit from status register until 0.
 2. The host sets the write bit and writes a byte into the *data-out* register.
 3. The host sets *command-ready* bit.
 4. When the controller notices that the *command-ready* bit is set, it sets the *busy* bit.
 5. The controller reads the command register and sees the write command. It reads the *data-out* register to get the byte and does the I/O to the device.
 6. Controller clears *command-ready* bit, clears the *error bit* in the status register to indicate that the device I/O succeeded, and clears the *busy* bit to indicate that it is finished.
- In step 1, the host is **busy-waiting** or **pooling**: it is in a loop, reading the status register over and over until the busy bit becomes clear.

Polling

- Step 1 is **busy-wait** cycle to wait for I/O from device
 - Reasonable if device is fast
 - But inefficient if device slow
 - CPU switches to other tasks?
 - But if miss a cycle data overwritten / lost

Interrupts

- In many computer architectures, three CPU-instruction cycles are sufficient to poll a device: read a device register, logic and to extract a status bit, and branch if not zero.
- Pooling is not efficient when it is attempted repeatedly but not find a device ready for service.
- Therefore, it is more efficient to arrange for the hardware controller to notify the CPU when the device becomes ready for service, rather than to require the CPU to poll repeatedly for an I/O completion.
- The hardware mechanism that enable a device to notify the CPU is called an **interrupt**.

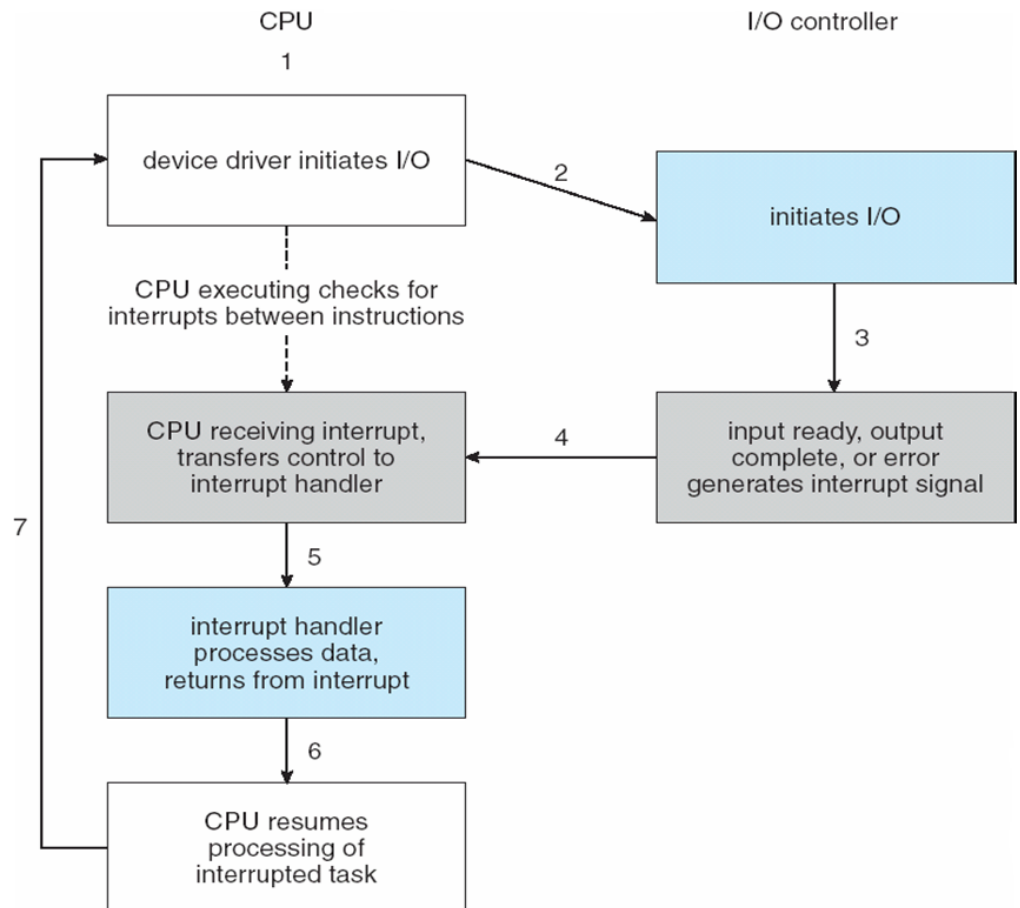
Interrupts

- The CPU hardware has a wire called the **interrupt-request line** that the CPU sense after executing every structure.
- CPU detecting that a controller has asserted a signal on this line, it performs a jump to **Interrupt handler**.
 - Determines the cause of the interrupt and performs the necessary processing.
- The device controller **raises** an interrupt by asserting a signal on the interrupt request line, the CPU **catches** the interrupt and dispatches it to the interrupt handler and the handler clear the interrupt by servicing the device.
- number

Interrupts

- CPUs have two interrupt line
 - **Nonmaskable**: reserved for events such as unrecoverable memory errors.
 - **Maskable**: it can be turned off by the CPU before the execution of critical instructions that must not be interrupted.
- The interrupt mechanism accept an address that selects a specific interrupt-handling routine from a small set. This address is an offset in a table called the **interrupt vector**.
 - **Interrupt vector** to dispatch interrupt to correct handler
 - Context switch at start and end
 - Based on priority
 - Some nonmaskable
 - Interrupt chaining if more than one device at same interrupt number

Interrupt-Driven I/O Cycle



Intel Pentium Processor Event-Vector Table

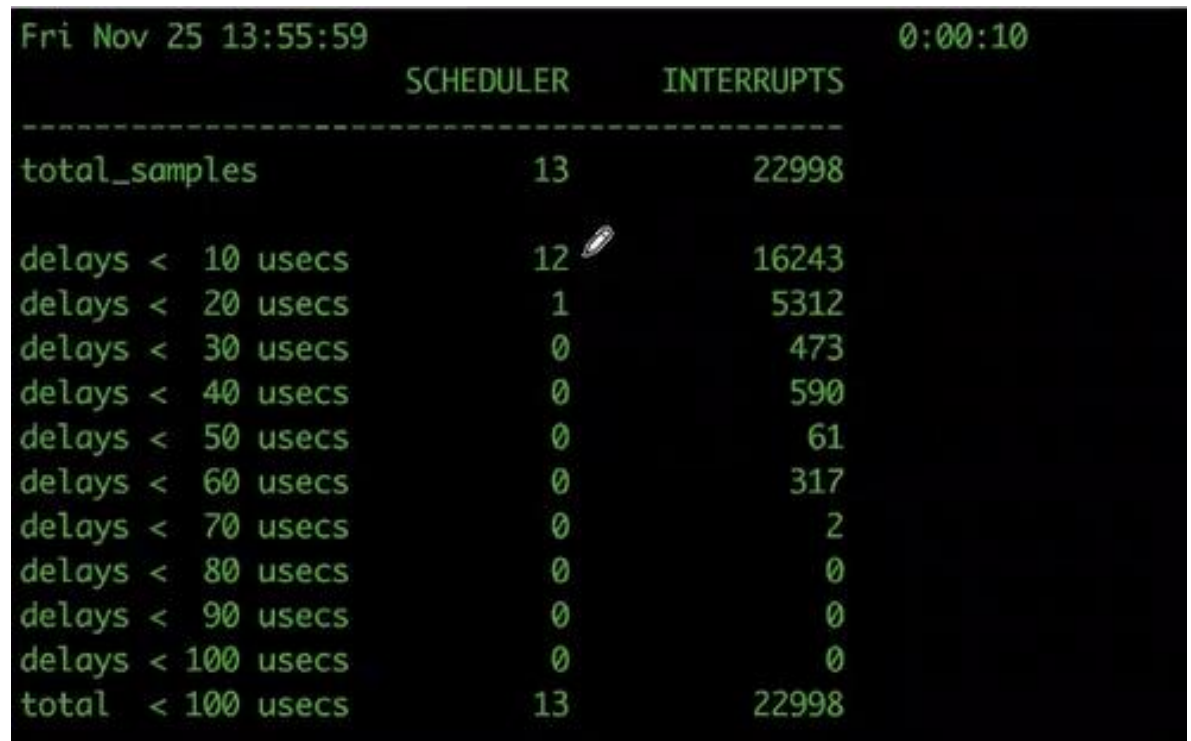
vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Interrupts

- Interrupt mechanism also used for **exceptions**
 - Terminate process, crash system due to hardware error
- Page fault executes when memory access error
- System call executes via **trap** to trigger kernel to execute request
- Multi-CPU systems can process interrupts concurrently
 - If operating system designed to handle it
- Used for time-sensitive processing, frequent, must be fast

Latency

- Stressing interrupt management because even single-user systems manage hundreds or interrupts per second and servers hundreds of thousands.
- For example, a quiet macOS desktop generated 23,000 interrupts over 10 seconds.



