I/O Management

Gaurav Somani

Reference – [Galvin]

I/O Systems

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

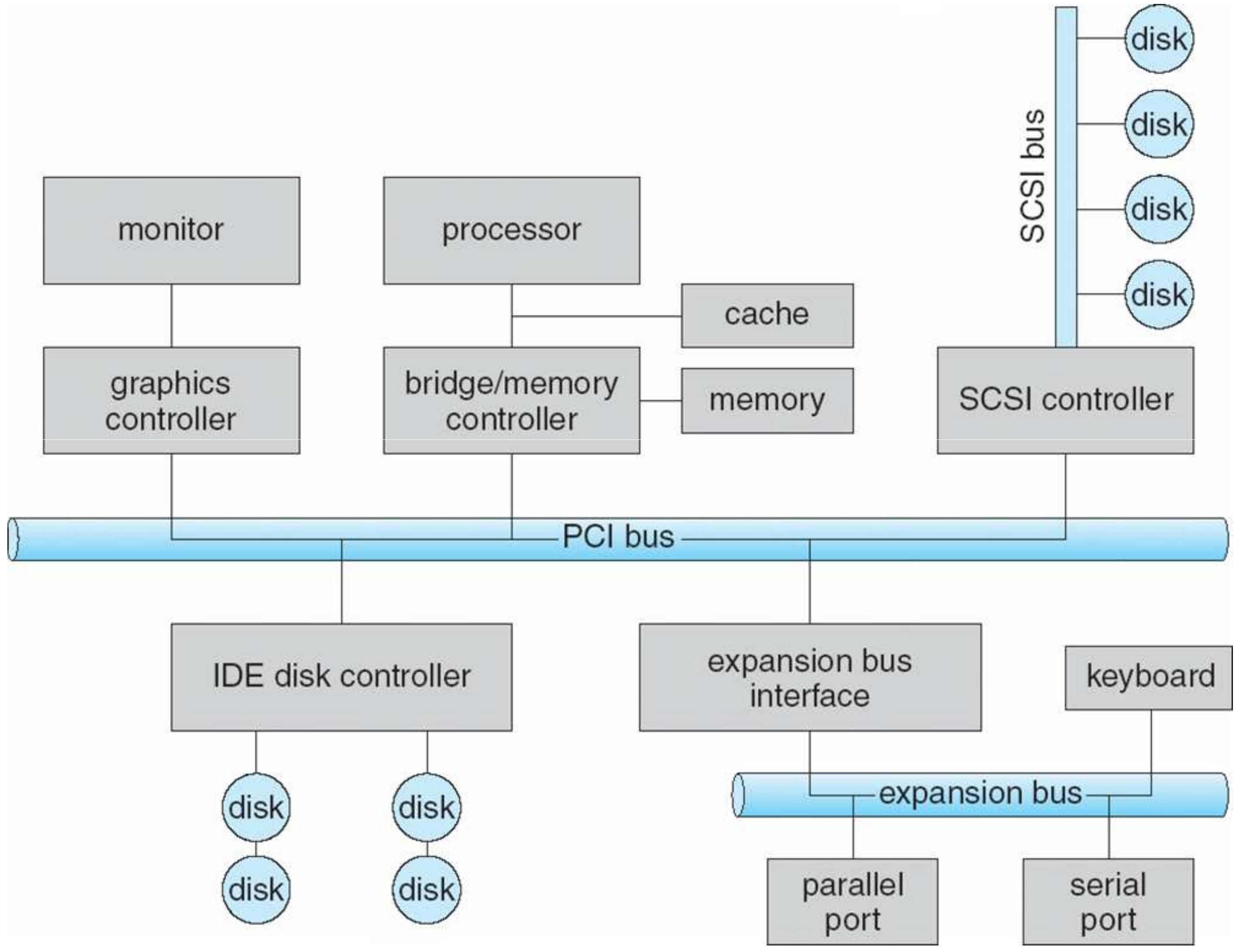
Objectives

- 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

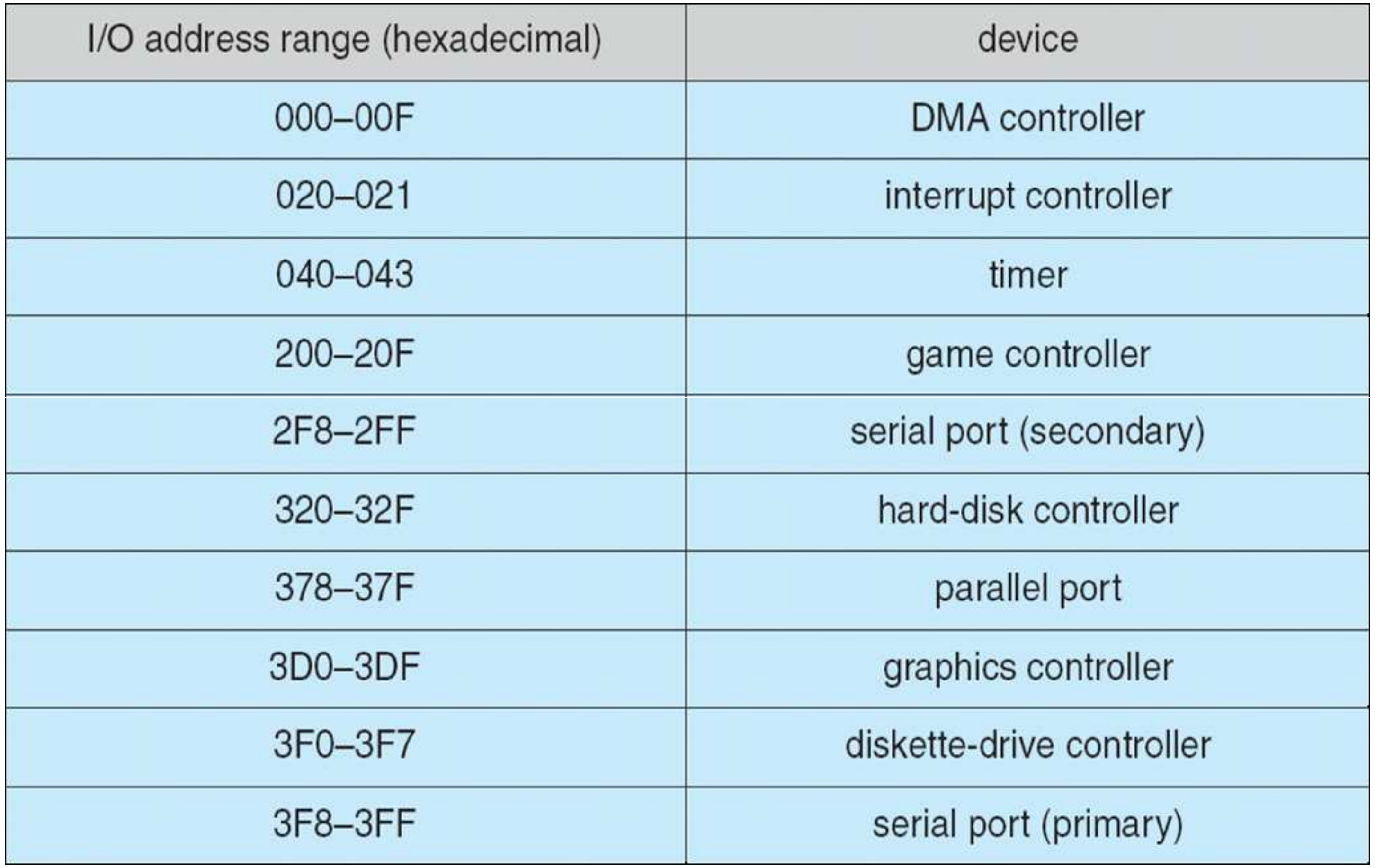
I/O Hardware

- Incredible variety of I/O devices
- Common concepts
  - Port
  - Bus (daisy chain or shared direct access)
  - Controller (host adapter)
- I/O instructions control devices
- Devices have addresses, used by
  - Direct I/O instructions
  - Memory-mapped I/O

A Typical PC Bus Structure



Device I/O Port Locations on PCs (partial)



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)

I/O ports

An I/O port typically consists of four registers, called the

• The **data-in register is read by the host to get input.**

• The **data-out register is written by the host to send output.**

