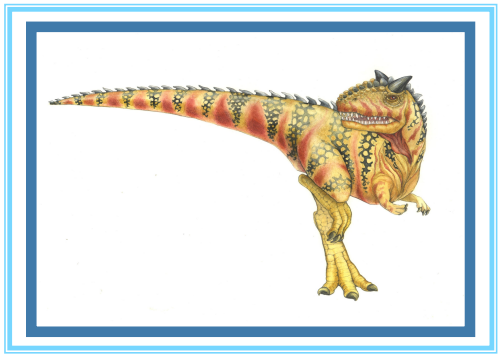
Chapter 13: I/O Systems



Chapter 13: 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

Un peu de français

- http://gdt.oqlf.gouv.qc.ca : Grand Dictionnaire Terminologique de l'Office Québécois de la Langue Francaise
- http://translate.google.com/

kernel	noyau
device controller	contrôleur de périphérique
port	port, interface avec un canal de communication
stream	train, action de transmettre en continu
multiplexing	multiplexage

Overview

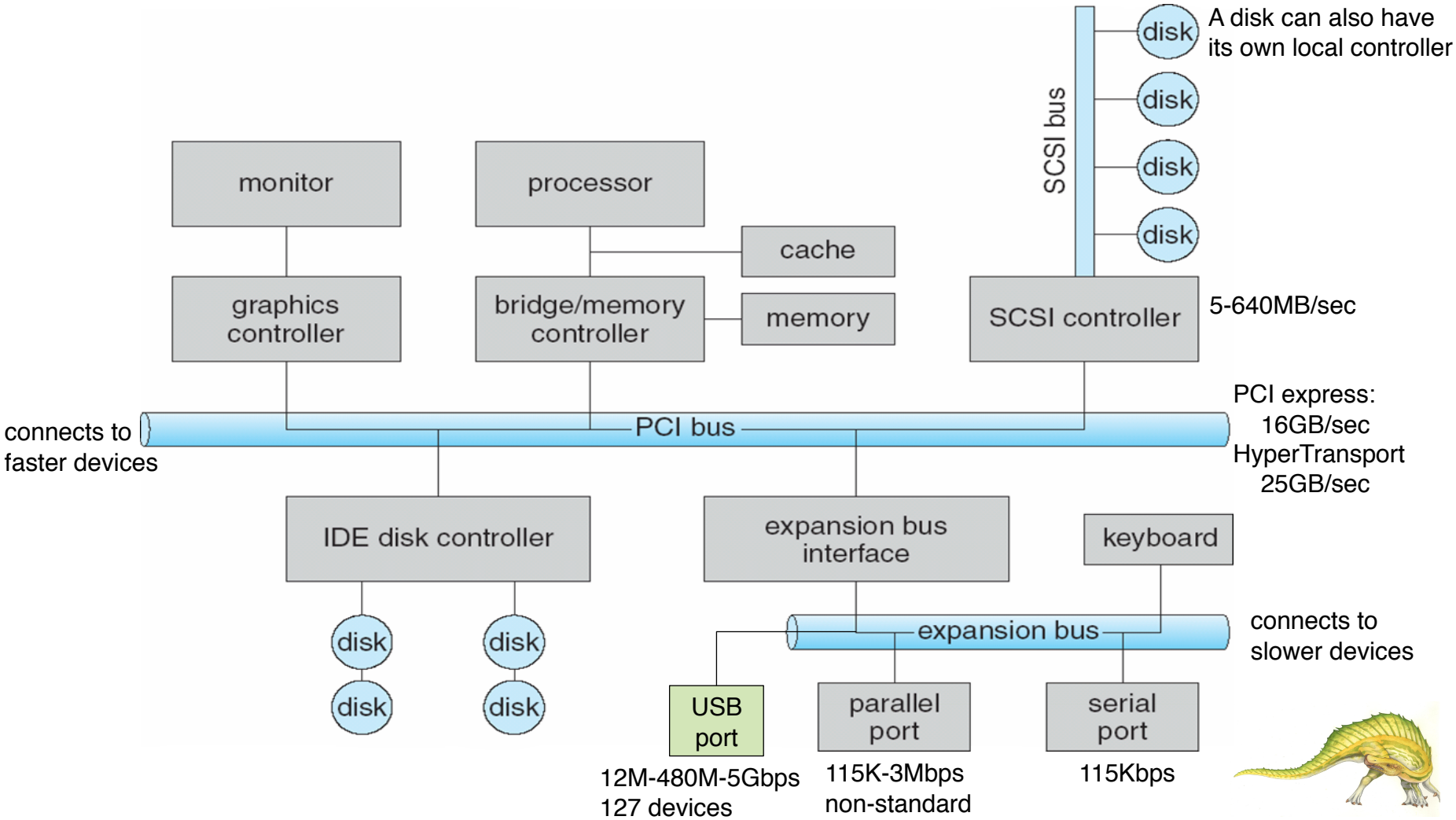
- I/O management is a major component of operating system design and operation
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent
- Ports, buses, device controllers connect to various devices
- **Device drivers** encapsulate device details
  - Present uniform device-access interface to I/O subsystem
  - Needs to standardize the interfaces
  - But some new devices may be very different from previous ones