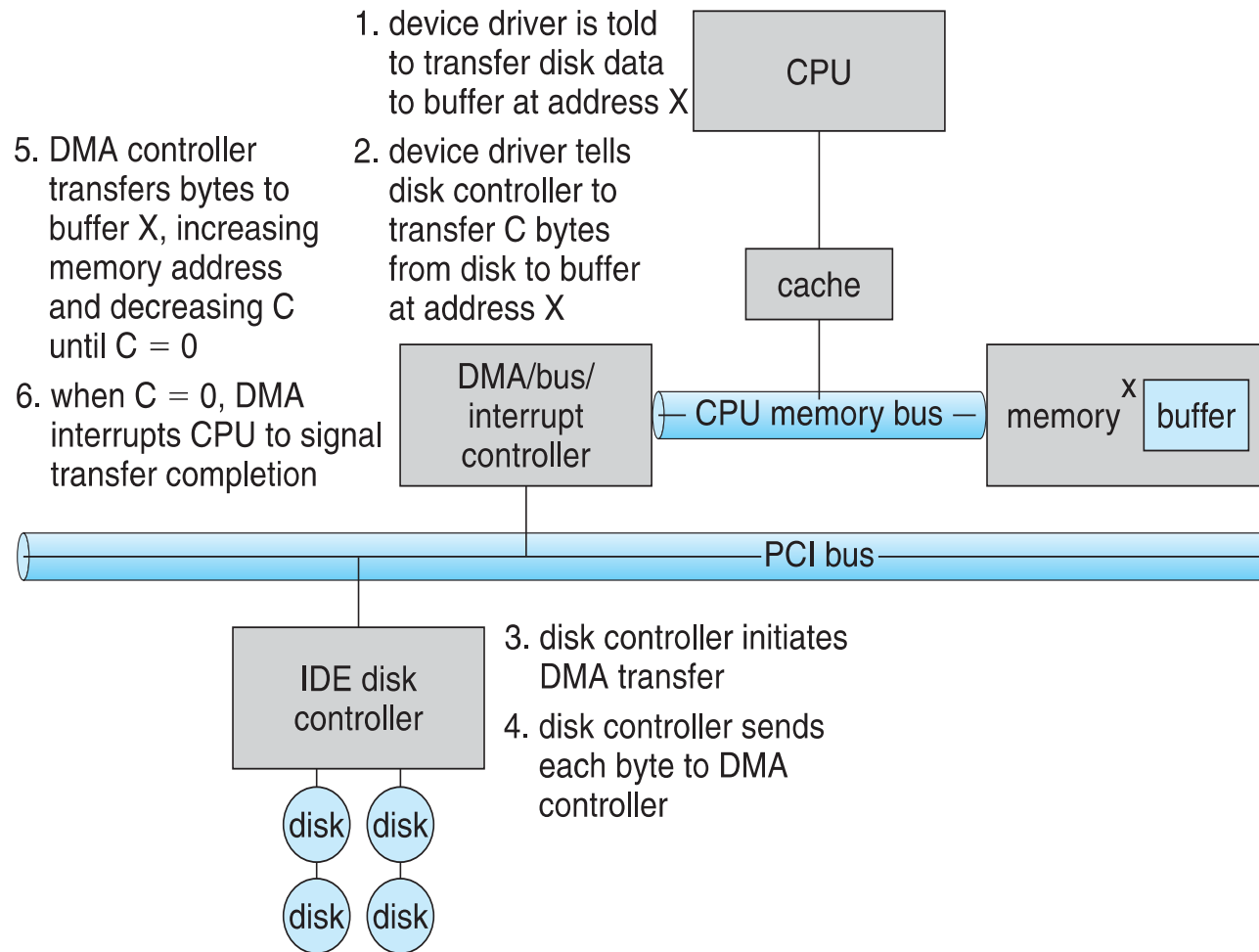
The screenshot shows a terminal window with a dark background and green text. At the top left, it displays the date and time 'Fri Nov 25 13:55:59'. At the top right, it shows a timer '0:00:10'. The main content is a table with two columns: 'SCHEDULER' and 'INTERRUPTS'. The table is separated from the header by a dashed line. The data rows show 'total_samples', various delay thresholds (from 10 to 100 usecs), and a 'total' row. The 'SCHEDULER' column has values 13, 12, 1, 0, 0, 0, 0, 0, 0, 0, 0, and 13. The 'INTERRUPTS' column has values 22998, 16243, 5312, 473, 590, 61, 317, 2, 0, 0, 0, and 22998.

	SCHEDULER	INTERRUPTS
total_samples	13	22998
delays < 10 usecs	12	16243
delays < 20 usecs	1	5312
delays < 30 usecs	0	473
delays < 40 usecs	0	590
delays < 50 usecs	0	61
delays < 60 usecs	0	317
delays < 70 usecs	0	2
delays < 80 usecs	0	0
delays < 90 usecs	0	0
delays < 100 usecs	0	0
total < 100 usecs	13	22998

Direct Memory Access

- Used to avoid **programmed I/O** (one byte at a time) for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
 - Source and destination addresses
 - Read or write mode
 - Count of bytes
 - Writes location of command block to DMA controller
 - Bus mastering of DMA controller – grabs bus from CPU
 - **Cycle stealing** from CPU but still much more efficient
 - When done, interrupts to signal completion
- Version that is aware of virtual addresses can be even more efficient - **DVMA**

