Chapter 12: I/O Systems

CS 3423 Operating Systems
Fall 2019
National Tsing Hua University

Outline

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

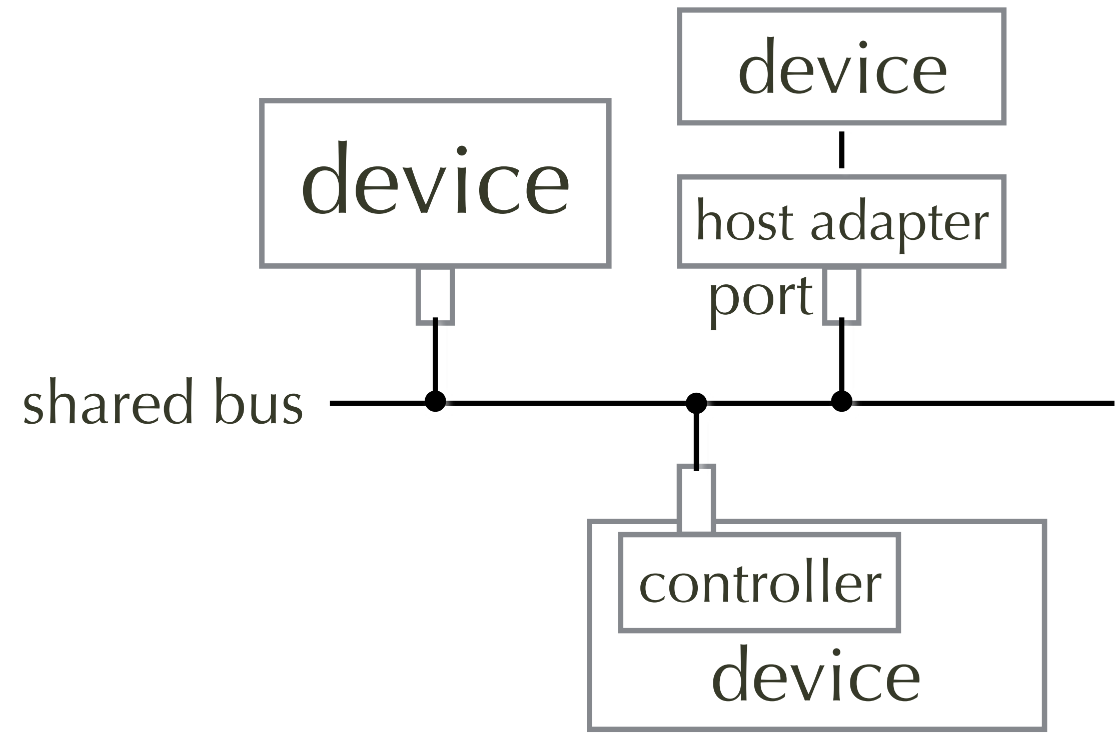
I/O Hardware

I/O Management

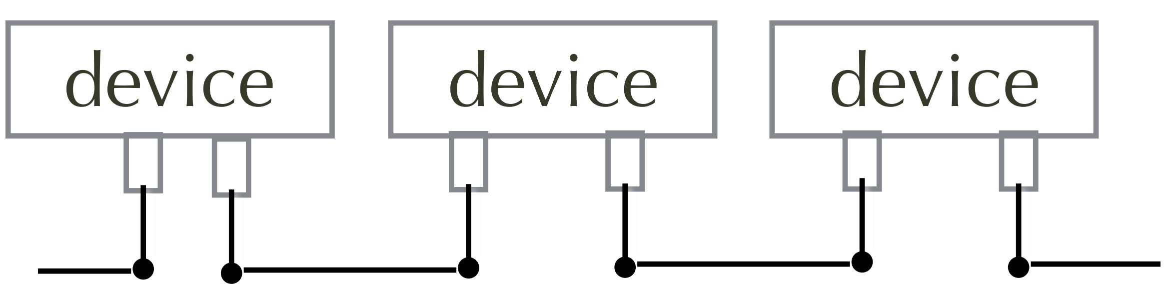
- I/O devices vary greatly
  - Types: Storage, Transmission, Human-interface
  - Connect via ports, busses, device controllers
  - Various methods to control them
  - Performance management
- Device drivers encapsulate device details
  - Present uniform device-access interface to I/O subsystem

I/O Interfaces

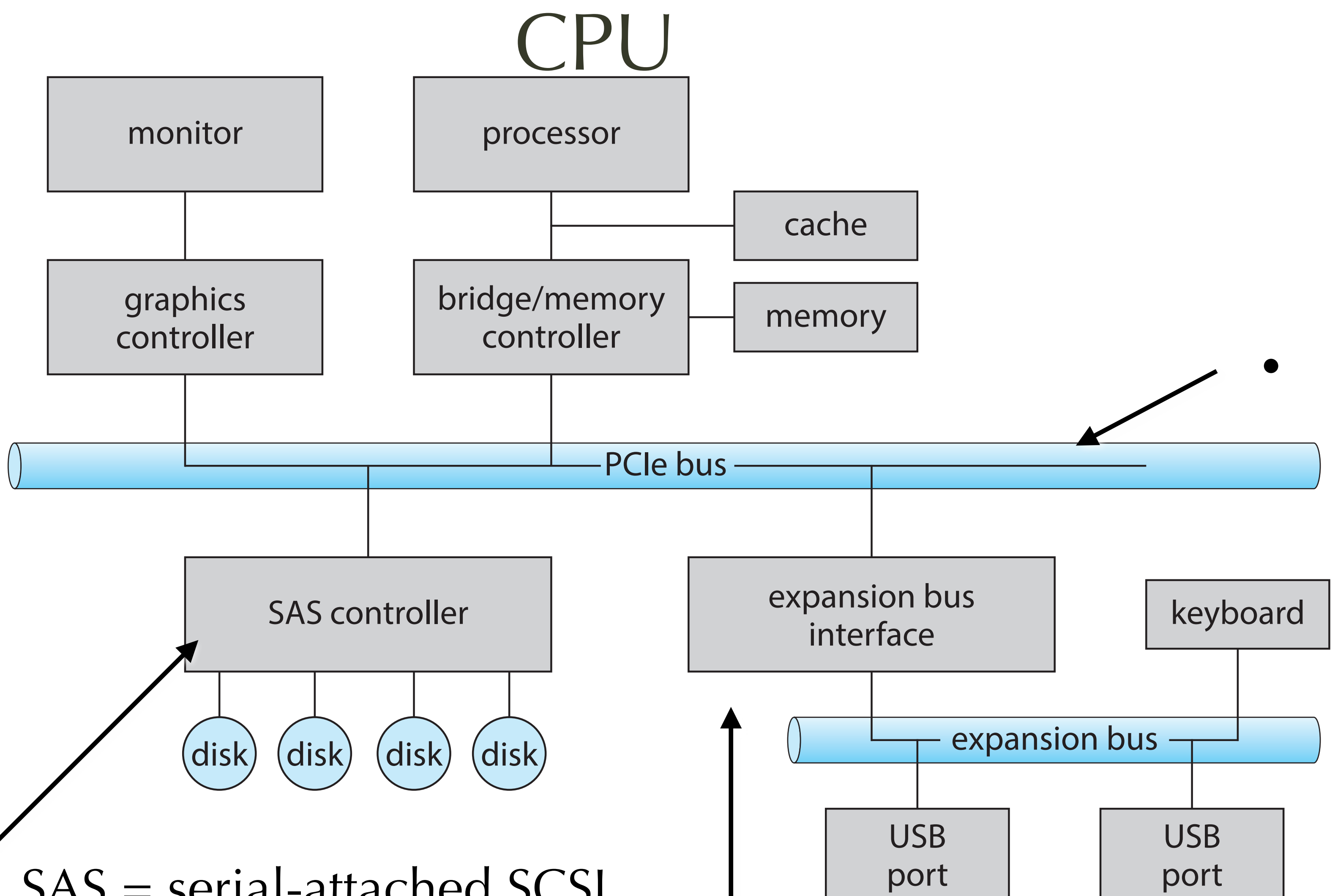
- Port: <u>connection point</u> for device
- Bus: shared (group of) wires for connecting ports
  - daisy chain or shared direct access
- Controller:
  - operates on port, bus, device
  - Sometimes separate circuit board (host bus adapter - HBA)
  - Contains own "processor", microcode, memory, bus controller
  - Some talk to per-device controller with bus controller, microcode, memory, etc