• The **status register contains bits that can be read by the host. These bits** indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.

• The control register can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex communication, another bit enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.

Polling

Basic handshaking notion.

1. The host repeatedly reads the *busy bit until that bit becomes clear.*
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 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. The controller clears the *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.

Overflow problem

Interrupts

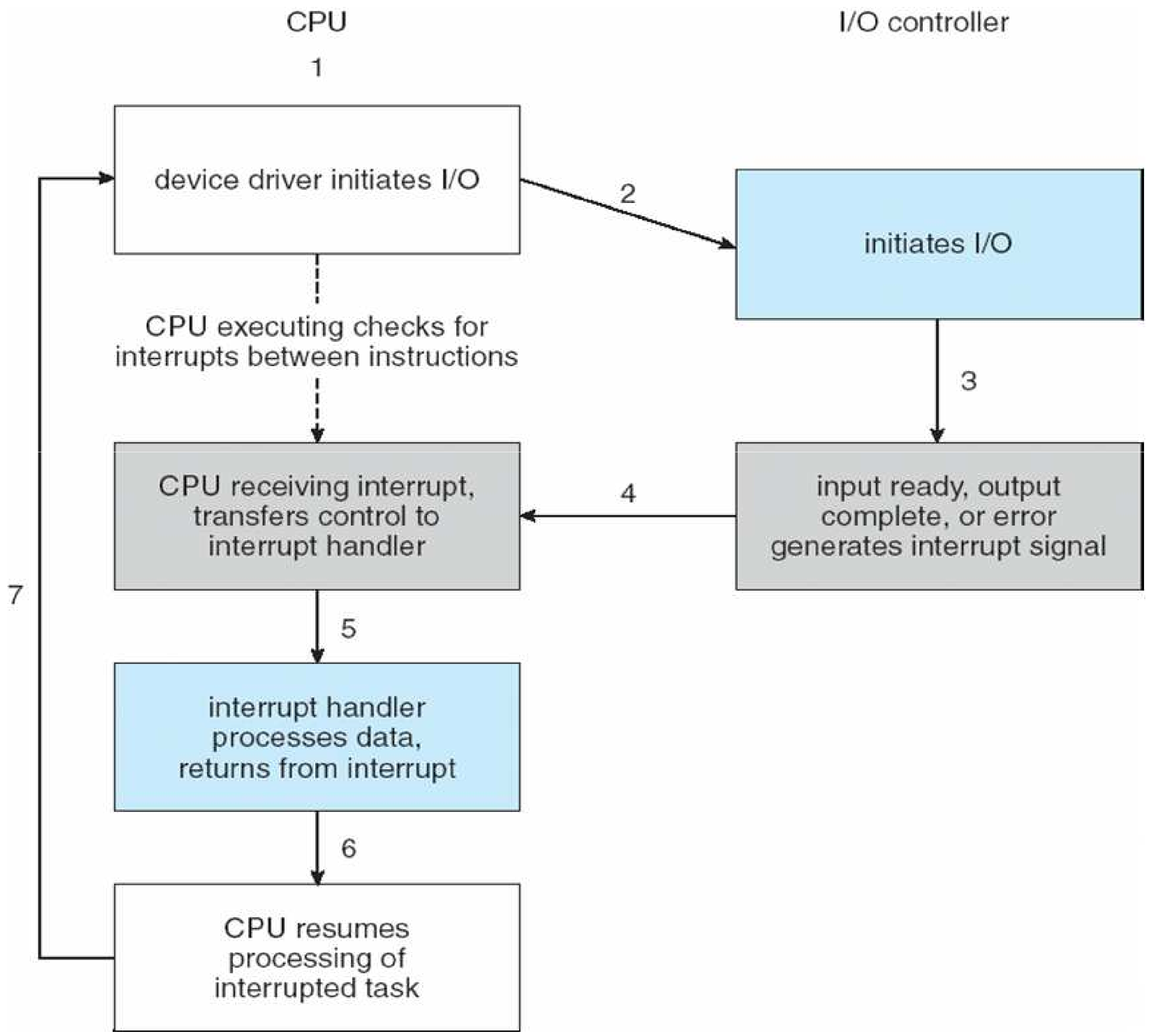
- CPU Interrupt-request line triggered by I/O device
- Interrupt handler receives interrupts
- Maskable to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
  - Based on priority
  - Some nonmaskable
- Interrupt mechanism also used for exceptions