Six Step Process to Perform DMA Transfer



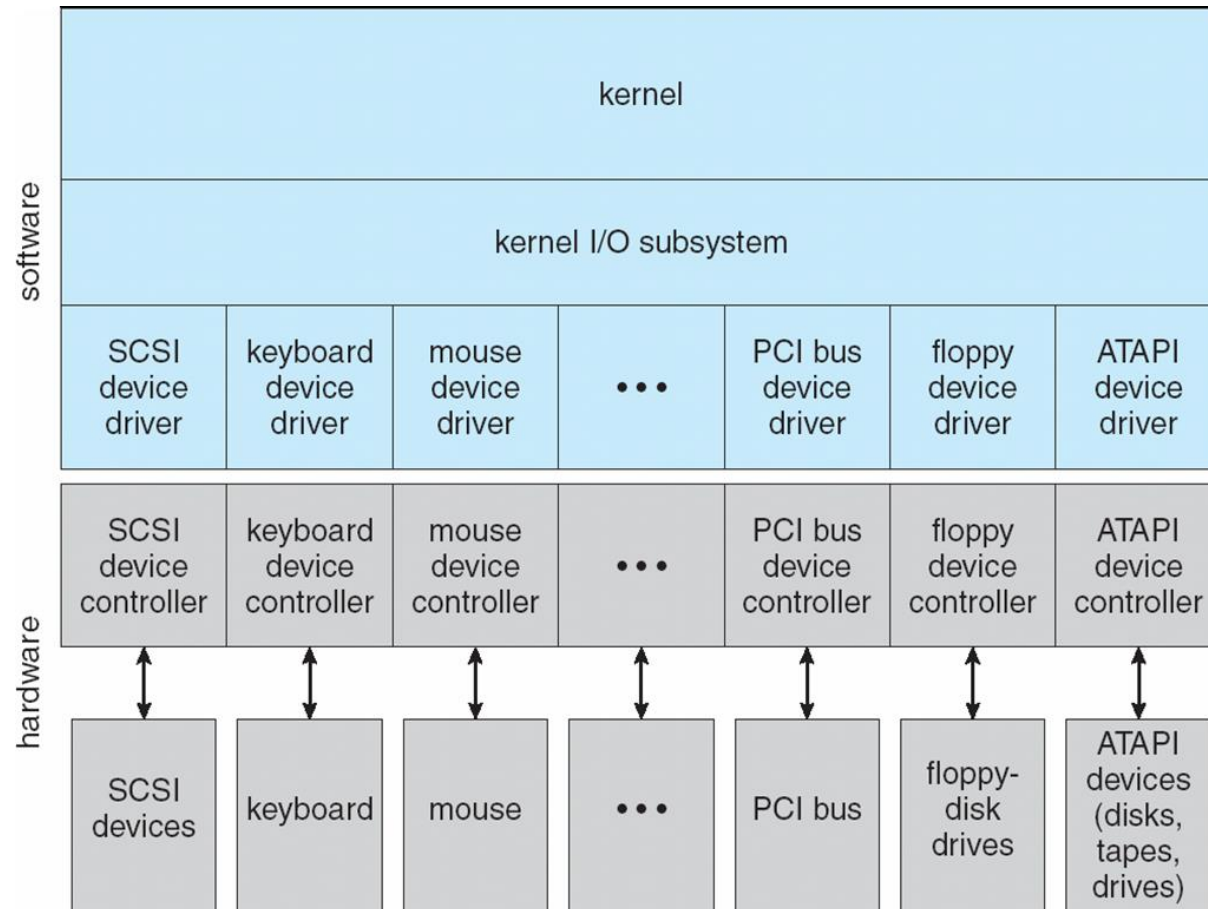
Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- Devices vary in many dimensions
 - **Character-stream** or **block**
 - **Sequential** or **random-access**
 - **Sharable** or **dedicated**
 - **Speed of operation**
 - **read-write, read only, or write only**

Application I/O Interface

- Devices vary in many dimensions
 - **Character-stream** or **block**
 - A character-stream device transfers bytes one by one, whereas block device transfer a block of bytes as a unit.
 - **Sequential** or **random-access**
 - A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.
 - **Synchronous** or **asynchronous** (or both)
 - A synchronous device performs data transfers with predictable response times, in coordination with other aspects of the system. As asynchronous device exhibits irregular or unpredictable response times not coordinated with other computer events.
 - **Sharable** or **dedicated**
 - A sharable device can be used concurrently by several processes or threads, a dedicated device no.
 - **Speed of operation**
 - Device speed range from a few bytes per second to gigabytes per second.
 - **read-write, read only, or write only**
 - Some devices perform both input and output, but others support only one data transfer direction.

A Kernel I/O Structure



Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Characteristics of I/O Devices (Cont.)

- Subtleties of devices handled by device drivers
- Broadly I/O devices can be grouped by the OS into
 - Block I/O
 - Character I/O (Stream)
 - Memory-mapped file access
 - Network sockets
- For direct manipulation of I/O device specific characteristics, usually an escape / back door
 - Unix `ioctl()` call to send arbitrary bits to a device control register and data to device data register

Block and Character Devices

- Block devices include disk drives
 - Commands include read, write, seek
 - **Raw I/O, direct I/O**, or file-system access
 - Memory-mapped file access possible
 - File mapped to virtual memory and clusters brought via demand paging
 - DMA
- Character devices include keyboards, mice, serial ports
 - Commands include `get()`, `put()`
 - Libraries layered on top allow line editing

Network Devices

- Varying enough from block and character to have own interface
- Linux, Unix, Windows and many others include **socket** interface
 - Separates network protocol from network operation
 - Includes **select()** functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

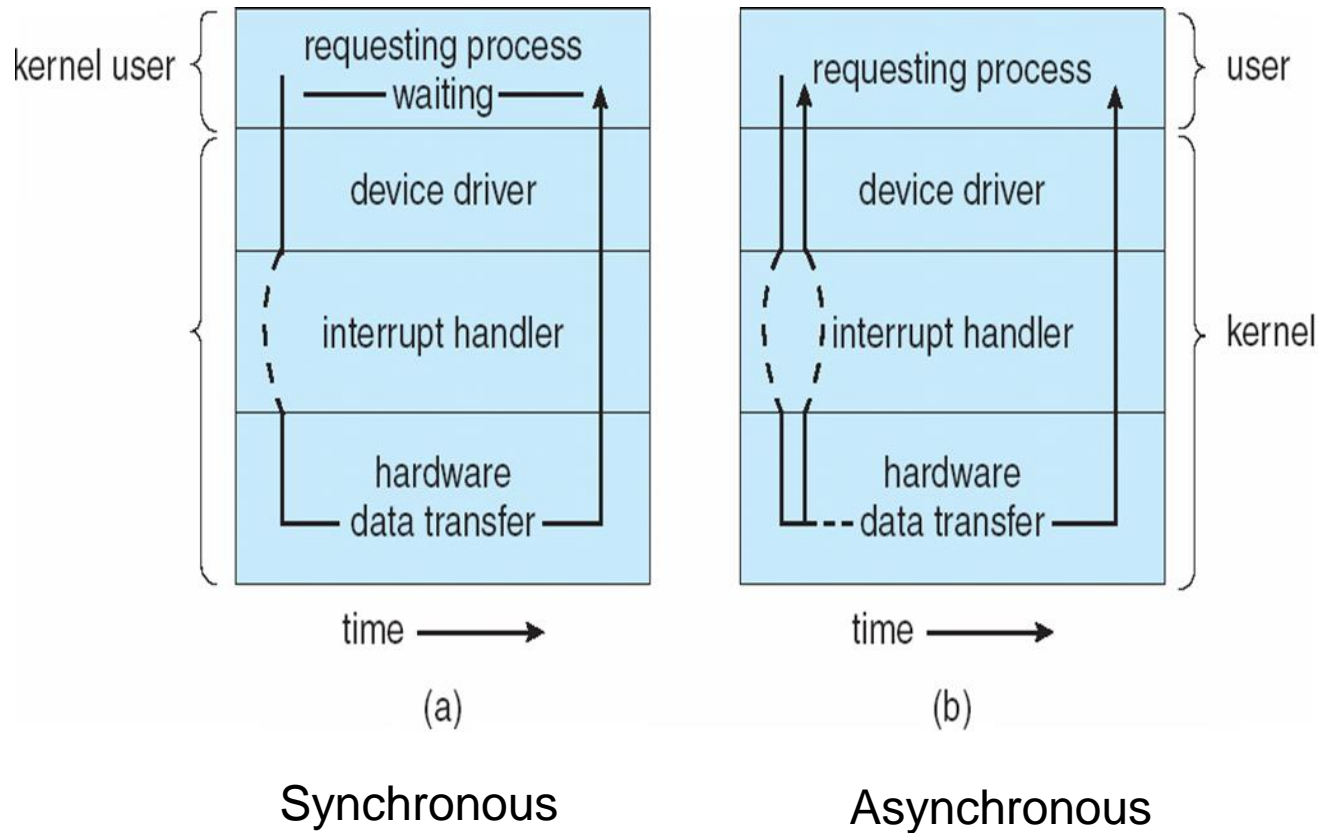
Clocks and Timers

- Provide current time, elapsed time, timer
- Normal resolution about 1/60 second
- Some systems provide higher-resolution timers
- **Programmable interval timer** used for timings, periodic interrupts
- `ioctl()` (on UNIX) covers odd aspects of I/O such as clocks and timers

Nonblocking and Asynchronous I/O

- **Blocking** - process suspended until I/O completed
 - Easy to use and understand
 - Insufficient for some needs
- **Nonblocking** - I/O call returns as much as available
 - User interface, data copy (buffered I/O)
 - Implemented via multi-threading
 - Returns quickly with count of bytes read or written
 - **select()** to find if data ready then **read()** or **write()** to transfer
- **Asynchronous** - process runs while I/O executes
 - Difficult to use
 - I/O subsystem signals process when I/O completed