Daisy chain (e.g., SCSI)



A Typical PC Bus Structure



- PCI bus is common in PCs and servers, parallel connection, 5 devices max, not hot pluggable (Intel, 1990's)
- PCIe = PCI Express, serial connection, switched, hot plugging, 32 devices max (Intel, 2001)

- SAS = serial-attached SCSI example of host-bus adapter (HBA) or just *host adapter*

expansion bus connects relatively slow devices

- was IDE

Device registers

- Registers on <u>devices</u> (not processor)
  - each register may have its own address (in device's own space)
  - registers may be written to or read from
- Device registers can cause I/O to happen
  - data-in (to be read by host)
  - data-out (to be written by host)
  - status (read by host to find I/O status, error, etc)
  - control (e.g., full/half duplex, parity, baud rate, etc)

I/O Instructions (on CPU)

- Instructions for processor to control I/O
- Direct I/O instructions (part of ISA)
  - Cause waveform to be generated for I/O
  - example: SPI, I2C, UART, ...
- Memory-mapped I/O
  - Map **device-control registers** into mem. address space of CPU
  - Load/store instructions like regular memory, but effect is to access device data and command registers
  - Especially for large address spaces (graphics)

Memory-mapped addresses of Device I/O Ports 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)

Polling vs. Interrupt

- Different ways to check if I/O is ready
- Polling: busy wait
  - loop until I/O hardware sets a flag value
  - simple to code, but not doing useful work while polling
- Interrupt
  - processor can do useful work
  - processor saves state before jumping to ISR
  - processor can do useful work or sleep (to save power)

Polling: sending vs receiving

- Sending
  - processor poll busy bit (from I/O status register).
  - I/O hardware clears busy bit on completing previous I/O.
  - processor makes sure I/O not busy before writing
- Receiving
  - processor polls data-ready bit (from I/O status register)
  - I/O hardware sets data-ready bit upon successful receiving
  - processor makes sure data is ready before trying to read it (or else getting garbage)

Issue with Polling

- Pros
  - Reasonable if device is fast
- Con
  - But inefficient if device slow
- CPU switches to other tasks?
  - But if miss a cycle data overwritten / lost
- Polling can happen in 3 instruction cycles
  - Read status, logical-and to extract status bit, branch if not zero
  - How to be more efficient if non-zero infrequently?

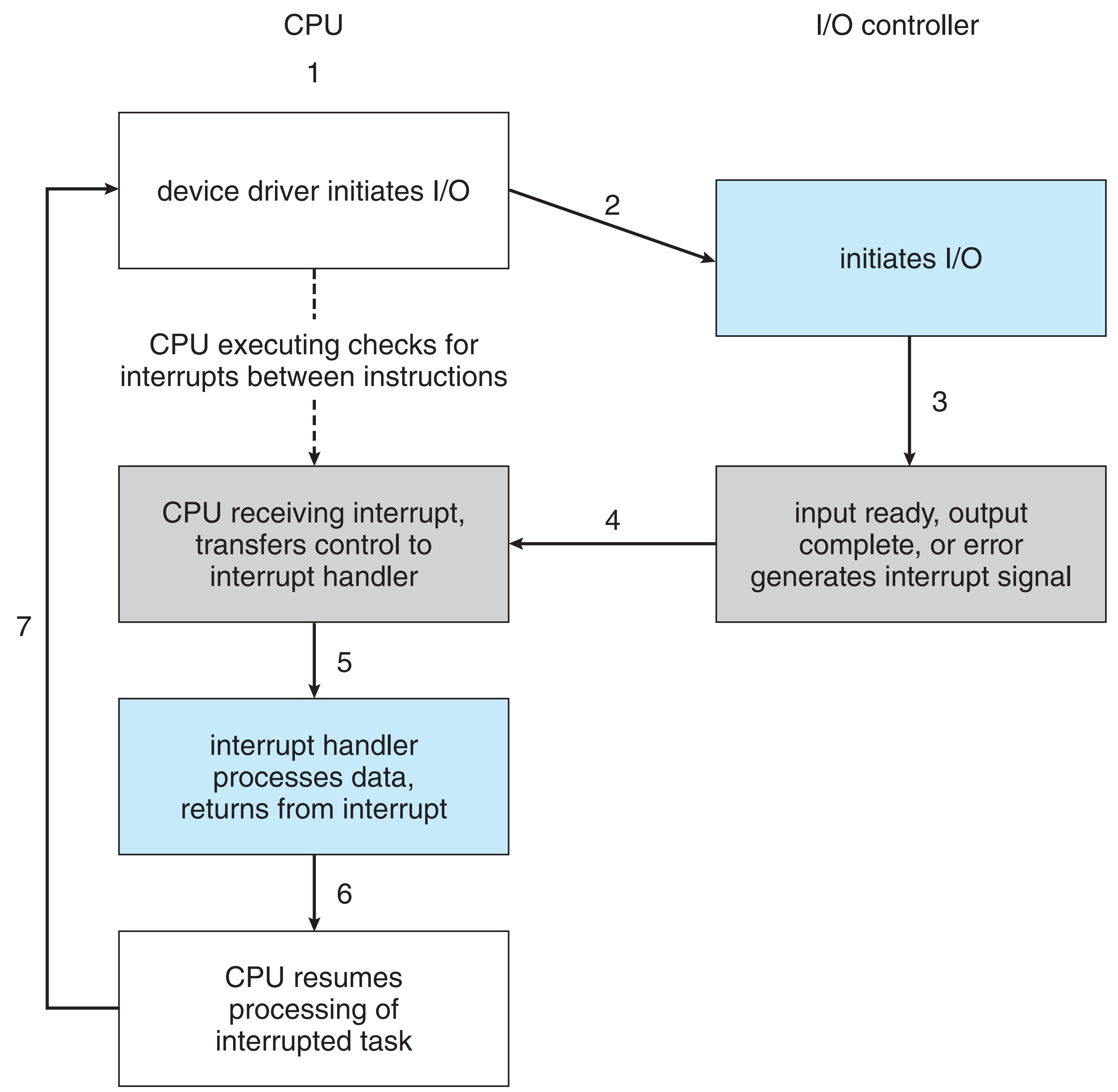
Interrupts

- CPU's Interrupt-request (IRQ) line
  - triggered by I/O device (usually pulled down)
  - Checked by CPU (hardware) after each instruction
- Interrupt handler (interrupt service routine) "receives interrupts"
  - Some interrupts are maskable => ignore or delay interrupts
  - Processor saves state of regular execution, switches context, jumps to ISR
  - Context switch at start and end