Advanced Features

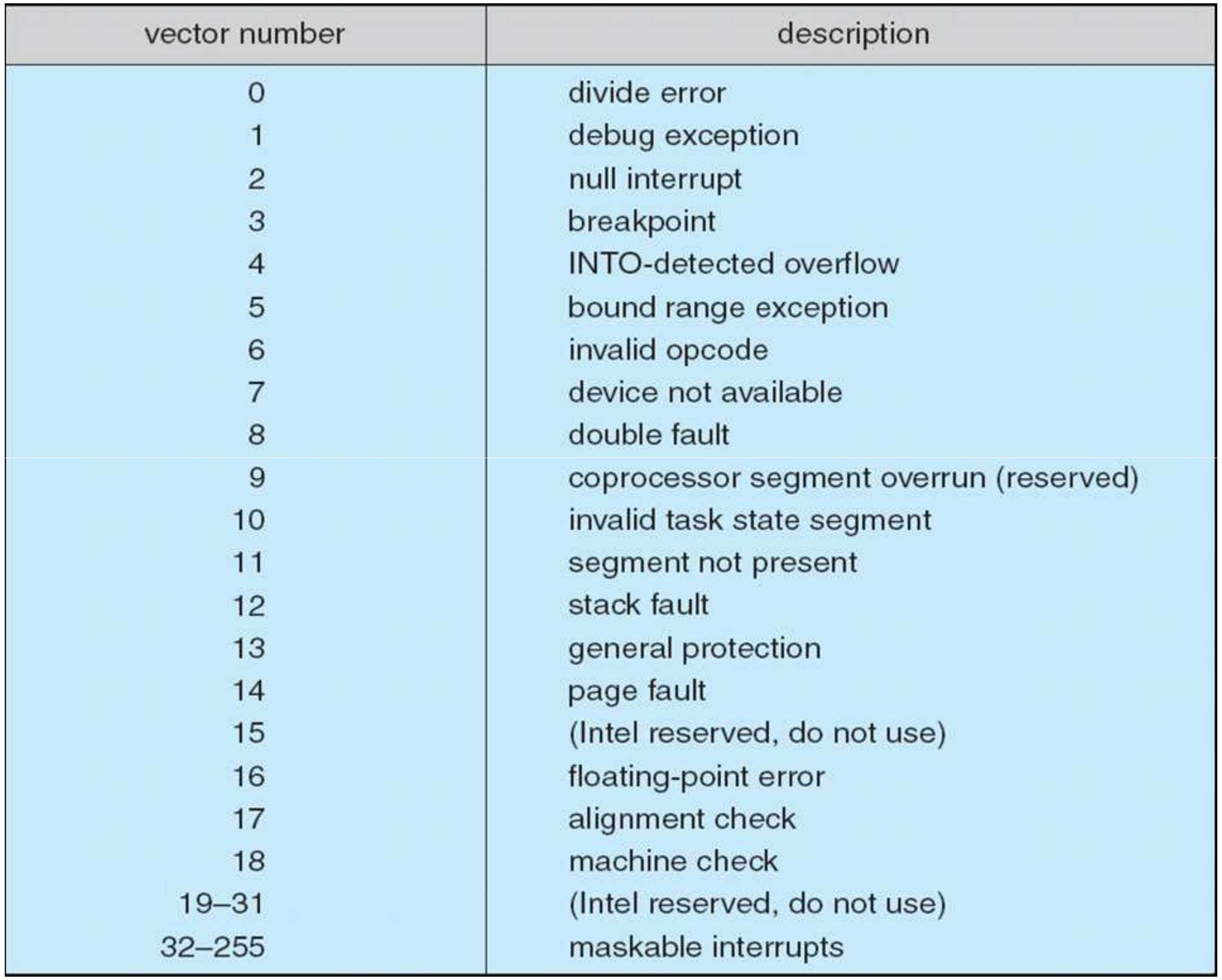
We need more sophisticated, interrupt handling features.

- The ability to defer interrupt handling during critical processing.
- An efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt.
- Multilevel interrupts, so that the operating system can distinguish between high and low-priority interrupts and can respond with the appropriate degree of urgency.

Interrupt-Driven I/O Cycle


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Intel Pentium Processor Event-Vector Table