I/O Hardware

- Incredible variety of I/O devices
  - Storage (disks, tapes, etc.)
  - Transmission (network connections, Bluetooth, etc.)
  - Human-interface (screen, keyboard, mouse, audio, video, etc.)
  - Specialized (jet joystick/control column, etc.)

- Common concepts – signals from I/O devices interface with computer
  - **Port** – connection point for device (e.g., serial, parallel, USB)
  - **Bus** – **daisy chain** (a chain of cables, acting as a bus) or shared direct access
  - **Controller** (**host adapter**) – electronics that operate port, bus, device
    - Sometimes integrated
    - Sometimes separate circuit board (host adapter)
    - 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



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 Hardware (cont.)

- I/O instructions from processor to control devices
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - Four registers: data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped I/O**
    - Device data and command registers mapped to processor address space
    - Especially for large address spaces (graphics)

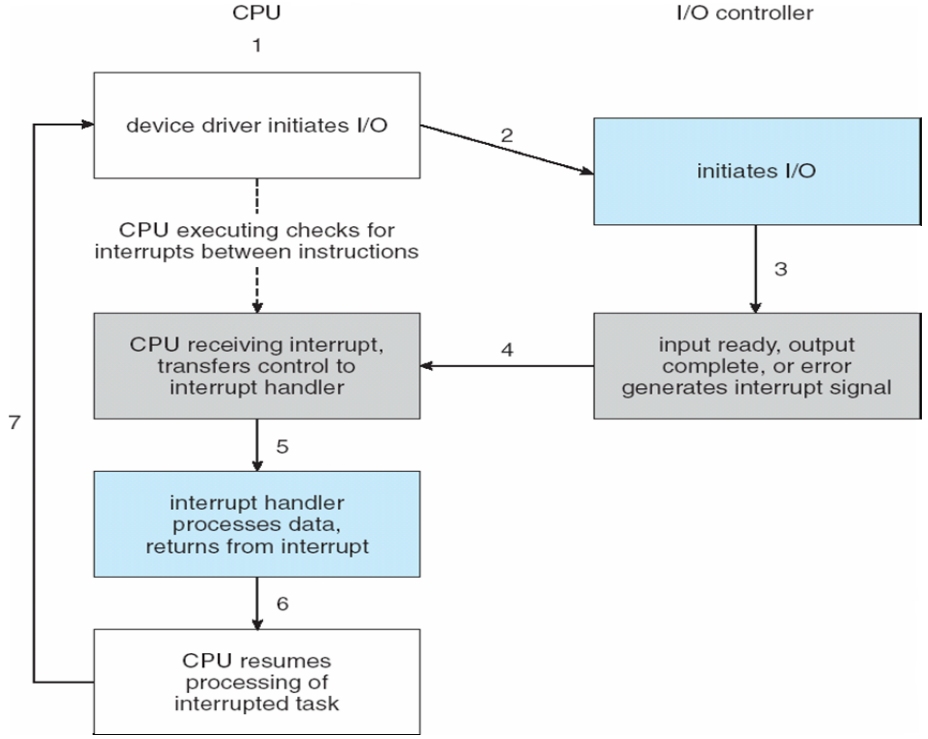
Polling

- In a handshaking protocol, for each byte of I/O
  1. Host reads busy bit from status register until 0
  2. Host sets read or write bit and if write, copies data into data-out register
  3. Host sets command-ready bit
  4. Controller sets busy bit
  5. Controller reads command register, reads data-out register, executes transfer
  6. Controller clears busy bit, error bit, command-ready bit when transfer done

- Step 1 is **busy-wait** (or **polling**) cycle to wait for I/O from device
  - Reasonable if device is fast
  - But inefficient if device slow
  - If CPU switches to other tasks, or reduces check frequency
    - But if miss a cycle, data overwritten/lost