Vectored Interrupt vs Interrupt Chaining

- Q: How many IRQ lines does the CPU have?
  - Ideally, one per device => Vectored interrupt
  - often there are many more devices than IRQ lines
- Vectored interrupt
  - IRQ# => index into interrupt vector to dispatch interrupt
  - Each device gets its own ISR
  - High overhead if table is huge
- Interrupt Chaining
  - Multiple devices share an IRQ => share ISR
  - Once invoked, the ISR must query each of shared device

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 mechanism used for Exceptions and Traps

- Exceptions
  - Divide by zero, memory access violation, insufficient privilege, page fault, etc
  - ISR for OS to decide how to handle
- Trap
  - to trigger kernel for system calls
- Split interrupt management
  - first-level interrupt handler (FLIH) - actual ISR to do I/O
  - second-level interrupt handler (SLIH) - separately scheduled routine to process the data (without I/O) for the OS

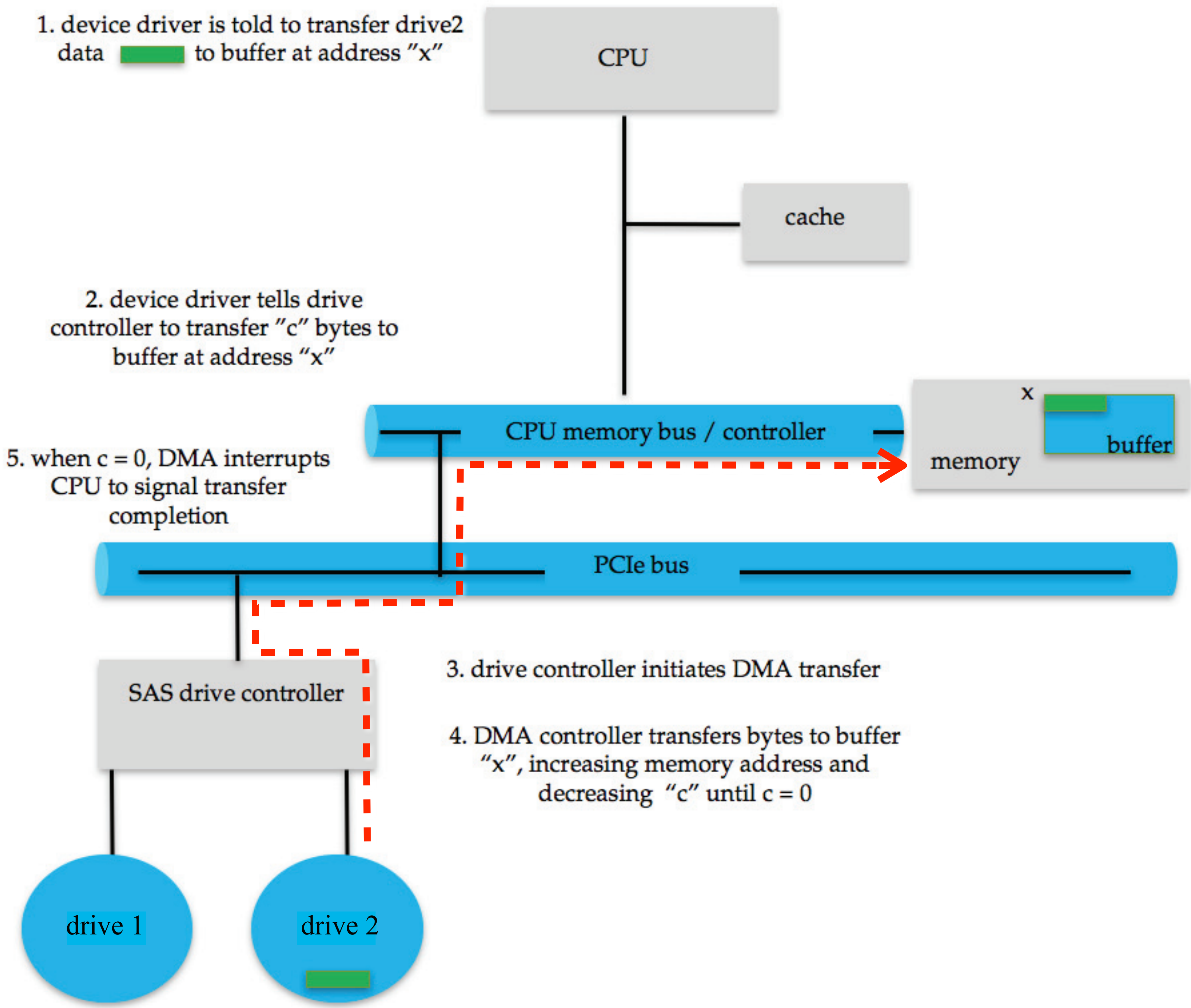
Direct Memory Access (DMA)

- Controller that offloads bulk I/O from CPU
  - Avoids programmed I/O (one byte at a time) for large data movement
  - Bypasses CPU to transfer data directly between I/O device and memory
    - CPU can do more useful work, or sleep to save power!
- OS writes DMA *command block* to DMA controller
  - Source and destination addresses of data
  - Read or write mode
  - Count of bytes

Direct Memory Access (DMA)

- Bus mastering of DMA controller
  - CPU & DMA can't use same memory at same time
  - DMA controller grabs bus from CPU
    => Cycle stealing from CPU, some slowdown, but still much more efficient than programmed I/O
  - When done, interrupts to signal completion
- Memory buffer?
  - default: kernel space, but wasteful to copy to user buffer (double buffering)
  - better to do memory mapping to map buffer to user address space
- DVMA
  - aware of virtual addresses, even more efficient