Two I/O Methods



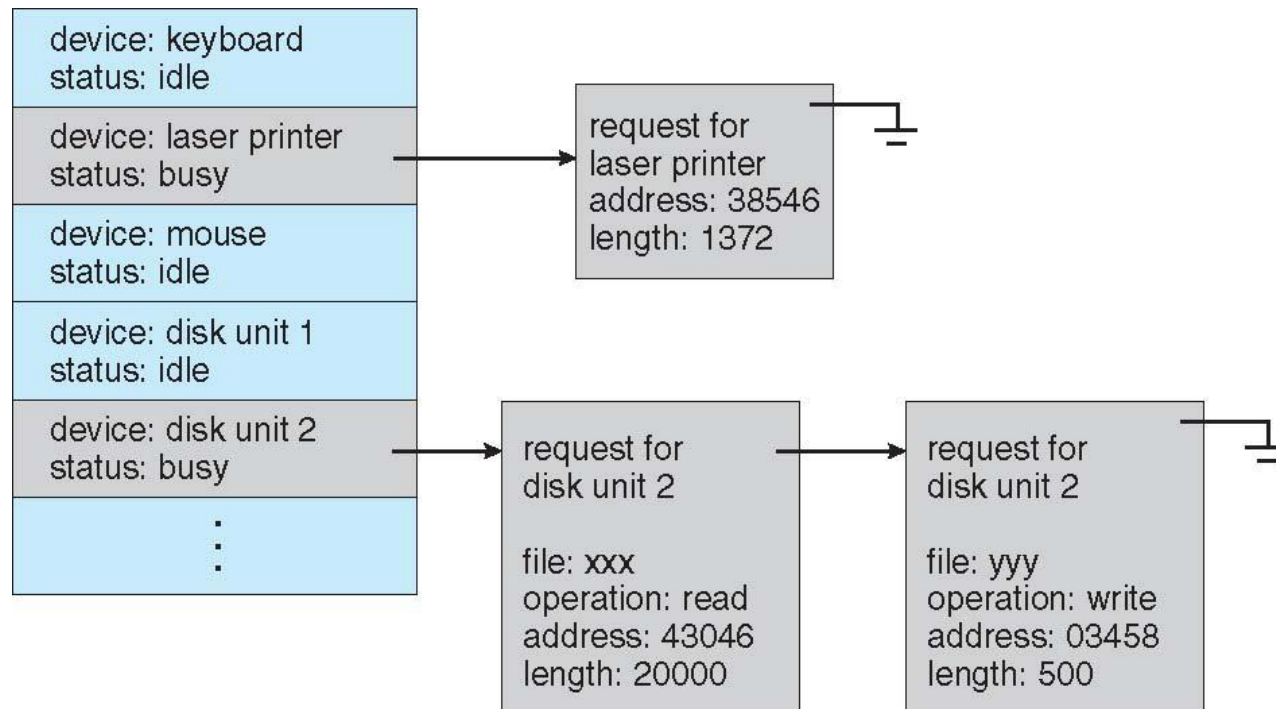
Vectored I/O

- **Vectored I/O** allows one system call to perform multiple I/O operations
- For example, Unix **readve()** accepts a vector of multiple buffers to read into or write from
- This scatter-gather method better than multiple individual I/O calls
 - Decreases context switching and system call overhead
 - Some versions provide atomicity
 - Avoid for example worry about multiple threads changing data as reads / writes occurring

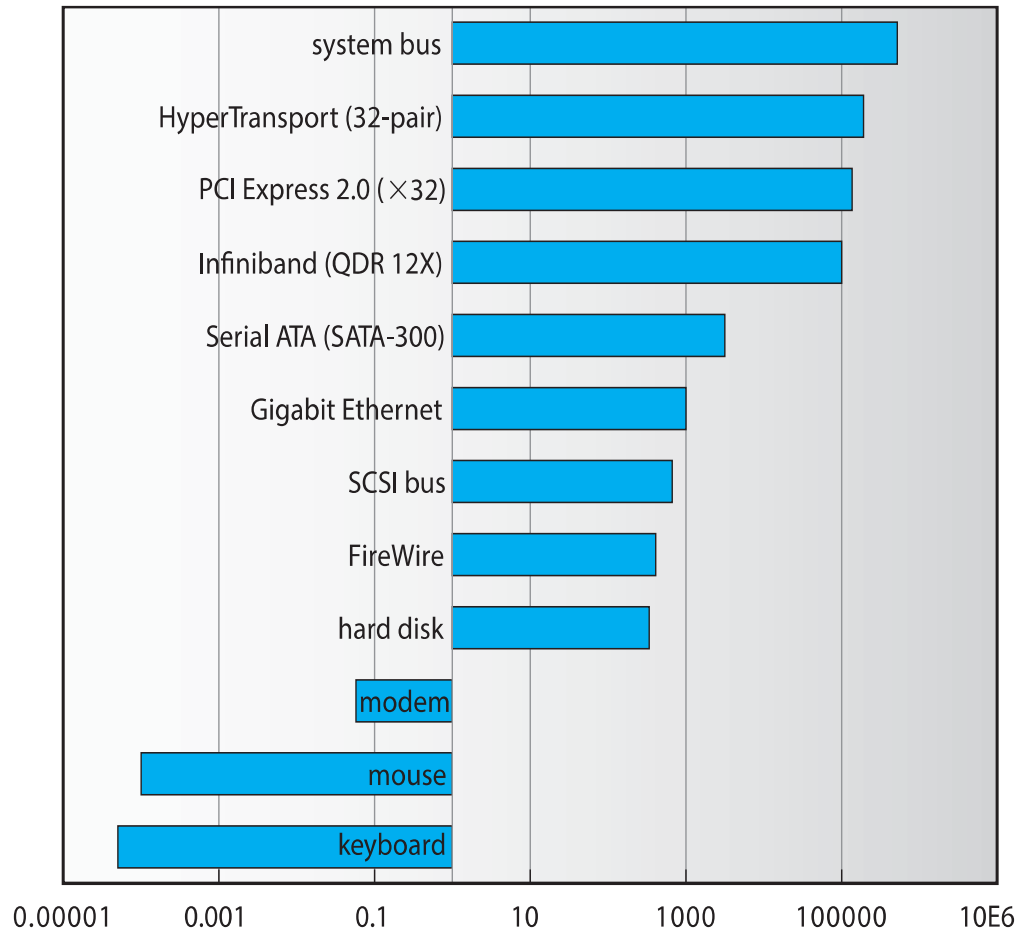
Kernel I/O Subsystem

- Scheduling
 - Some I/O request ordering via per-device queue
 - Some OSs try fairness
 - Some implement Quality Of Service (i.e. IPQOS)
- **Buffering** - store data in memory while transferring between devices
 - To cope with device speed mismatch
 - To cope with device transfer size mismatch
 - To maintain “copy semantics”
 - **Double buffering** – two copies of the data
 - Kernel and user
 - Varying sizes
 - Full / being processed and not-full / being used
 - Copy-on-write can be used for efficiency in some cases

Device-status Table



Sun Enterprise 6000 Device-Transfer Rates



Kernel I/O Subsystem

- **Caching** - faster device holding copy of data
 - Always just a copy
 - Key to performance
 - Sometimes combined with buffering
- **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 de-allocation
 - Watch out for deadlock