Interrupts

- 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?
- CPU **Interrupt-request line** triggered by I/O device
  - Checked by processor **after each** instruction
- **Interrupt handler** receives interrupts
  - **Maskable** to ignore or delay some interrupts
- **Interrupt vector** to dispatch interrupt to proper handler
  - Context switch at start and end
  - Based on priority
  - Some **nonmaskable** (e.g., unrecoverable memory errors)
  - Interrupt chaining if more than one device at same interrupt number



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

non-maskable

maskable

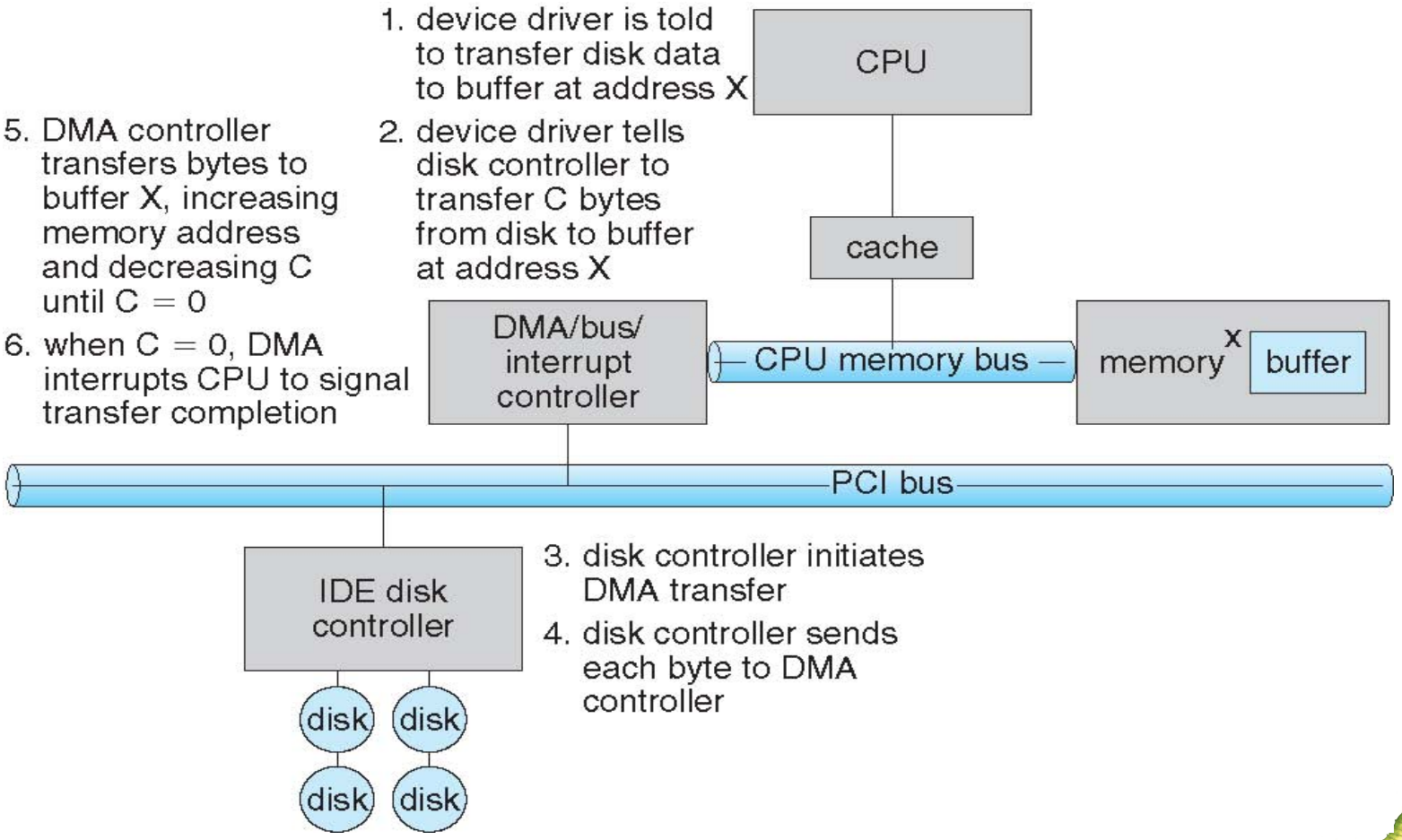
Interrupts (cont.)