Steps in a DMA Transfer



Application I/O Interface

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

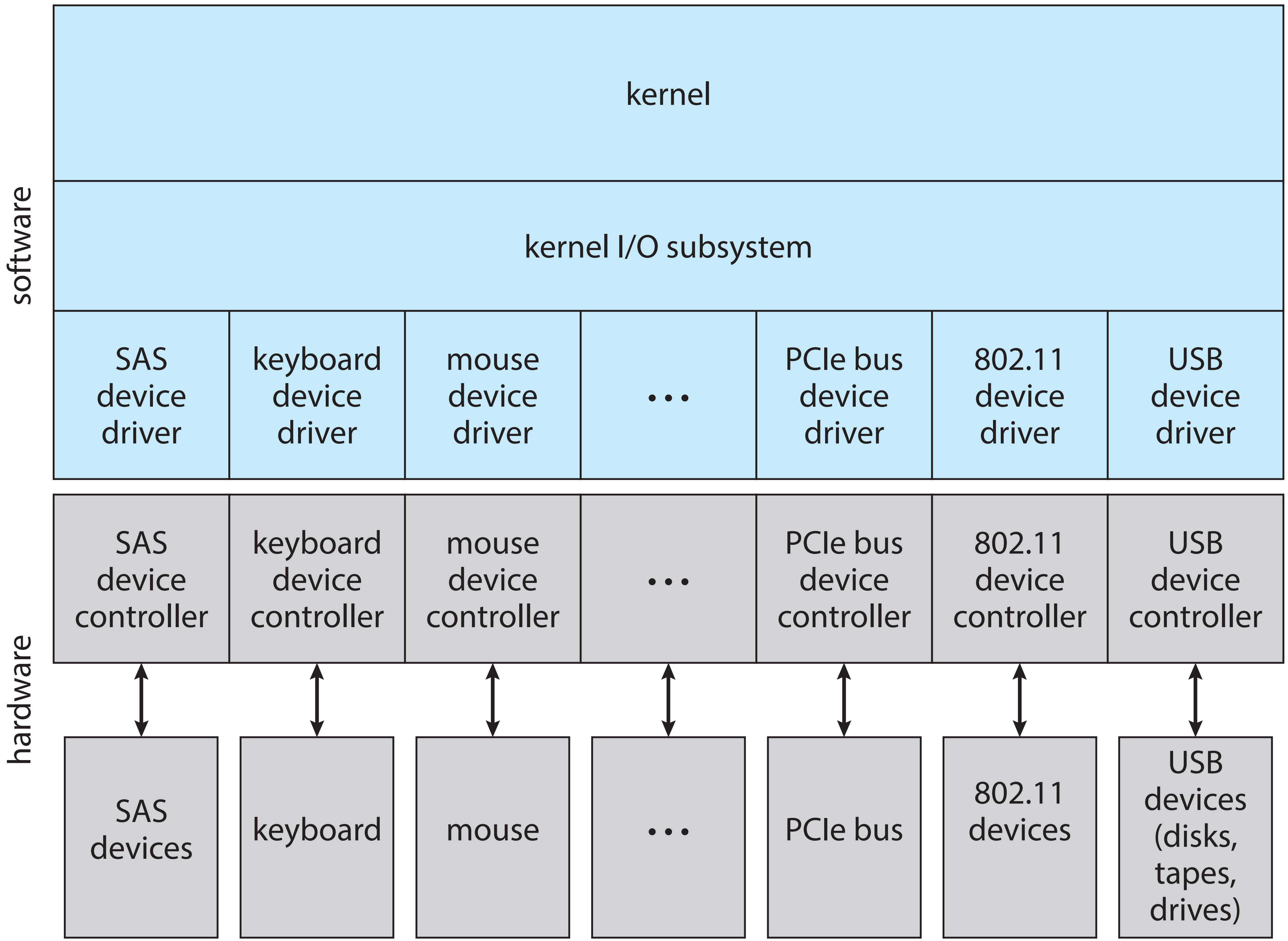
Types of Device I/O

- Data Transfer Mode
  - Character stream: one byte at a time (e.g., terminal, keyboard)
  - Block device: one whole block of data (e.g., disk)
- Access Method
  - Sequential: (e.g., webcam, modem, network card)
  - Random access: (e.g., USB drive, CD-ROM)
- Transfer Schedule
  - Synchronous: (e.g., display, tape drive)
  - Asynchronous: (e.g., keyboard, mouse)

Types of I/O devices (cont'd)

- Sharing (at a given moment)
  - Sharable among several processes: (e.g., keyboard)
  - Dedicated: (e.g., printer, tape drive)
- Speed of operation
  - latency, seek time, transfer rate, delay between operations
- I/O direction
  - read-write (e.g., disk)
  - read-only (e.g., CD-ROM, DVD-ROM, etc)
  - write-only (e.g., graphics controller, actuator)

A Kernel I/O Structure



Characteristics of I/O Devices

- 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 e.g., 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 e.g., keyboard, mouse, serial ports, printer
  - Commands include `get()`, `put()`
  - Libraries layered on top allow *line editing* (arrow keys, backspace)

Network Devices

- Higher level than block and character
- socket interface
  - Separates network protocol from network operation
  - Includes `select()` functionality - returns
    - which socket has packet waiting to be received,
    - which sockets have room to accept a packet to send
    - eliminates polling and busy waiting
- Approaches vary widely
  - pipes, FIFOs, streams, queues, mailboxes

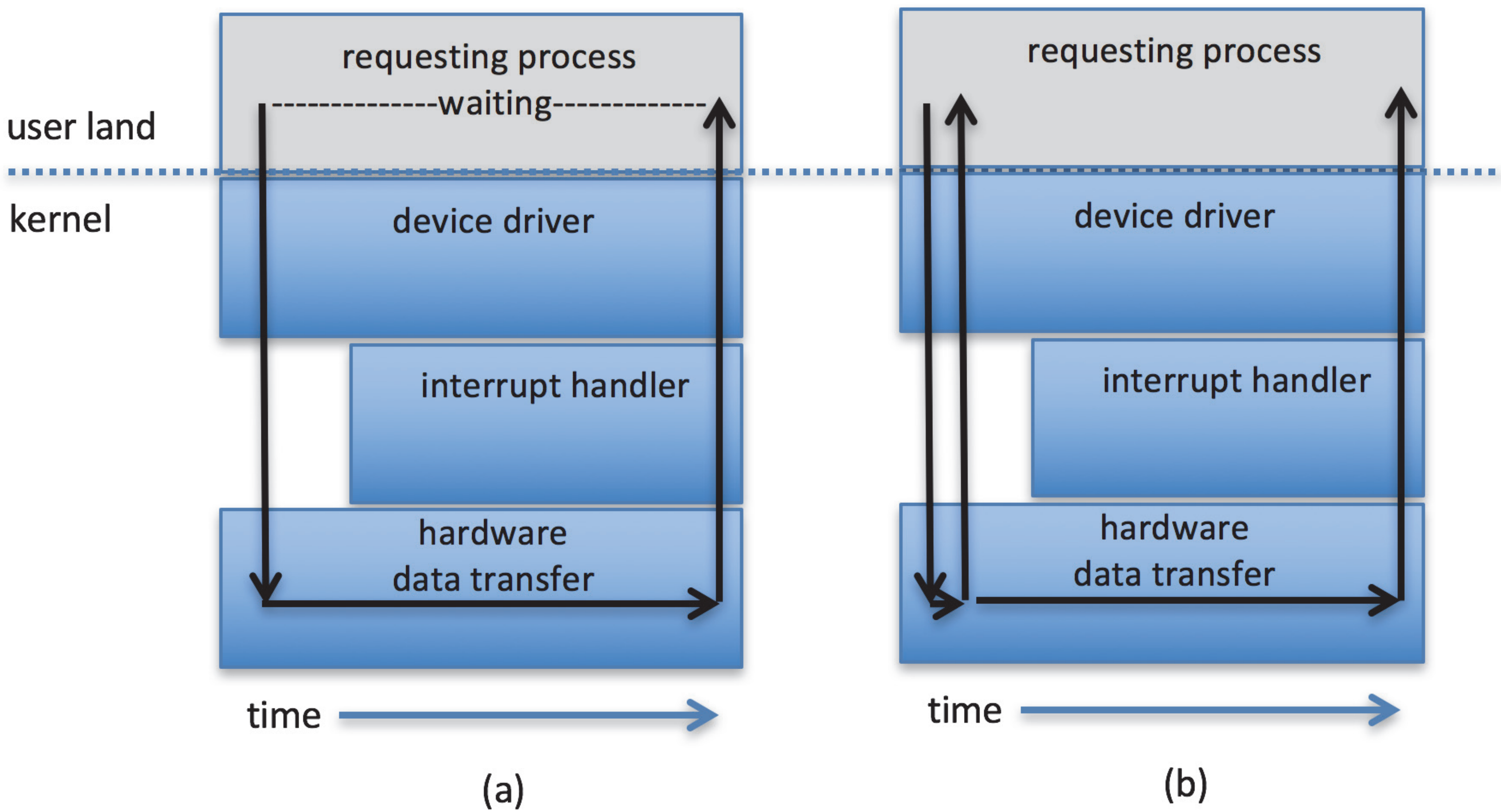
Clocks and Timers

- Provide *current time*, *elapsed time*, timer to trigger
  - Normal resolution about 1/60 second
  - Some systems provide higher-resolution timers
- Programmable interval timer
  - used for timings, periodic interrupts
- `ioctl()` (on Unix, for "I/O Control")
  - purpose: "backdoor" to pass command & pointer to driver
  - covers odd aspects of I/O such as clocks and timers
- NTP - network time protocol
  - to correct timer drift, uses latency calculation