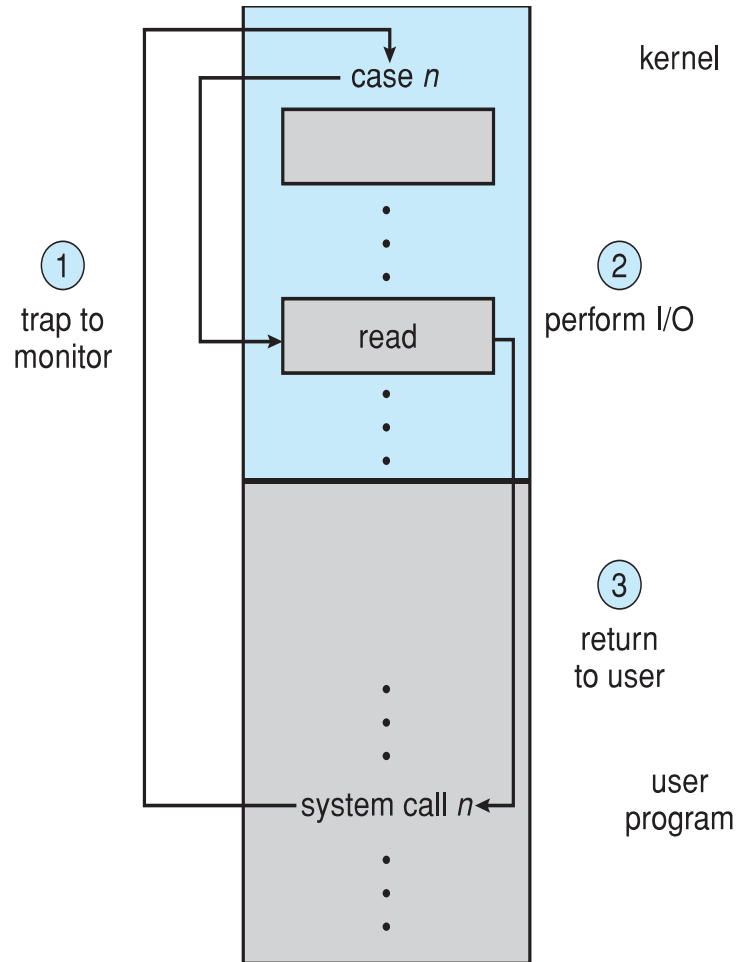
Error Handling

- OS can recover from disk read, device unavailable, transient write failures
 - Retry a read or write, for example
 - Some systems more advanced – Solaris FMA, AIX
 - Track error frequencies, stop using device with increasing frequency of retry-able errors
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

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

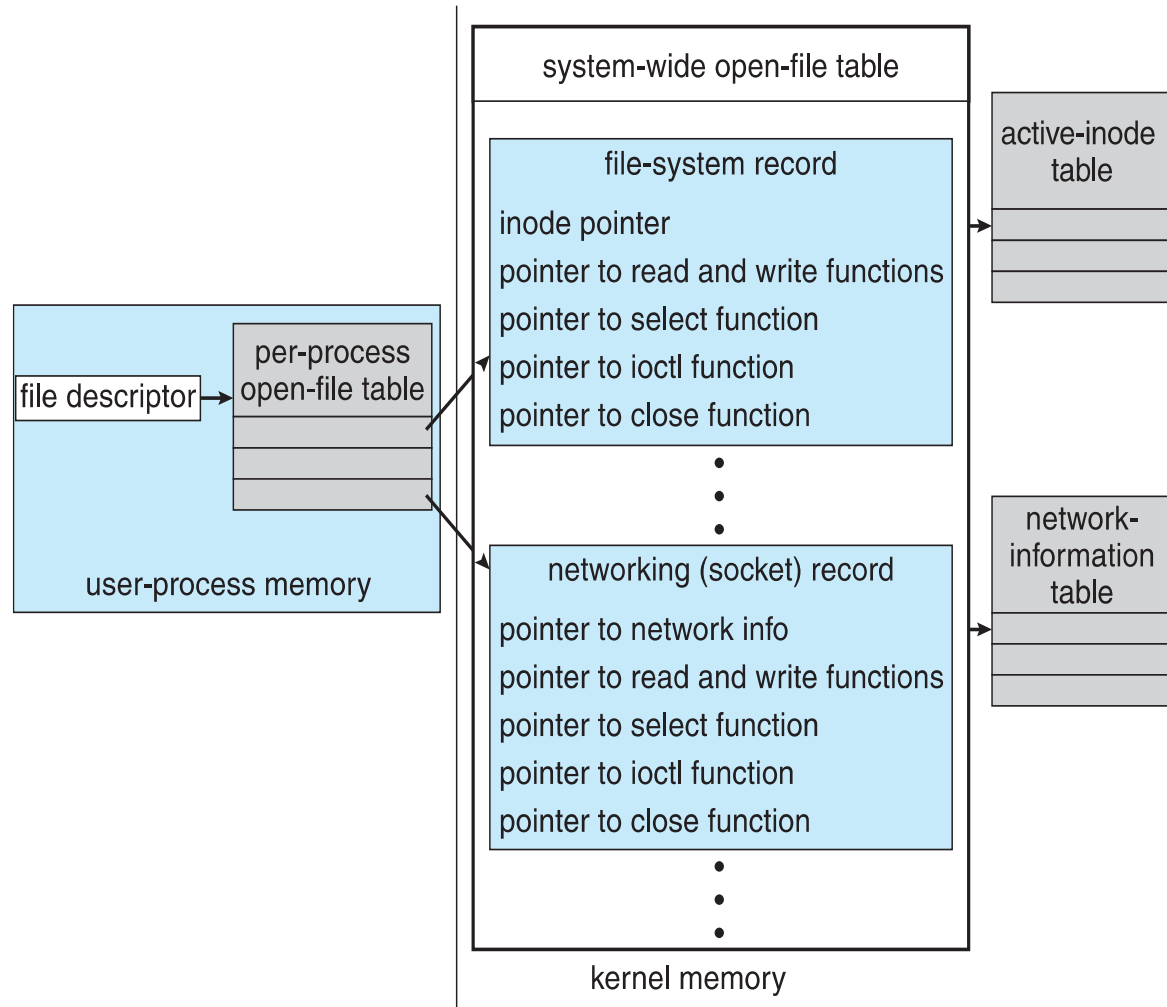
Use of a System Call to Perform I/O



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
 - Windows uses message passing
 - Message with I/O information passed from user mode into kernel
 - Message modified as it flows through to device driver and back to process
 - Pros / cons?

UNIX I/O Kernel Structure



Power Management

- Not strictly domain of I/O, but much is I/O related
- Computers and devices use electricity, generate heat, frequently require cooling
- OSes can help manage and improve use
 - Cloud computing environments move virtual machines between servers
 - Can end up evacuating whole systems and shutting them down
- Mobile computing has power management as first class OS aspect

Power Management (Cont.)

- For example, Android implements
 - Component-level power management
 - Understands relationship between components
 - Build device tree representing physical device topology
 - System bus -> I/O subsystem -> {flash, USB storage}
 - Device driver tracks state of device, whether in use
 - Unused component – turn it off
 - All devices in tree branch unused – turn off branch
 - Wake locks – like other locks but prevent sleep of device when lock is held
 - Power collapse – put a device into very deep sleep
 - Marginal power use
 - Only awake enough to respond to external stimuli (button press, incoming call)

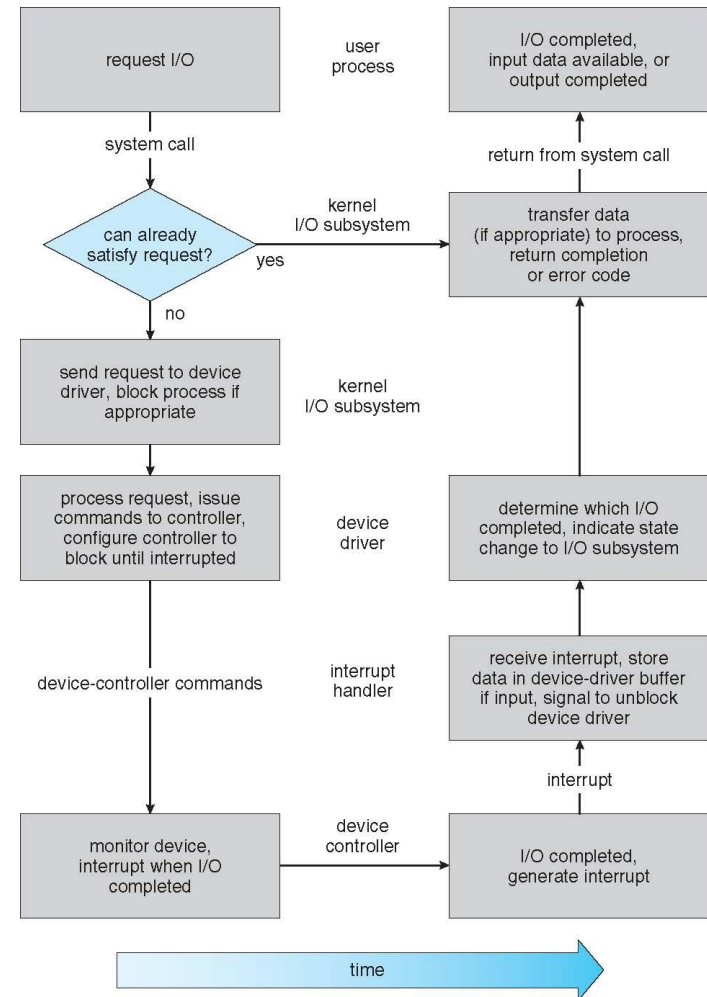
I/O Requests to Hardware Operations

- Consider reading a file from disk for a process:
 - Determine device holding file
 - Translate name to device representation
 - Physically read data from disk into buffer
 - Make data available to requesting process
 - Return control to process