- Interrupt mechanism also used for exceptions
  - Terminate process, crash system due to hardware error
- Page fault executes an interrupt when memory access error in virtual memory paging
  - Suspend process, jump to page-fault handler, move process to wait queue, page-cache management, schedule I/O, schedule another process, return from interrupt
- System call executes via **trap** (**software interrupt**) to trigger kernel to execute request
  - Of lower interrupt priority compared to device interrupts
- Multi-CPU systems can process interrupts concurrently
  - If operating system designed to handle it
- Interrupt mechanisms are used for time-sensitive processing, frequent, must be fast, and manage different priorities

Direct Memory Access

- Used to avoid **programmed I/O** (one byte at a time) for large data movement that would slow down the CPU
  - 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
    - DMA grabs bus from CPU
    - CPU can access only its primary/secondary caches
    - Could result in CPU slow down, but generally improves the overall system performance
  - When done, interrupts to signal completion

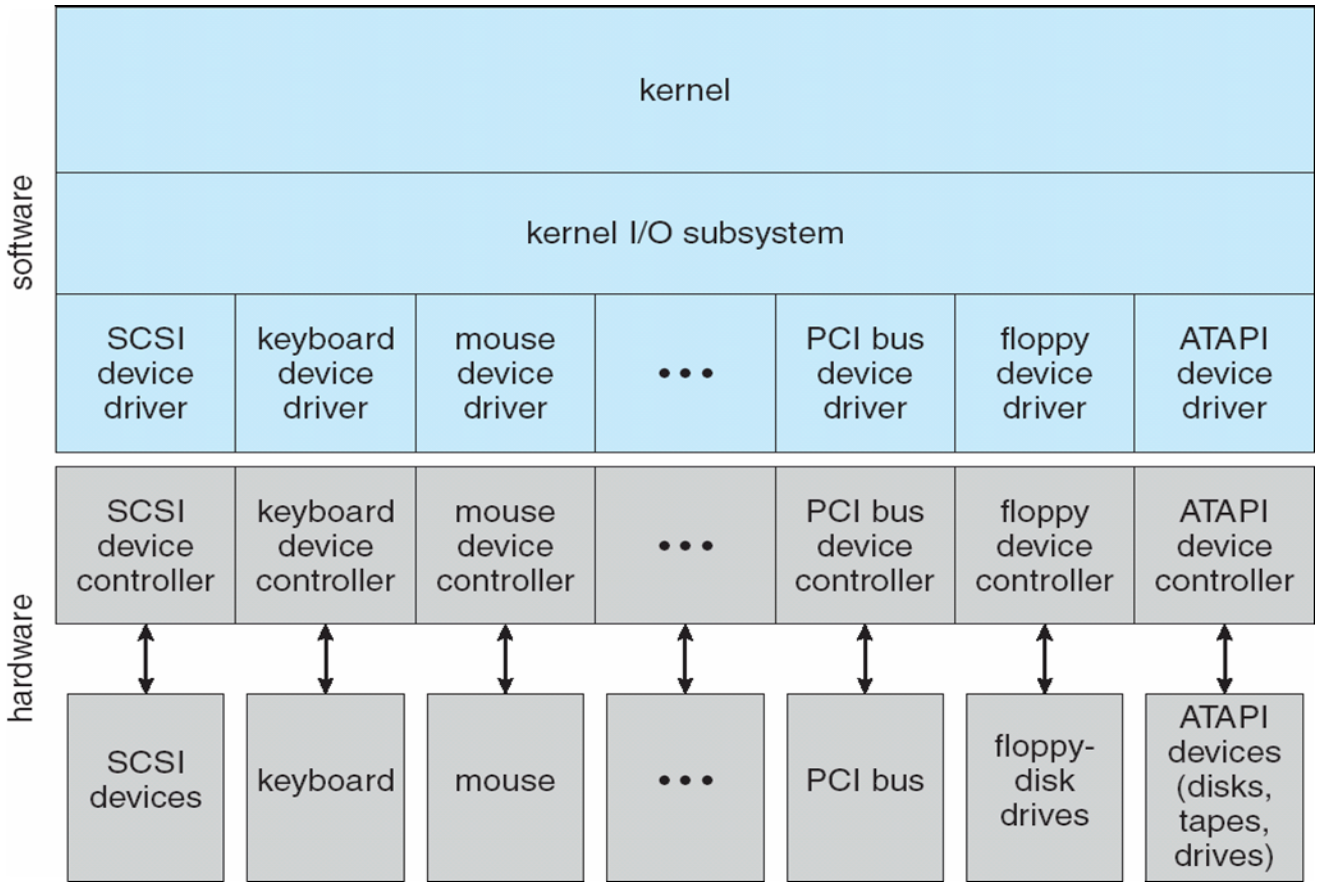
Six Step Process to Perform DMA Transfer



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**
  - **Synchronous** or **asynchronous** (or both)
  - **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

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 offers an escape/back door
  - UNIX `ioctl()` call to send arbitrary bits to a device control register and data (structure) 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
  - Convenient for spontaneous such input, and also for output devices such as printers, audio boards, etc.