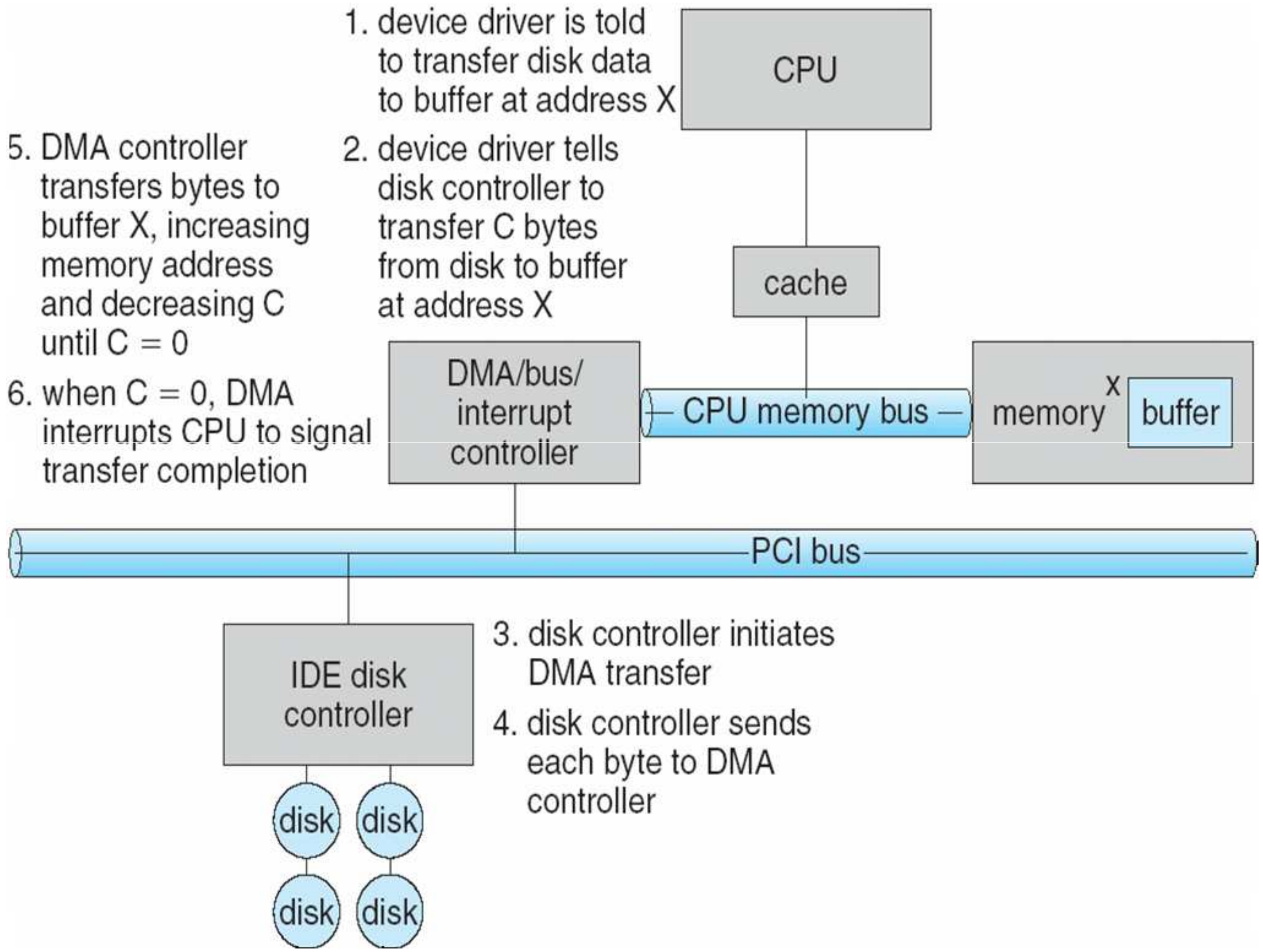


vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
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

Direct Memory Access

- Used to avoid programmed I/O for large data movement
- Requires DMA controller
- Bypasses CPU to transfer data directly between I/O device and memory

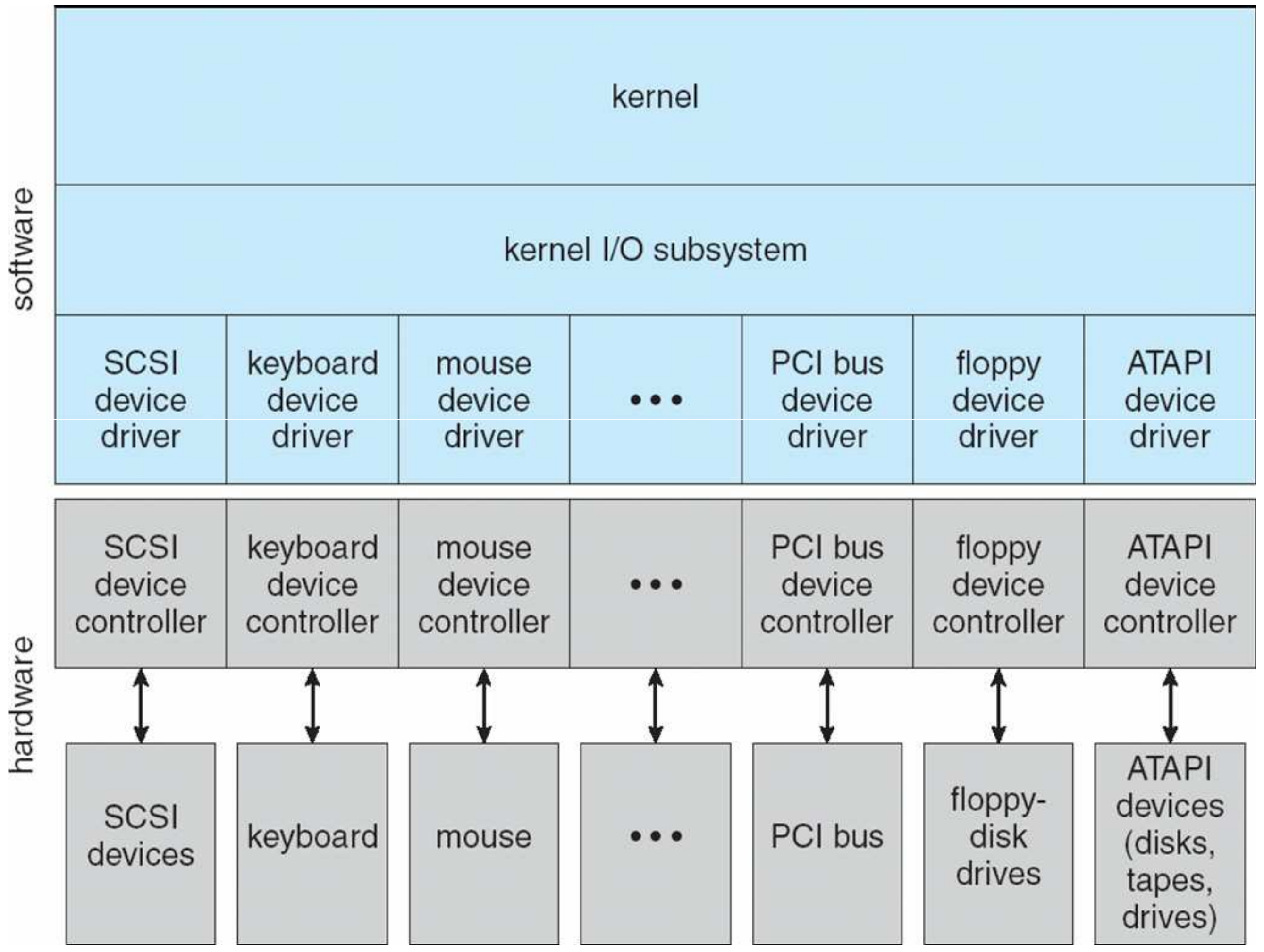
Six Step Process to Perform DMA Transfer


1. device driver is told to transfer disk data to buffer at address X
CPU
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
cache
6. when C = 0, DMA interrupts CPU to signal transfer completion
DMA/bus/ interrupt controller
CPU memory bus
memory
X
buffer
PCI bus
IDE disk controller
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
disk
disk
disk
disk