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



Synchronous

Asynchronous

Vectored I/O

- "Vector" => think "array" of commands
- Allows one system call to perform multiple I/O operations
  - Example: Unix system call `readv()` accepts a vector of multiple buffers to read into or write from
- This is called "scatter-gather" method
  - better than multiple individual I/O calls
  - Decreases context switching and system call overhead
- Some versions provide atomicity
  - for example, avoid worrying about multiple threads changing data as reads / writes occurring

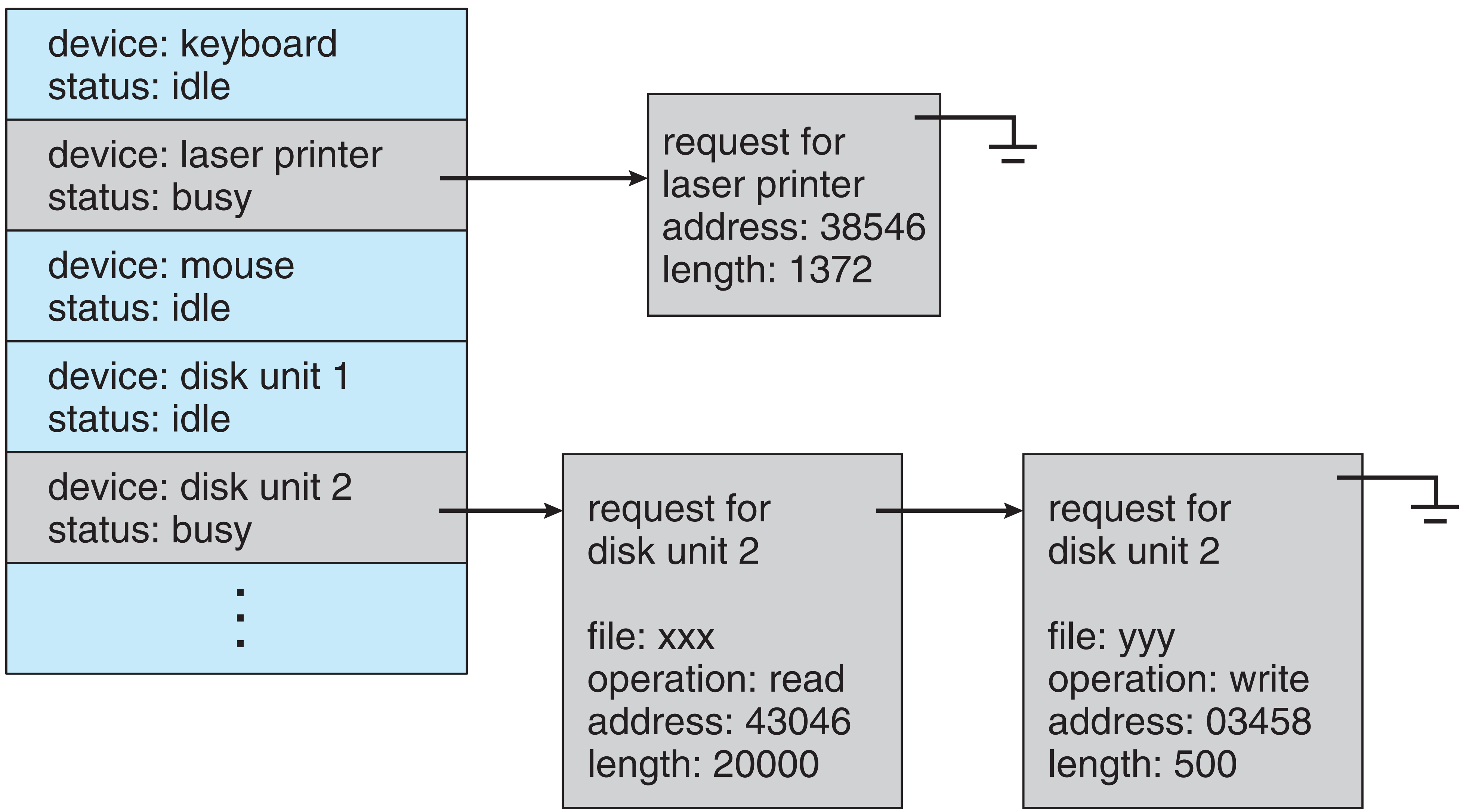
Kernel I/O Subsystem

Kernel I/O Subsystem

- First-come-first-serve usually not good
  - Different I/O speeds, seek/rotational latency
  - transfer sizes, types
- Scheduling
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness, priority,
  - Some implement Quality Of Service (i.e. IPQOS)