Network Devices

- Varying enough from block and character to have their own interface
- UNIX and Windows NT/9*x*/2000 include **socket** interface
  - Separates network protocol (hidden) from network operation; encapsulates the essential behaviors of networks
  - Allows for socket creation, connection to remote address, listening for plugging into its local socket, sending/receiving packets, etc.
  - Includes `select()` functionality
- Approaches vary widely, and provides other communication methods, e.g., pipes, FIFOs, streams, queues, mailboxes

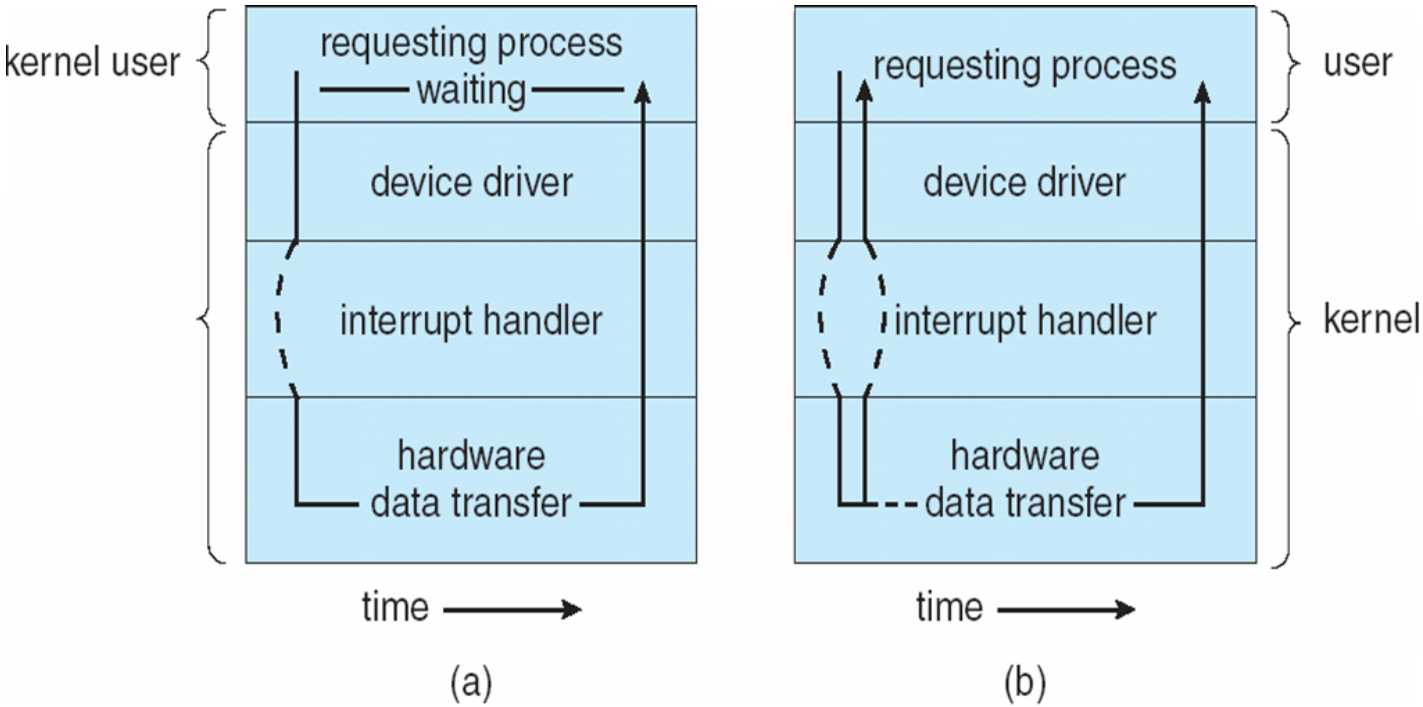
Clocks and Timers

- Provide
  - current time
  - elapsed time
  - timer (to trigger an operation at a given time)
- These functions are heavily used by OS (and user libraries)
  - schedulers for time slice
  - periodic flush of dirty cache buffers to disk
  - cancels for delayed network responses
- 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