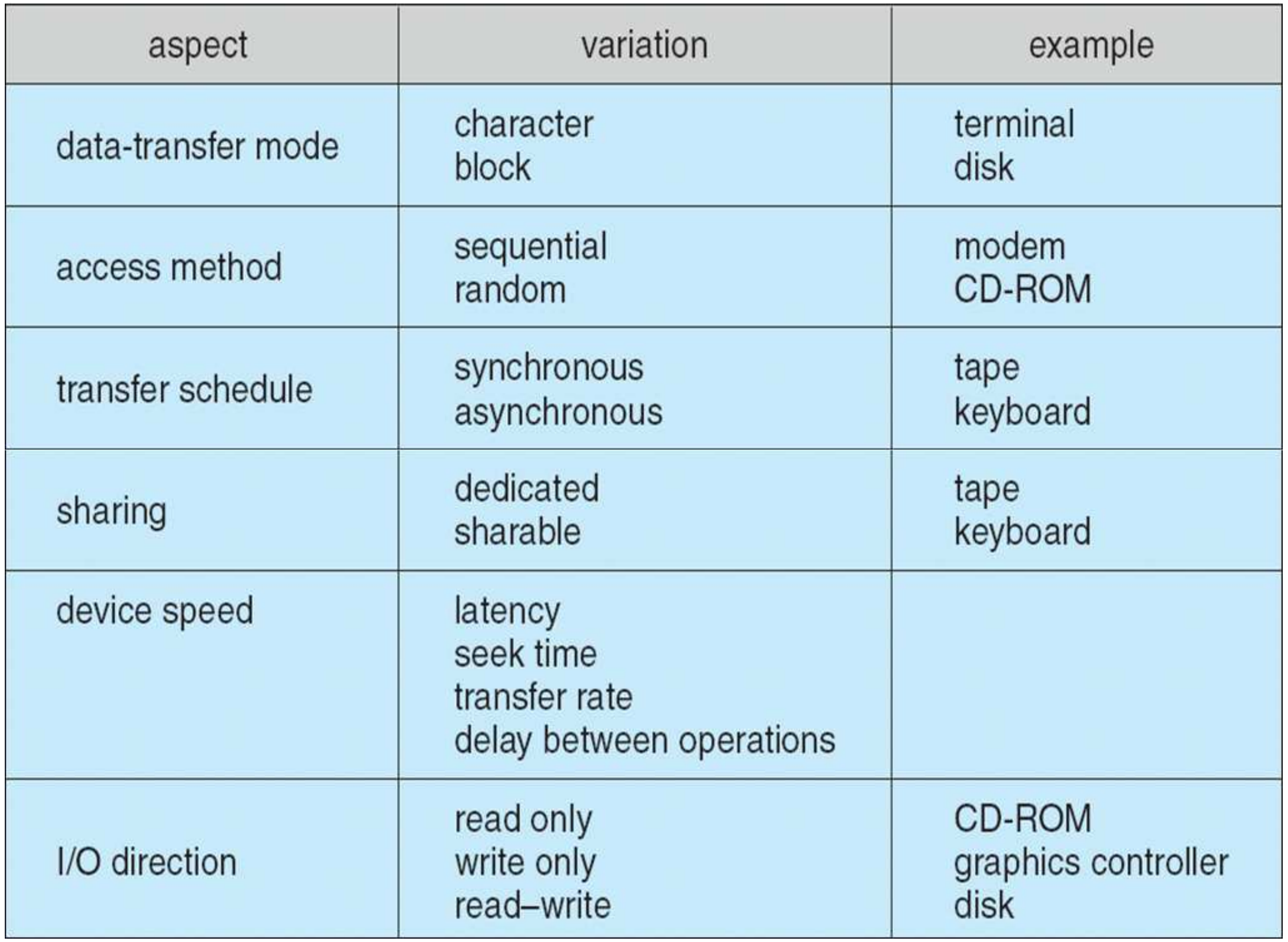
Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- 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

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

Block and Character Devices

- Block devices include disk drives
  - Commands include read, write, seek
  - Raw I/O or file-system access (e.g. Database)
  - Memory-mapped file access possible

- 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

- Unix and Windows NT/9*x*/2000 include socket() interface
  - Separates network protocol from network operation
  - Includes select() functionality – No need of polling/busy wait