Life Cycle of An I/O Request

- An I/O operation requires a great many steps that together consume a tremendous number of CPU cycles:

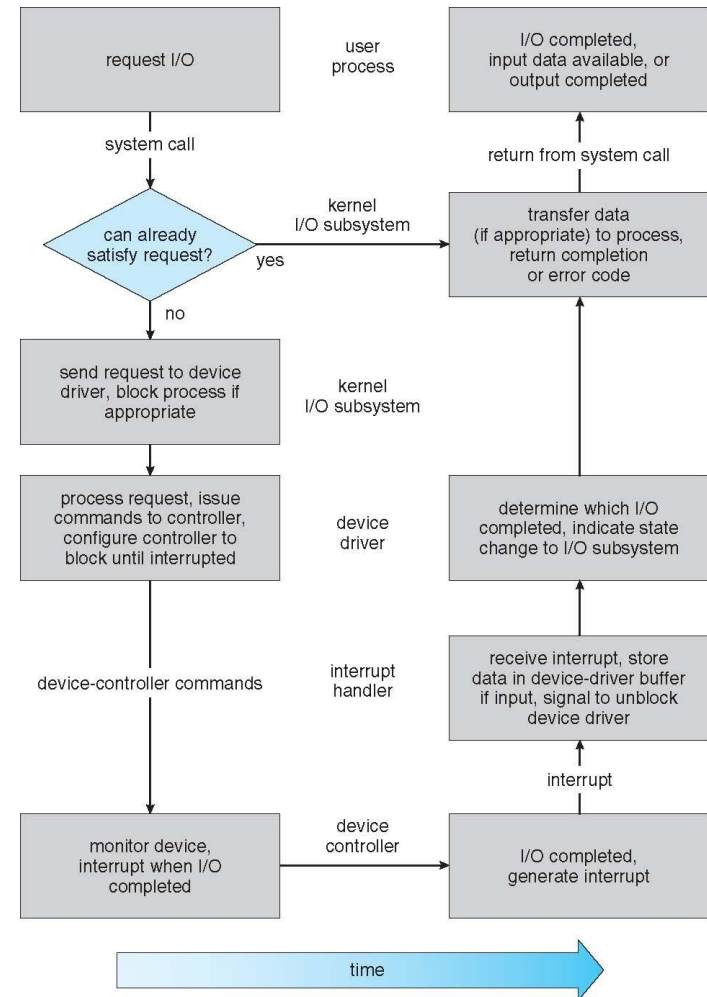
1. A process issues a blocking read() system call to a file descriptor of a file that has been opened previously.
2. The system call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is complete.



Life Cycle of An I/O Request

- An I/O operation requires a great many steps that together consume a tremendous number of CPU cycles:

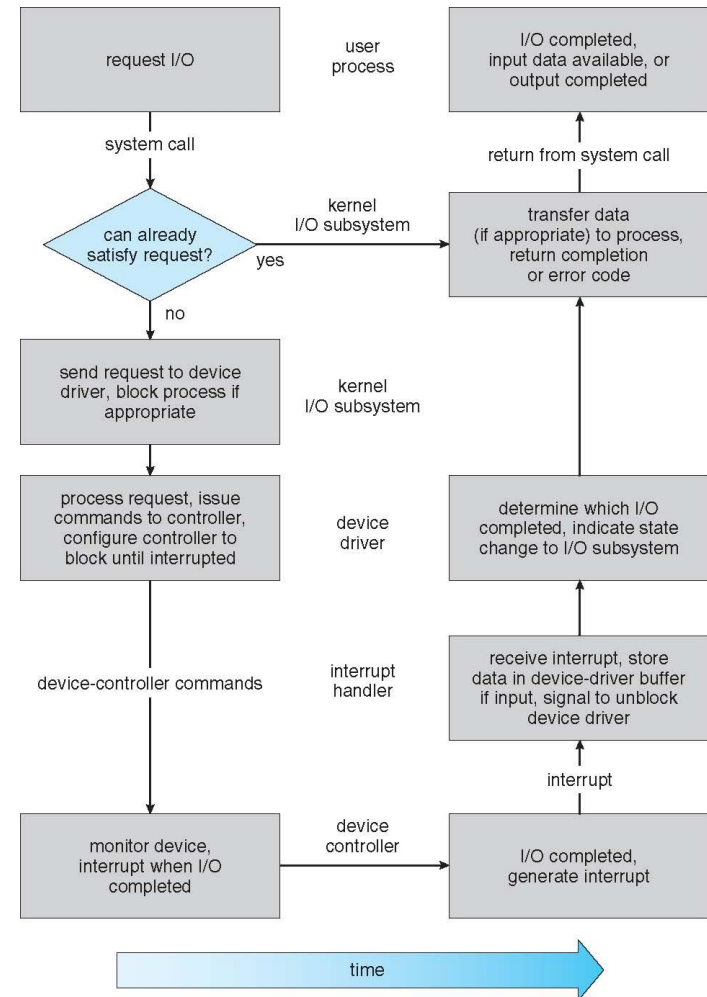
3. otherwise, a physical I/O must be performed. The process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.



Life Cycle of An I/O Request

- An I/O operation requires a great many steps that together consume a tremendous number of CPU cycles:

- The device driver allocates kernel buffer space to receive the data and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device-control registers.
- The device controller operates the device hardware to perform the data transfer.

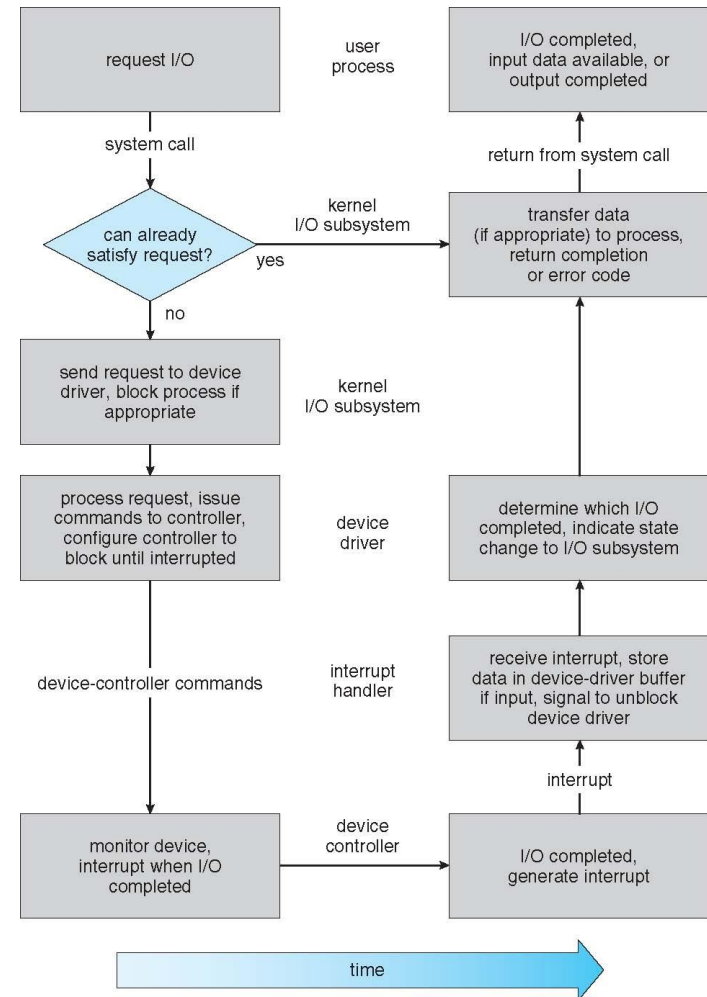


Life Cycle of An I/O Request

- An I/O operation requires a great many steps that together consume a tremendous number of CPU cycles:

5. The device controller operates the device hardware to perform the data transfer.

6. The driver may pool for status and data, or it may have set up to a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.