Blocking and Nonblocking I/O

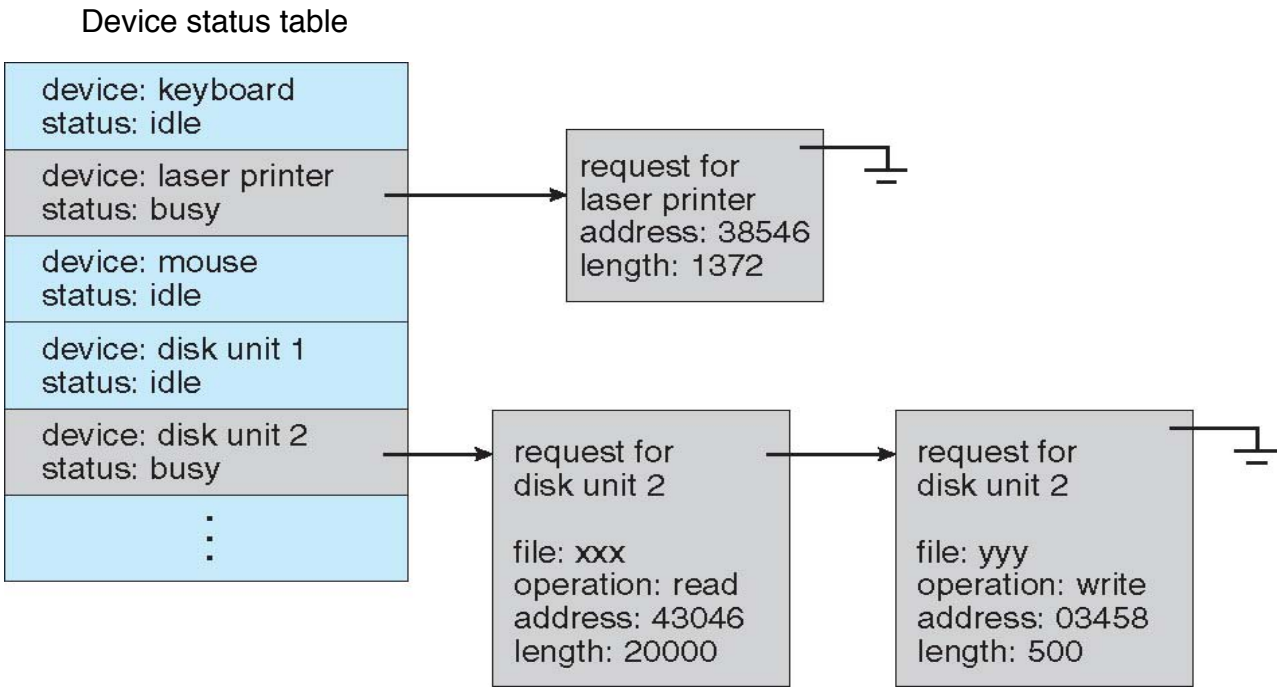
- System calls can be blocking or nonblocking I/O
- **Blocking** - process suspended until I/O completed
  - Process moves from running to waiting queue
  - Easy to use and understand
  - Insufficient for some needs
- **Nonblocking** - I/O call returns as much as available
  - User interface, data copy (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, might require to use locking mechanisms
  - I/O subsystem signals process when I/O completed