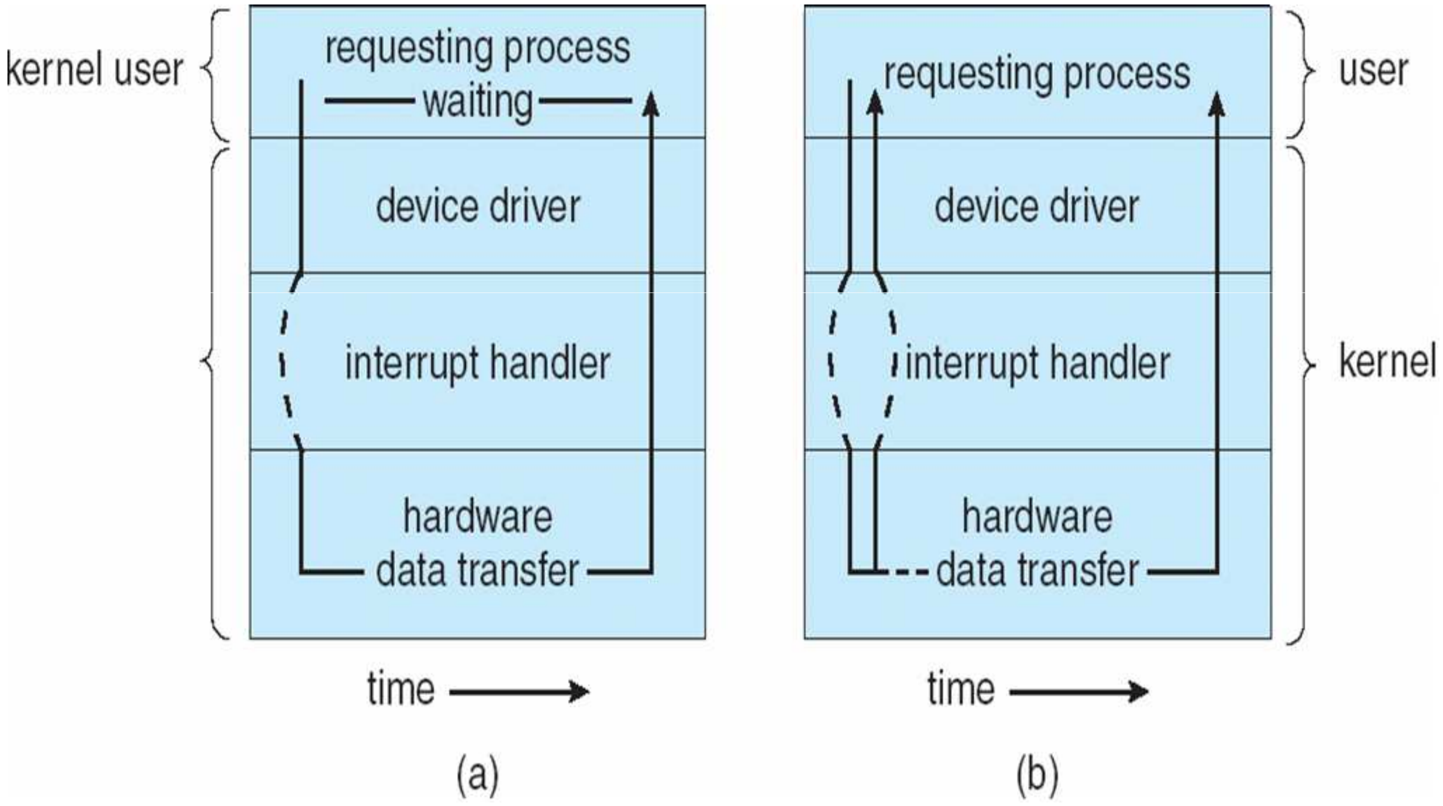
Clocks and Timers

- Provide current time, elapsed time, timer
- Programmable interval timer used for timings, periodic interrupts
- ioctl() (on UNIX) covers all aspects of I/O)

Blocking and Nonblocking 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

Two I/O Methods



Kernel I/O Subsystem

- Scheduling
  - Some I/O request ordering via per-device queue (Device status table)
  - Some OSs try fairness
- 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"

I/O Scheduler

deadline scheduler: this is a cyclic elevator but with a twist: requests are given a deadline by which they get served. When a request starts to look like it's going to expire, the kernel will skip intermediate sectors and move to that request, thus giving some realtime behaviour.

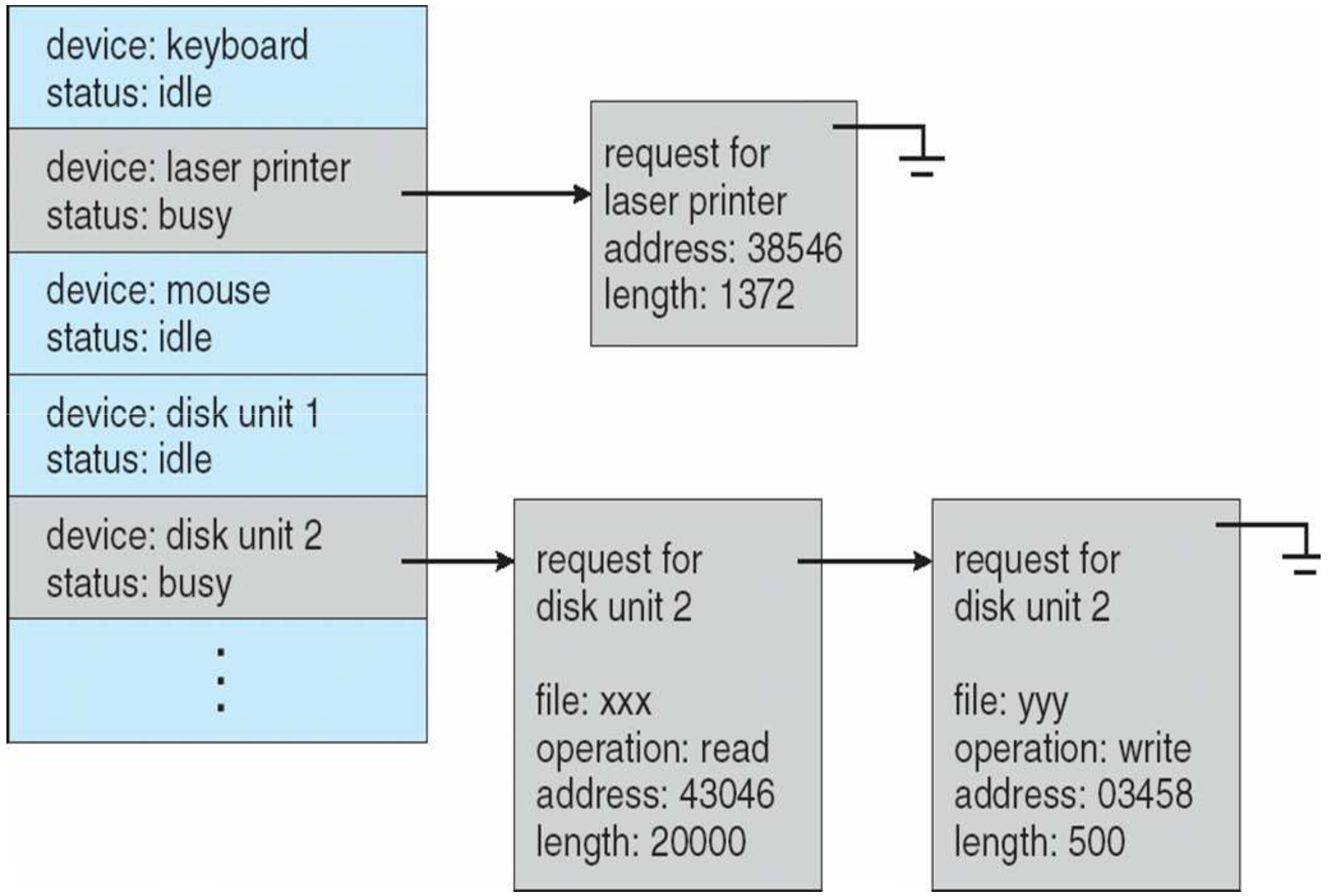
AS scheduler: a cyclic elevator with waiting policy: after you service a request that looks like it might have future requests coming nearby, you pause even if there's more sectors in your work queue. The anticipatory scheduler literally anticipates more requests to follow on this track or very close by. How AS decides whether to anticipate is basically just lot of guesswork based on typical access patterns.