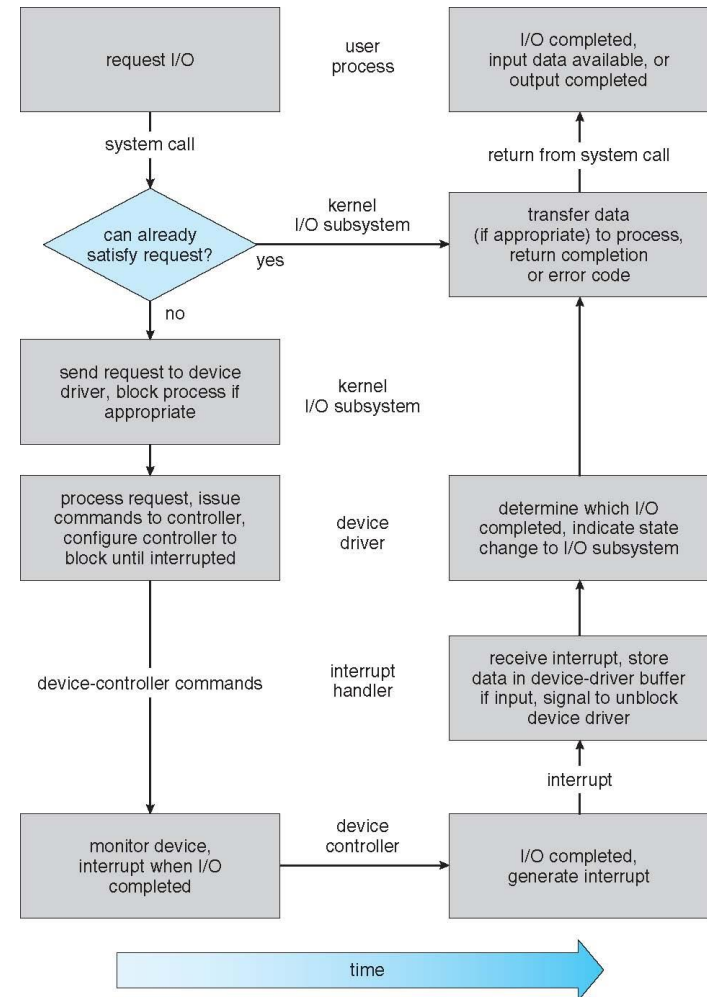


Life Cycle of An I/O Request

- An I/O operation requires a great many steps that together consume a tremendous number of CPU cycles:

7. The correct interrupt handler receives the interrupt via the interrupt-vector table, stores any necessary data, signal the device driver and returns from the interrupt.

8. The device driver receives the signal, determines which I/O request has completed, determines the request's status, and signal the kernel I/O subsystem that the request has been completed.

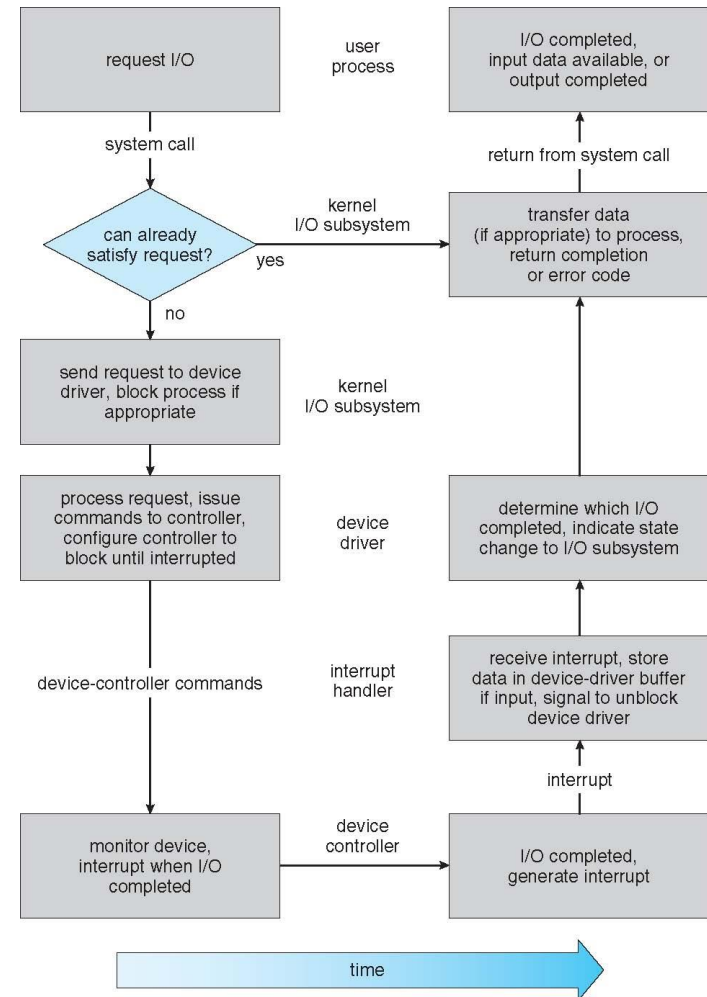


Life Cycle of An I/O Request

- An I/O operation requires a great many steps that together consume a tremendous number of CPU cycles:

9. 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.

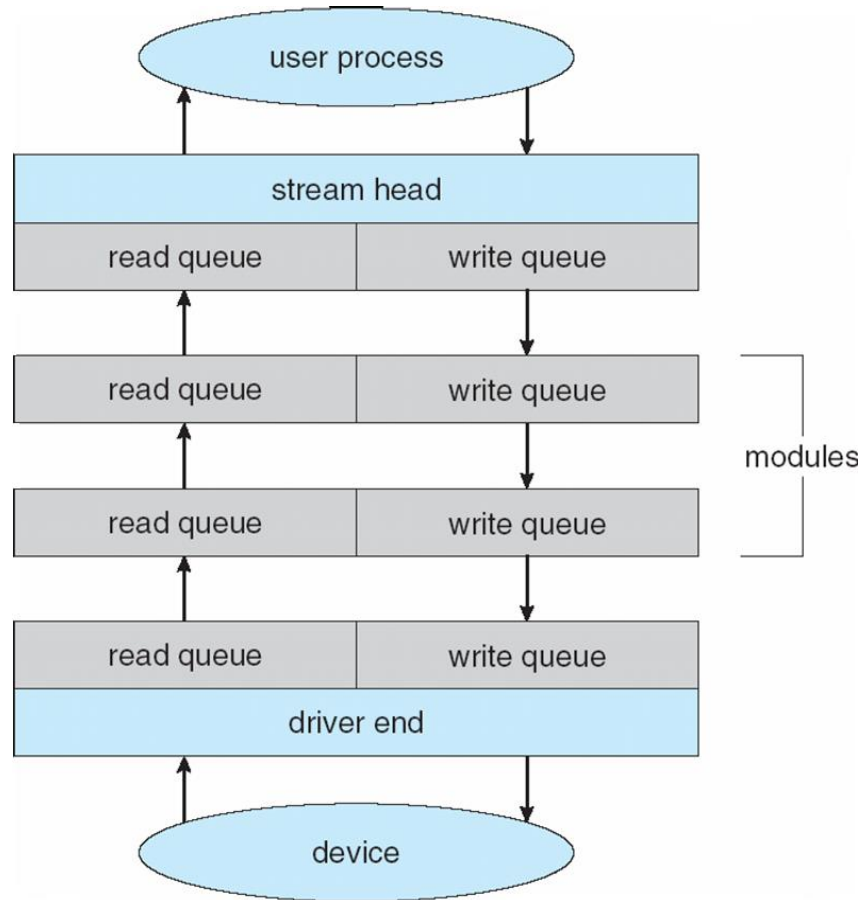
10. Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.