Two I/O Methods



Kernel I/O Subsystem - Scheduling

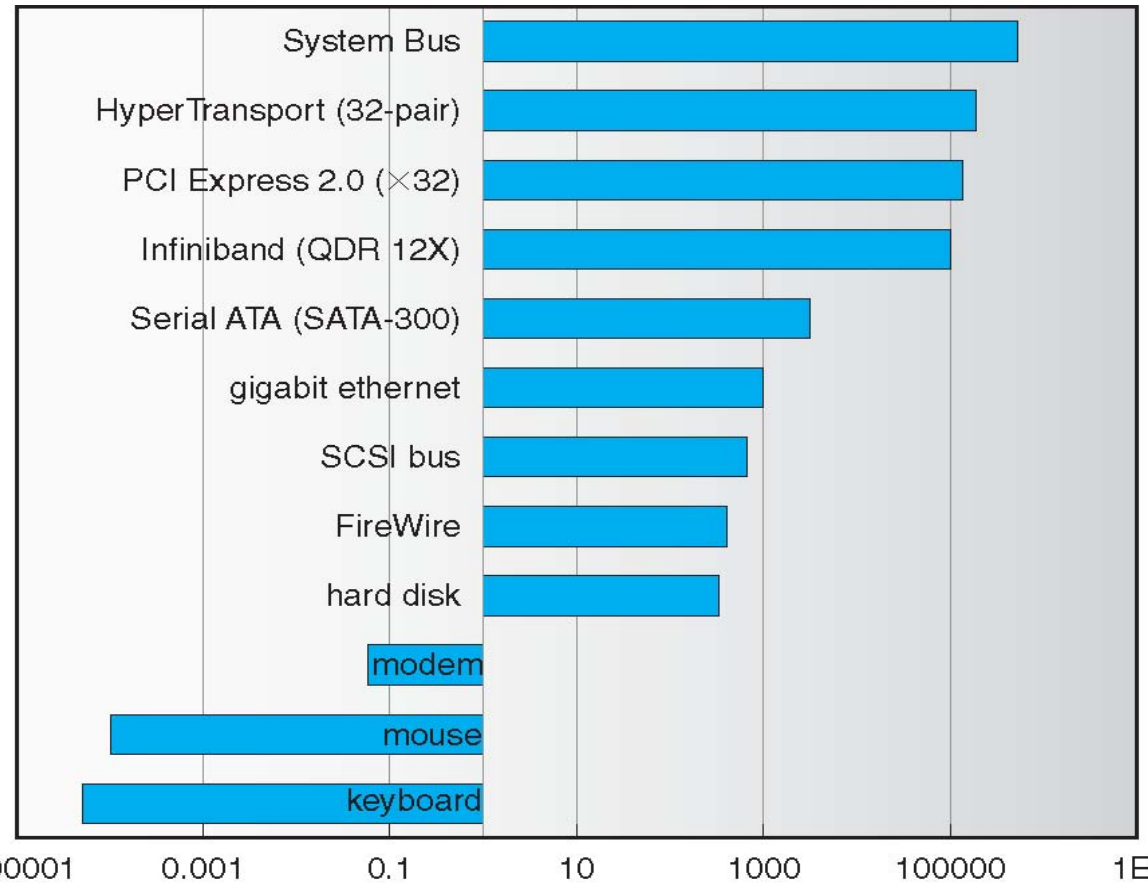
- I/O scheduling
  - Some I/O request ordering via per-device queue
  - Some OSes try to provide fairness, reorder for efficiency, handle priorities, etc.
  - Some implement Quality Of Service (e.g., IPQOS)



Kernel I/O Subsystem - Buffering

- Buffering - store data in memory while transferring between devices
  - To cope with device speed mismatch
    - E.g., modem to disk, 1000x slower, needs a second buffer (**double buffering**) so the modem continues transferring when disk controller copies one full buffer on its disk
  - To cope with device transfer size mismatch
    - E.g., large message split into small packets over the network and reassembled at the receiving end
  - To maintain “copy semantics”
    - E.g., the buffer to write to disk is copied from the application space to the kernel space, and it is the kernel copy that is written to disk; more costly, but ensure consistency of the write command
    - This can be accomplished also with copy-on-write command
  - 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

Sun Enterprise 6000 Device-Transfer Rates



This table shows how buffering between very different transfer rates can be beneficial