Device-status Table

Needed for keeping track of asynchronous I/O status



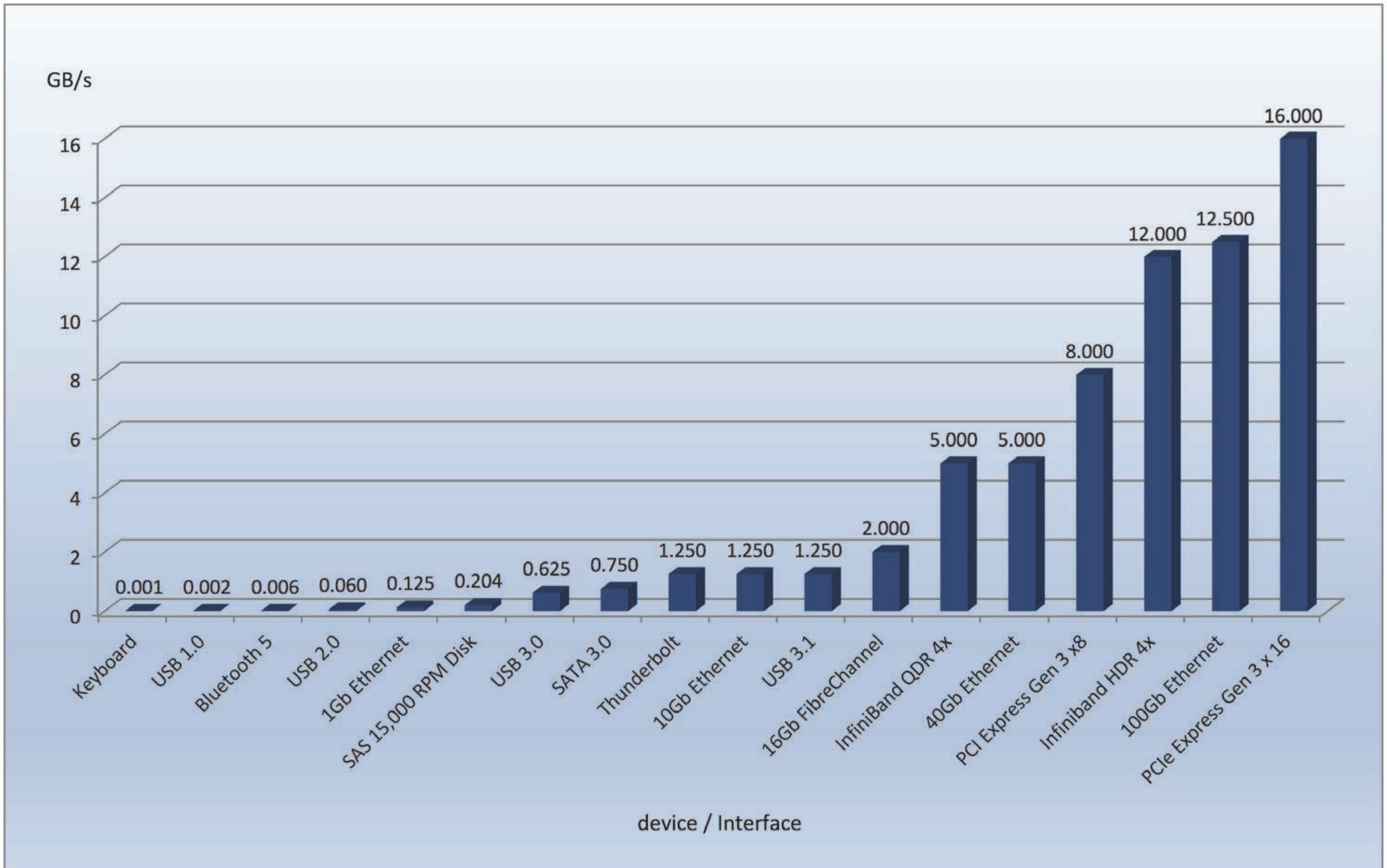
Buffering in Kernel I/O

- Between devices or
  between device and application
- Purposes: to cope with
  1. mismatched device speed between producer and consumer
  2. device transfer size: disassemble data into packets, reassemble
  3. copy semantics

Buffering - for speed mismatch

- Example
  - receive data on modem, save data to disk
  - modem: slower; buffer up before writing to disk
- Double Buffering
  - allows producer and consumer to buffer simultaneously without conflict
  - example use: bitmapped display
    graphics card fills up one frame buffer while display controller renders pixels from previous frame
    (saves jitter or incomplete frame)

Interface speeds for devices



Buffering - for different transfer sizes

- Example: network interfaces
  - Application: payload can be arbitrary size
  - TCP/IP packet: 64KB
  - Ethernet: 1500 bytes
- Buffer is needed for size matching
  - Packetize application data to IP packets for TCP/IP
  - Packetize IP packets over Ethernet
  - Assemble Ethernet packets into IP packets on receiving end
  - Assemble IP packets for application to read

Buffering - copy-semantics

- Buffer in user space vs kernel space
  - example: `write(buf)` => `buf` is user space
  - `write()` returns before I/O completes
  - What if you modify `buf` before I/O completes??
- Copy-semantics
  - `buf` data is copied before `write()` returns
    => does not matter if you modify `buf`, because the kernel already made a copy!
  - it helps to implement it with copy-on-write.

Caching in Kernel I/O

- Distinct concept from Buffering
  - Caching is a faster, redundant copy of a slower original
  - Always just a copy, Key to performance
  - buffer: app is aware; cache: app not aware
- Sometimes combined with buffering
  - especially disk access, avoid physical I/O

Kernel I/O Subsystem
Spooling vs. Device Reservation

- Spooling
  - hold output for a device
  - If device can serve only one request at a time e.g., Printing
  - OS prints to spool file, queues print jobs
- Device reservation
  - provides exclusive access to a device
  - System calls for allocation and de-allocation
  - Watch out for deadlock

Error Handling

- Errors in I/O
  - 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
- Handling
  - 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

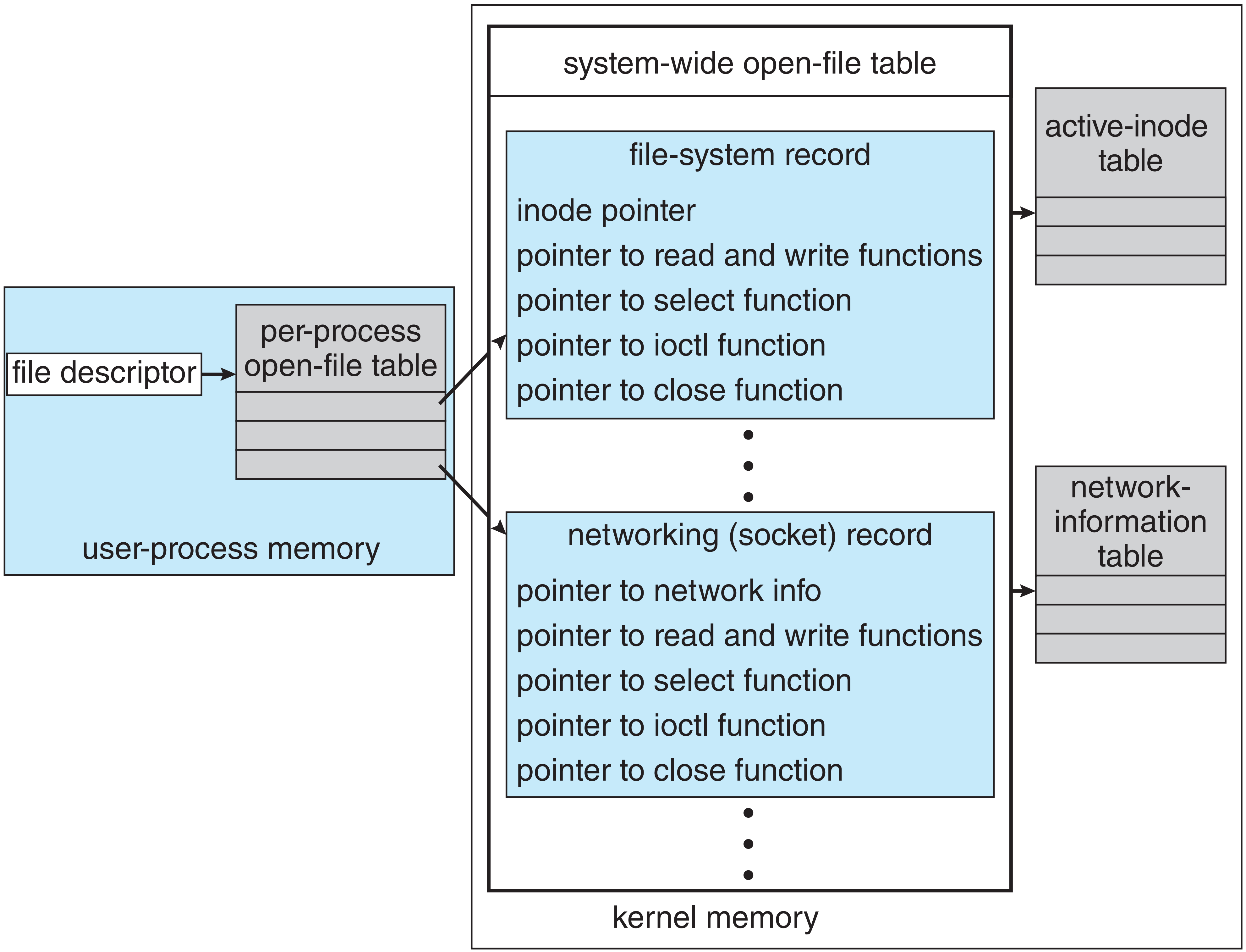
I/O Protection

- Types of disruption
  - accidental (program bug)
  - purposeful (external attack - infected virus)
- OS Protects against illegal I/O instructions
  - All I/O instructions defined to be privileged
  - I/O must be performed via system calls
  - Protect Memory-mapped and I/O port locations