STREAMS

- **STREAM** – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond
- A STREAM consists of:
 - STREAM head interfaces with the user process
 - driver end interfaces with the device
 - zero or more STREAM modules between them
- Each module contains a **read queue** and a **write queue**
- Message passing is used to communicate between queues
 - **Flow control** option to indicate available or busy
- Asynchronous internally, synchronous where user process communicates with stream head

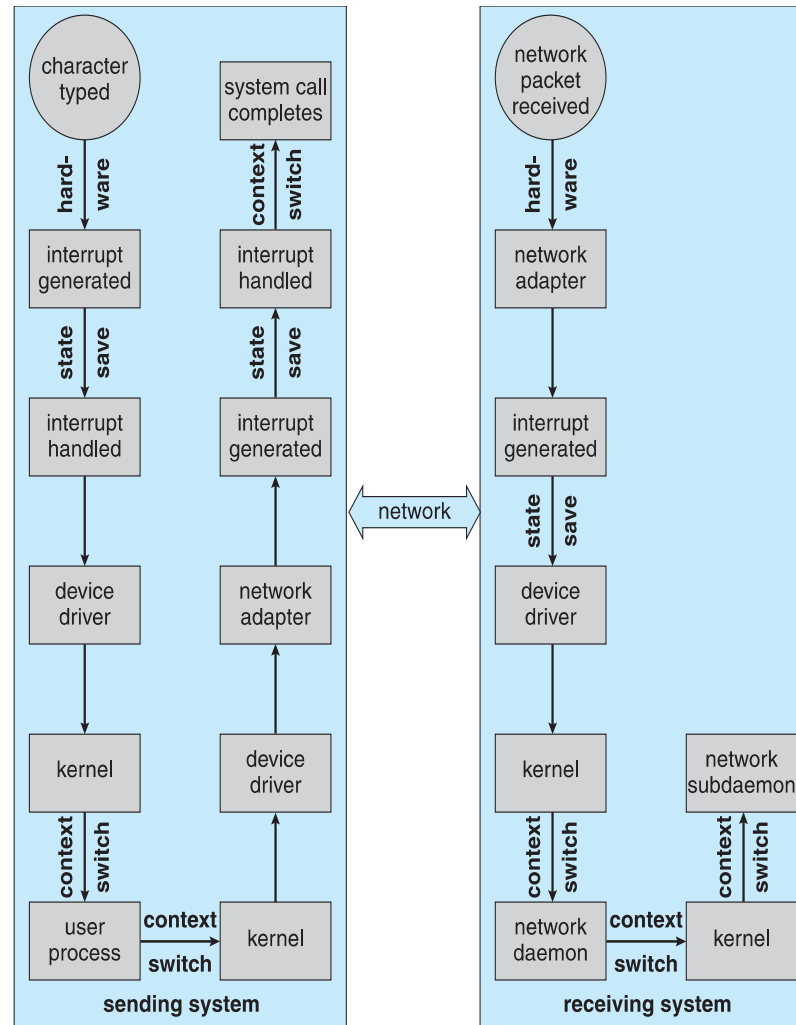
The STREAMS Structure



Performance

- I/O a major factor in system performance:
 - Demands CPU to execute device driver, kernel I/O code
 - Context switches due to interrupts
 - Data copying
 - Network traffic especially stressful

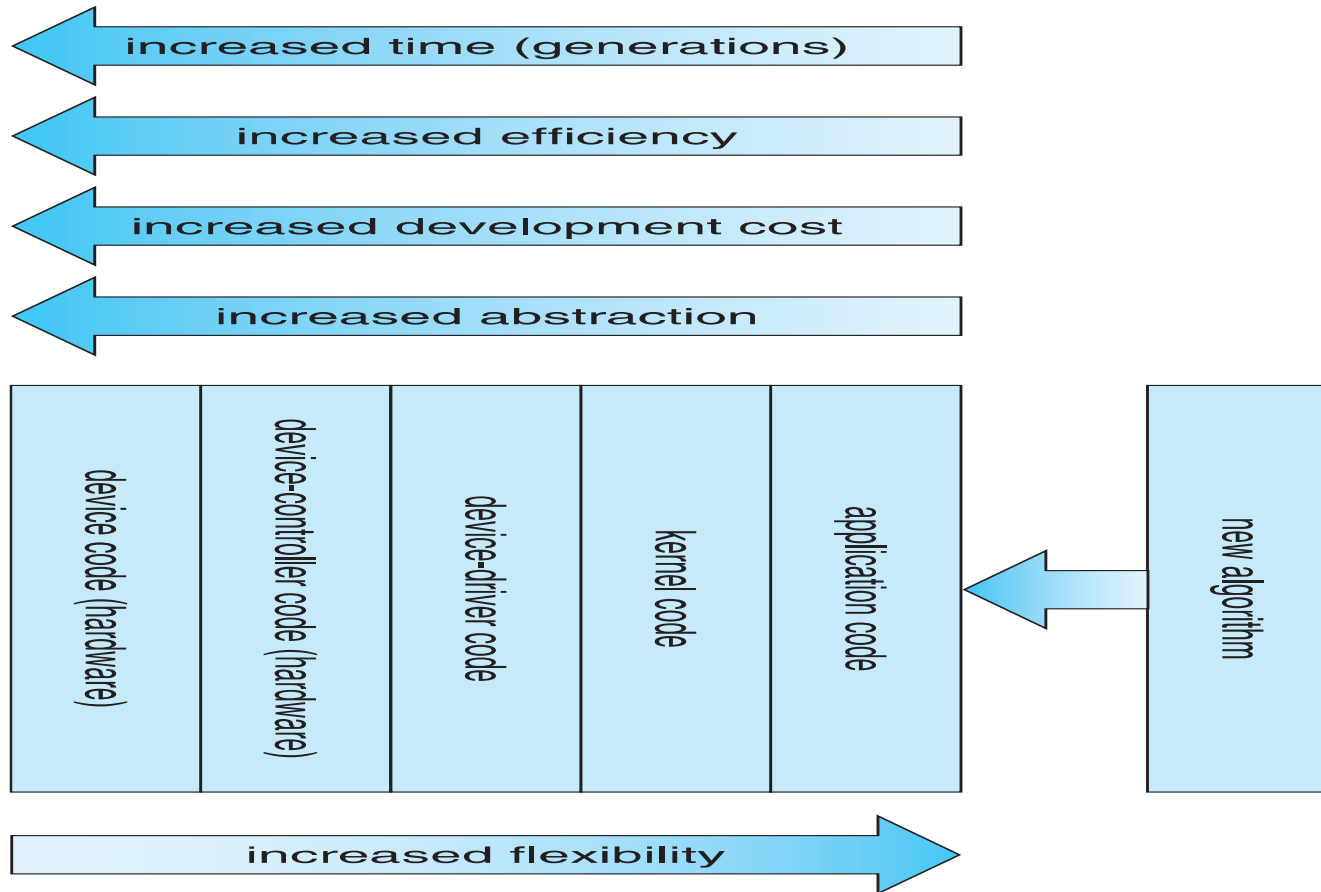
Slides content from Operating Systems 10th Edition — Silberschatz, Galvin and Gagne © 2018



Improving Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput
- Move user-mode processes / daemons to kernel threads

Device-Functionality Progression



Thanks



**Politecnico
di Torino**

Department of Control and
Computer Engineering



Questions?

sarah.azimi@polito.it