cfq scheduler: different sort of stab at fairness. It's different from either of these two, and doesn't use cyclic elevator and has realtime guarantees and aggressively avoids starvation. It could be a good scheduler for multiuser systems.

noop scheduler: just service next request in the queue without any algorithm to prefer this or that request.

echo noop > /sys/block/sda/queue/scheduler

Device-status Table



Kernel I/O Subsystem

- Caching - fast memory holding copy of data
  - Always just a copy
  - Key to performance

- Spooling - hold output for a device
  - If device can serve only one request at a time
  - i.e., Printing

- Device reservation - provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock

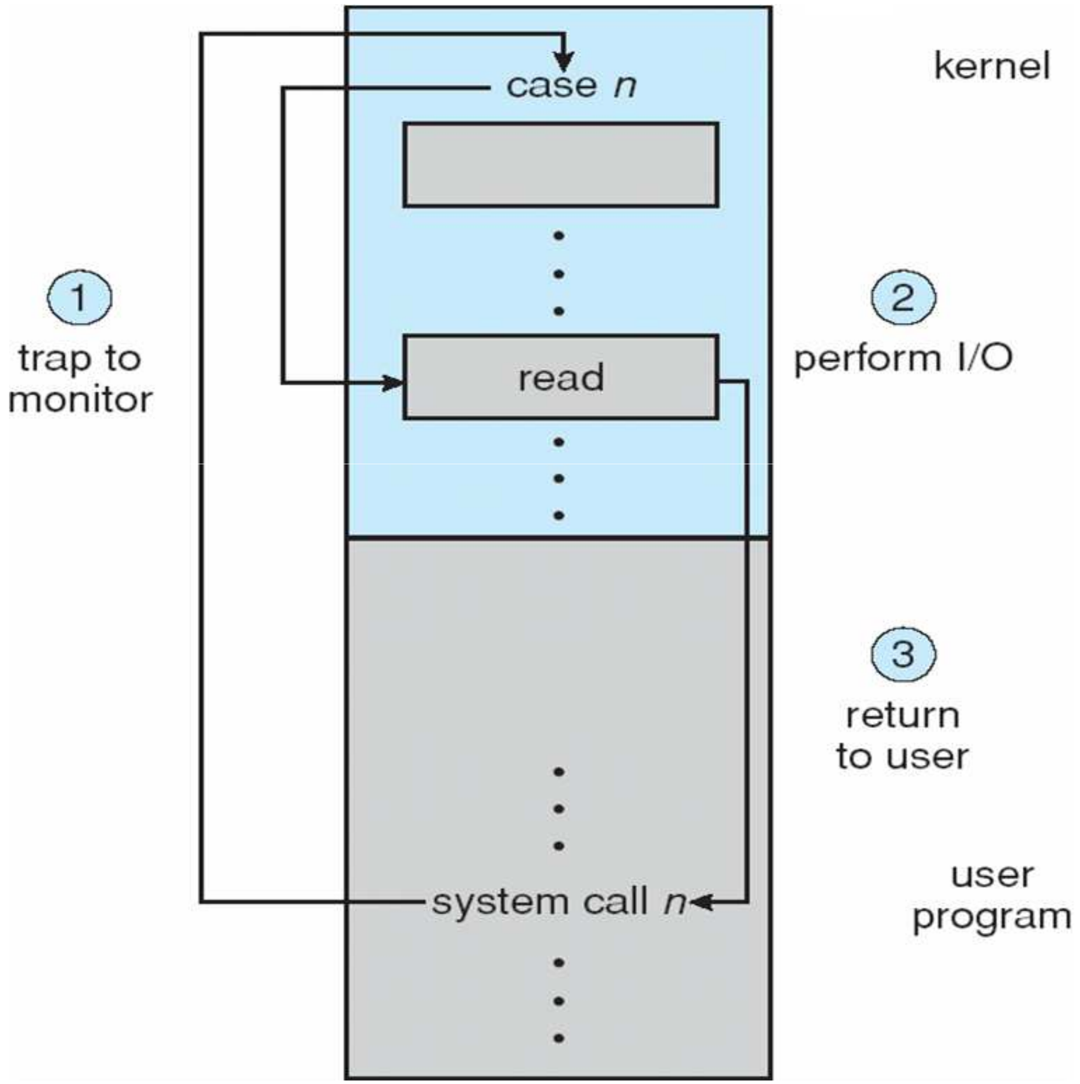
Error Handling

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