Kernel Data Structures

- State of I/O components
  - open file tables
  - network connections
  - character device state
- Buffers, memory allocation, “dirty” blocks
- Object-oriented methods and message passing for I/O
  - example: 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

UNIX I/O Kernel Structure



Power Management

- CPU speed and voltage
  - Dynamic Voltage/Frequency Scaling (DVFS)
  - Sprint-and-halt CPU mode setting
- I/O Devices
  - Dynamic Power Management (DPM): device on/off, mode setting.
- OS supports power management at different levels
  - One computing system (mobile, laptop, server, ...)
  - Cloud computing environments
    - move virtual machines between servers
    - Can control and shutting down whole systems

Power Management in Mobile OS

- Component-level power management
  - Understands relationship between components
  - OS builds device tree representing physical device topology
    - System bus -> I/O subsystem -> {flash, USB storage}
- Device driver tracks state of device use
  - Unused component – turn it off
  - All devices in tree branch unused – turn off branch

Power Management "knobs" in Mobile OS

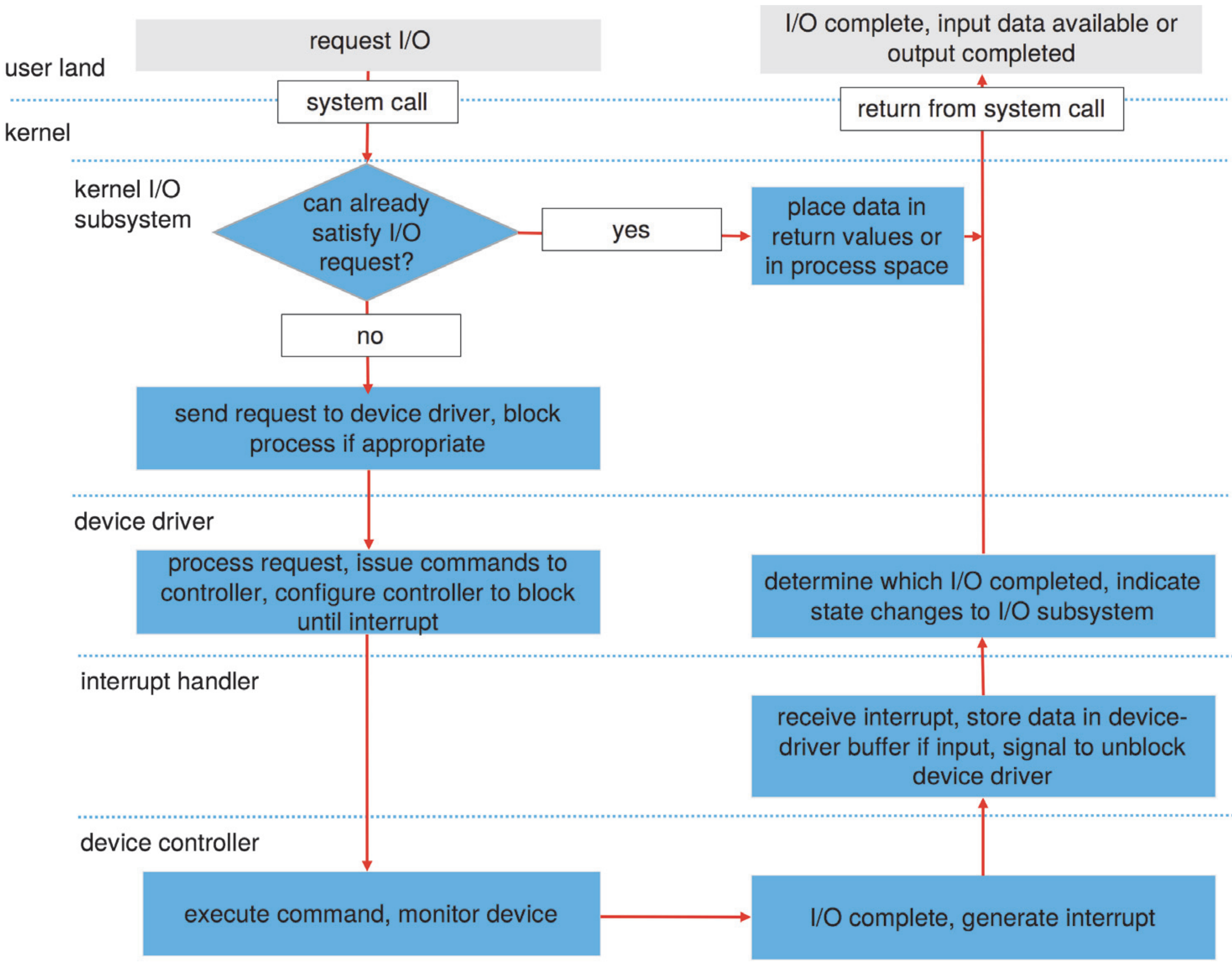
- Wake locks
  - like other locks but prevent sleep of device when lock is held
  - example: screen dimming while doing slide show
- Power collapse
  - put a device into very deep sleep, marginal power use
  - Only awake enough to respond to external stimuli
    - e.g., button press, incoming call
- ACPI - Advanced Configuration & Power Interface
  - industry standard firmware code callable from kernel to manage device power

I/O Requests to Hardware Operations

- Consider reading a file from disk for a process

1. Determine device holding file
   - [OS queries table filled with data from hardware]
2. Translate name to device representation
   - [OS invokes driver]
3. Physically read data from disk into buffer
   - [OS invokes driver for hardware operation]
4. Make data available to requesting process
5. Return control to process

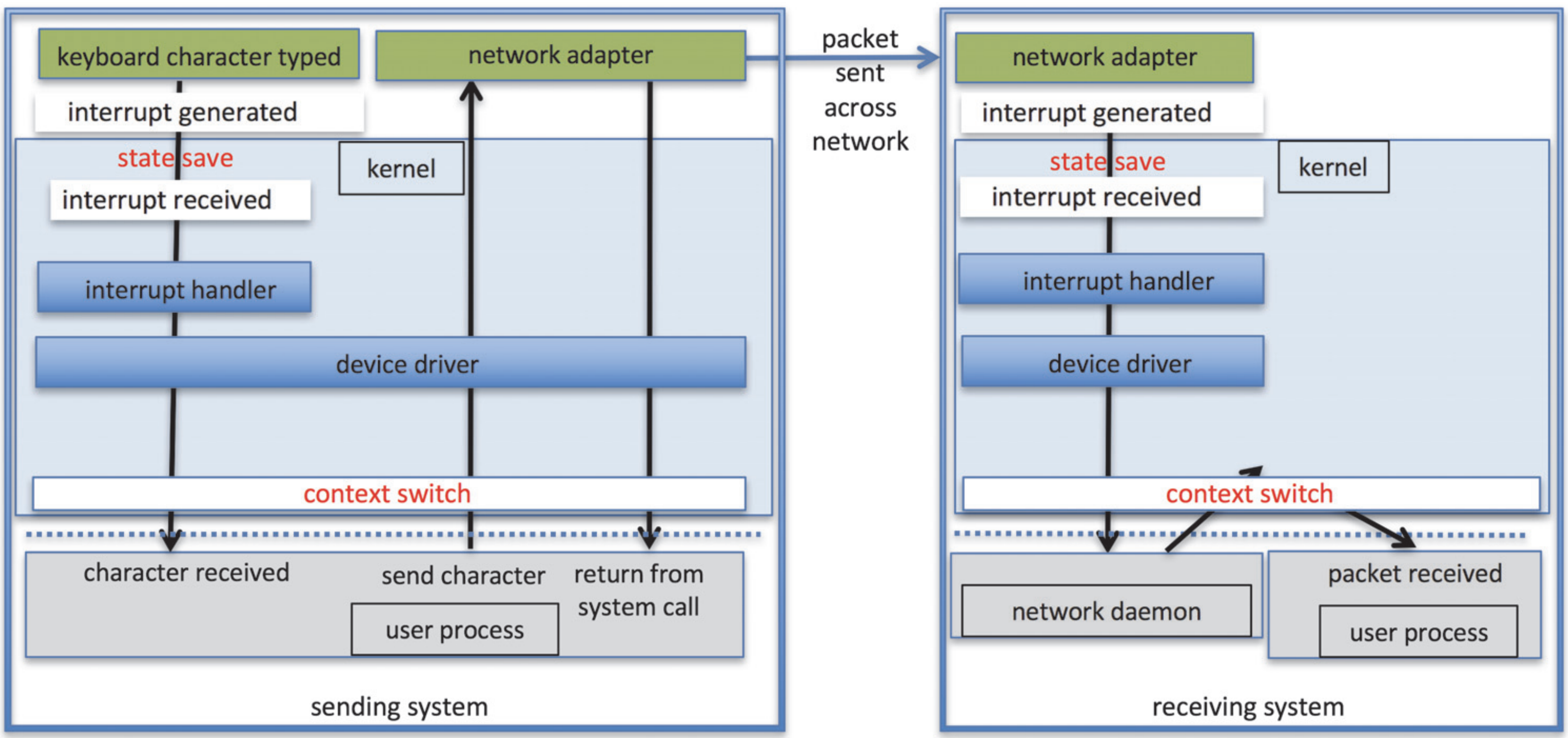
Life Cycle of An I/O Request



Performance

- I/O a major factor in system performance
- OS is impacted by
  - Code execution of
    - device driver
    - kernel I/O code
  - Context switches due to interrupts
  - Data copying
- In general, need to balance
  - CPU, memory, bus, I/O performance

Intercomputer Communications



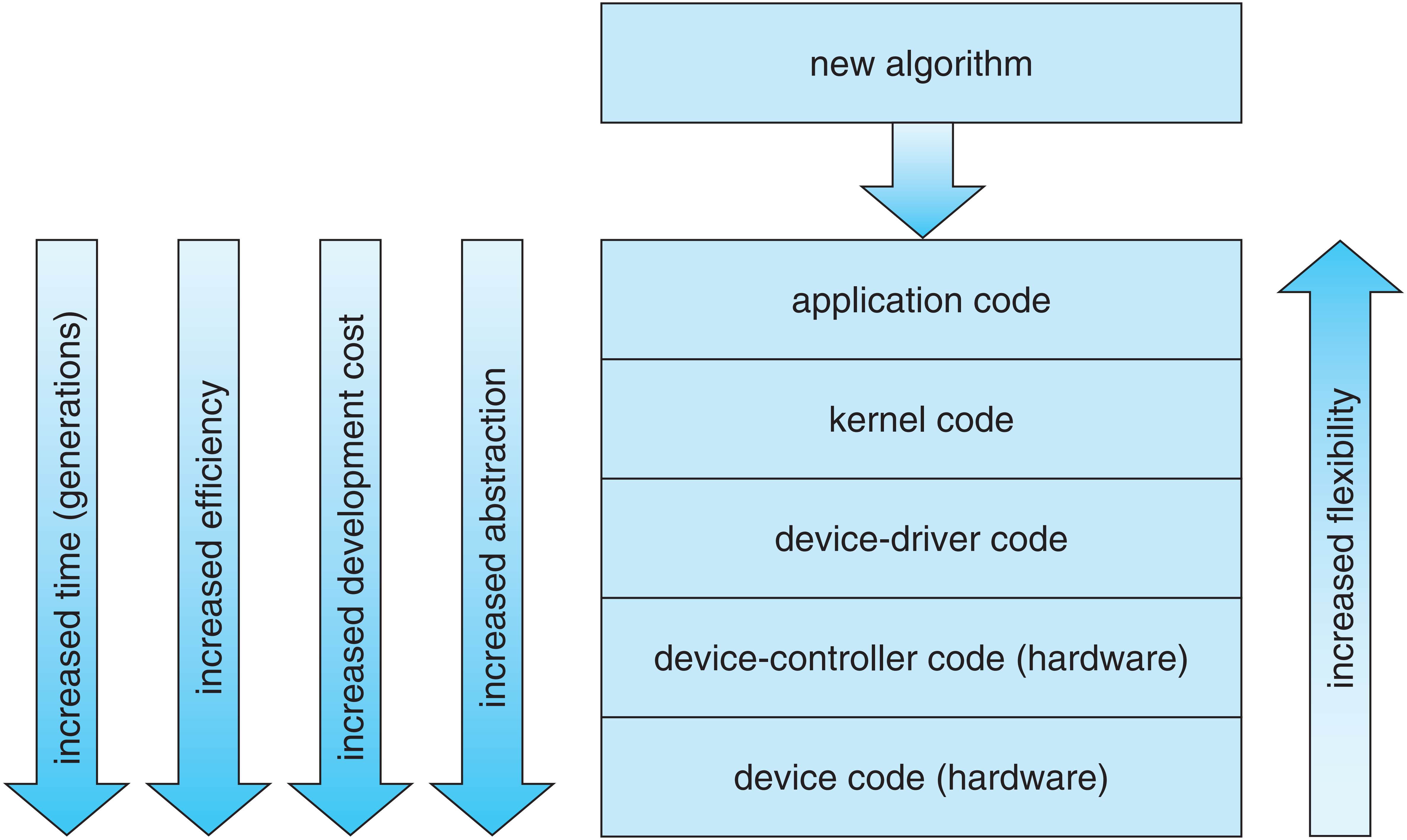
Improving Performance - CPU

- Reduce number of context switches
  - e.g.,: front-end processor, terminal concentrator
- Move user-mode processes / daemons to kernel threads
  - e.g.,: Solaris uses in-kernel thread for telnet daemon
- Reduce data copying
  - e.g.,: copy-on-write for I/O buffer

Improving Performance - Controller

- Large transfers by DMA
  - DMA controller is often built-in to processors
- Small transfers by polling
  - assuming busy-waiting can be minimized
- Smart controllers in device and on computer system
  - e.g., RAID controller

Device-Functionality Progression



I/O Performance of Storage and Network latency