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

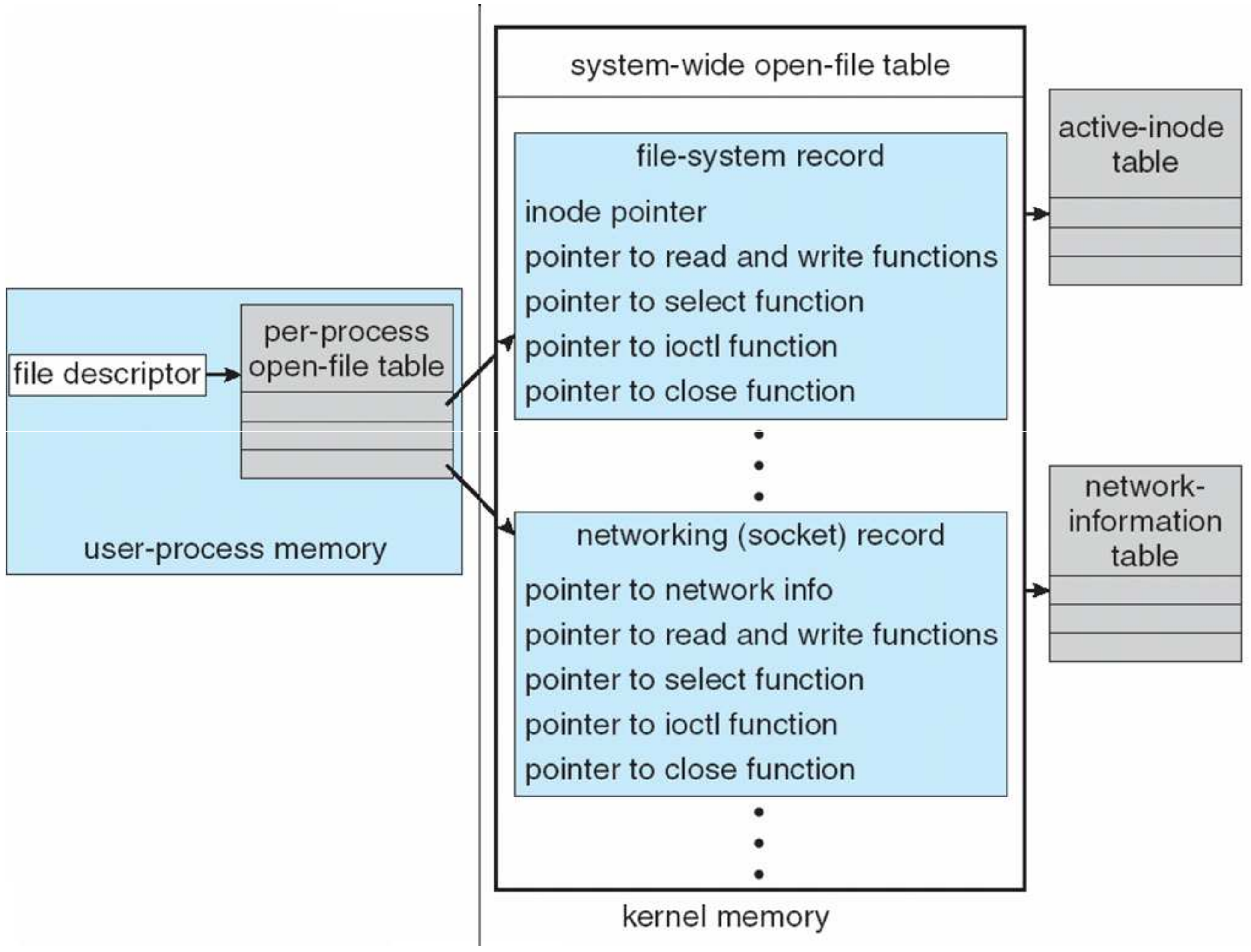
Use of a System Call to Perform I/O



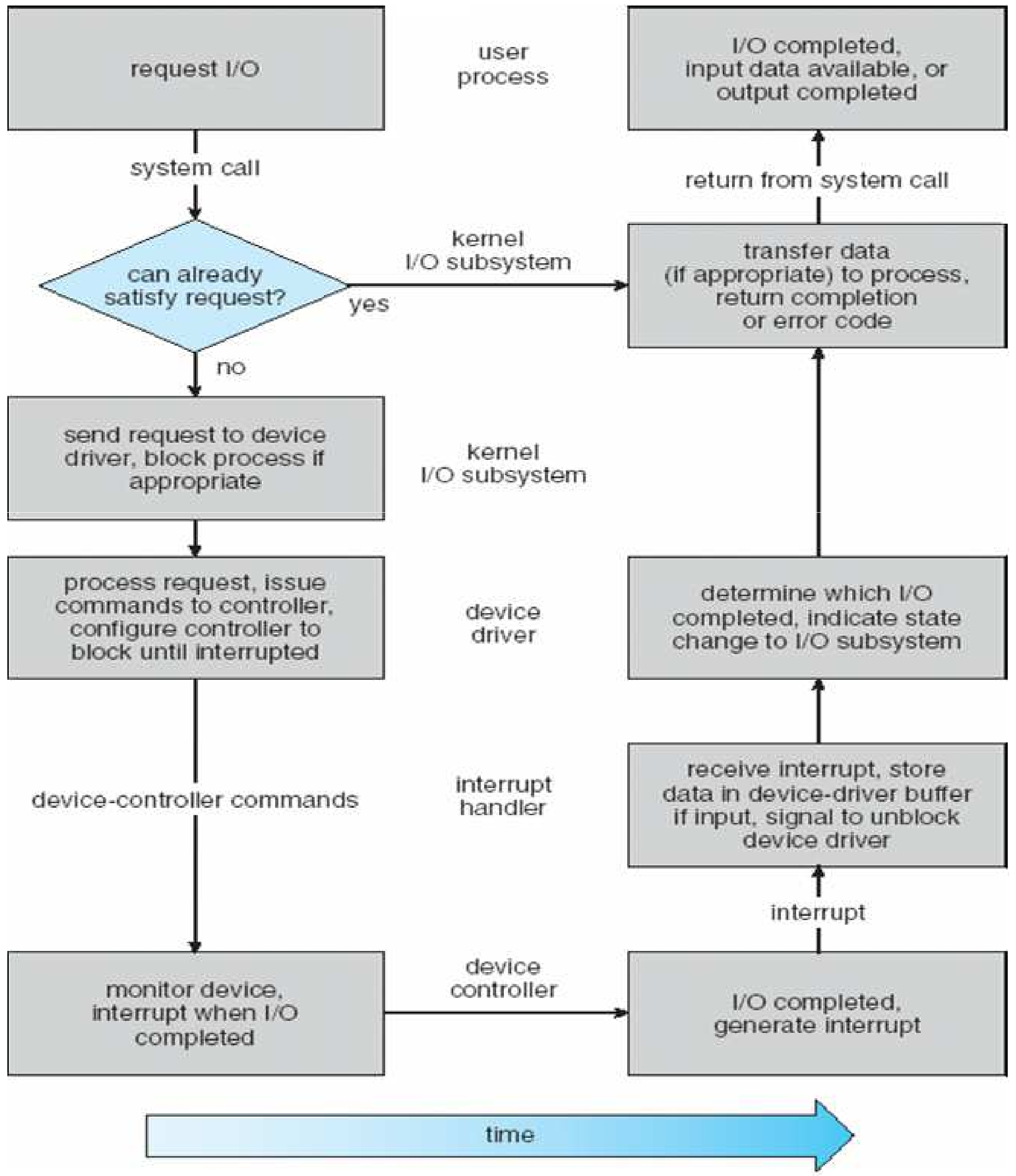
Kernel Data Structures

- 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

UNIX I/O Kernel Structure


system-wide open-file table
file-system record
inode pointer
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
active-inode table
file descriptor
per-process open-file table
user-process memory
networking (socket) record
pointer to network info
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
network-information table
kernel memory

Life Cycle of An I/O Request


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
kernel I/O subsystem
yes
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time