Kernel I/O Subsystem - Other Tools

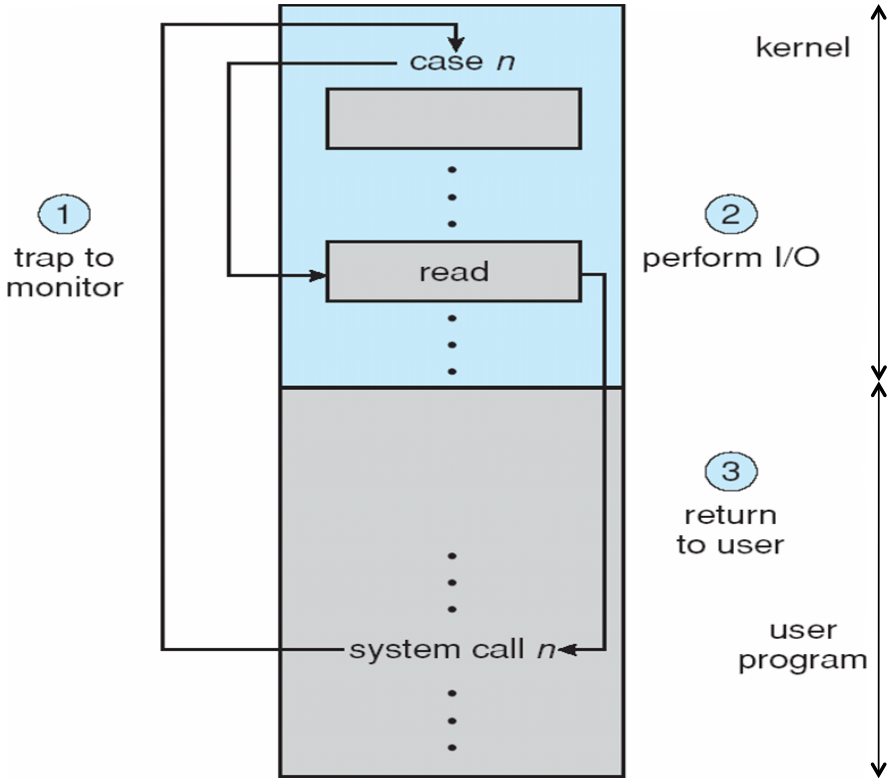
- **Caching** - faster device holding a copy of the data
  - Always just a **copy** (while buffer does not imply another copy somewhere else)
  - Key to performance, as cache is consulted first instead of more delayed original data source
  - Sometimes combined with buffering
    - E.g., buffer for disk I/O can also be used as a cache for other reads
    - E.g., disk writes are delayed/accumulated in buffer caches for efficient disk write schedules
- **Spooling** - device holds output for a device
  - If device can serve only one request at a time (no multiplexing)
  - Can allow for more than one job into a spool queue (thus allow for job removal, or suspension)
  - E.g., printer, tape
  - **Device reservation** - provides exclusive access to a device
    - System calls for allocation and de-allocation
    - Watch out for deadlock

Error Handling

- OS can try to recover from disk read, device unavailable, transient (network) write failures
  - Simple: OS issues a retry for a re-read, re-write, re-send, for example
  - Some systems are more advanced – Solaris FMA, AIX
    - Track error frequencies, stop using device with increasing frequency of retry-able errors
- Most error returns an error number or code when I/O request fails
  - But not all OSes return the complete error codes to the application
- System error logs hold problem reports

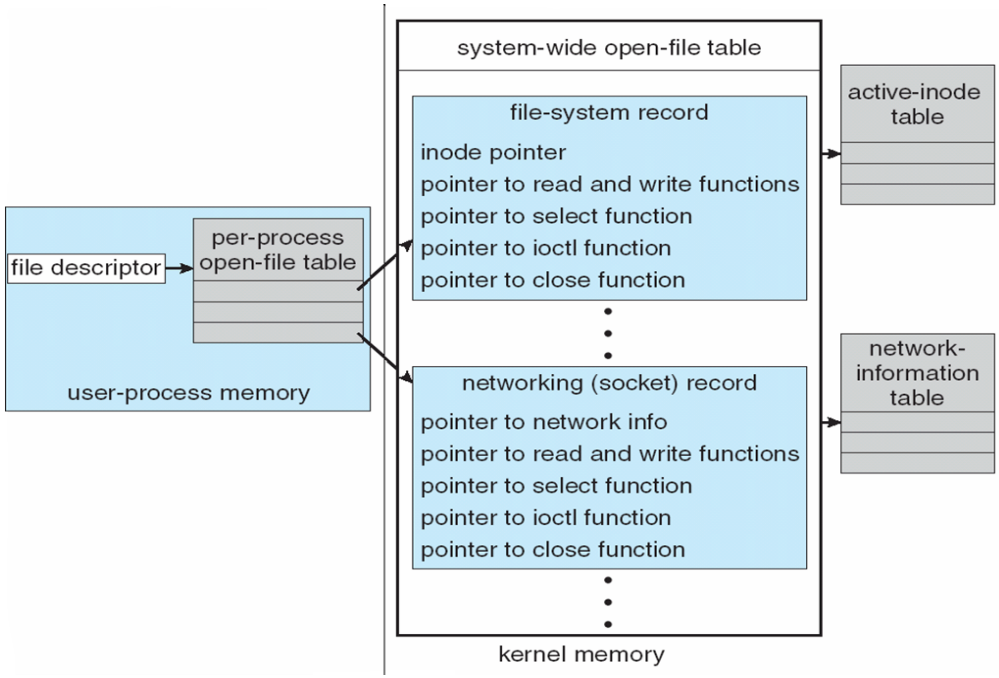
I/O Protection

- Errors and protection are closely related
- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
- One solution is that all I/O instructions be defined to be privileged
- I/O must be performed via **system calls**, first validated by OS
- Memory-mapped and I/O port memory locations must be protected too, sometimes (e.g., graphics memory) allocated to one process at a time



Kernel Data Structures

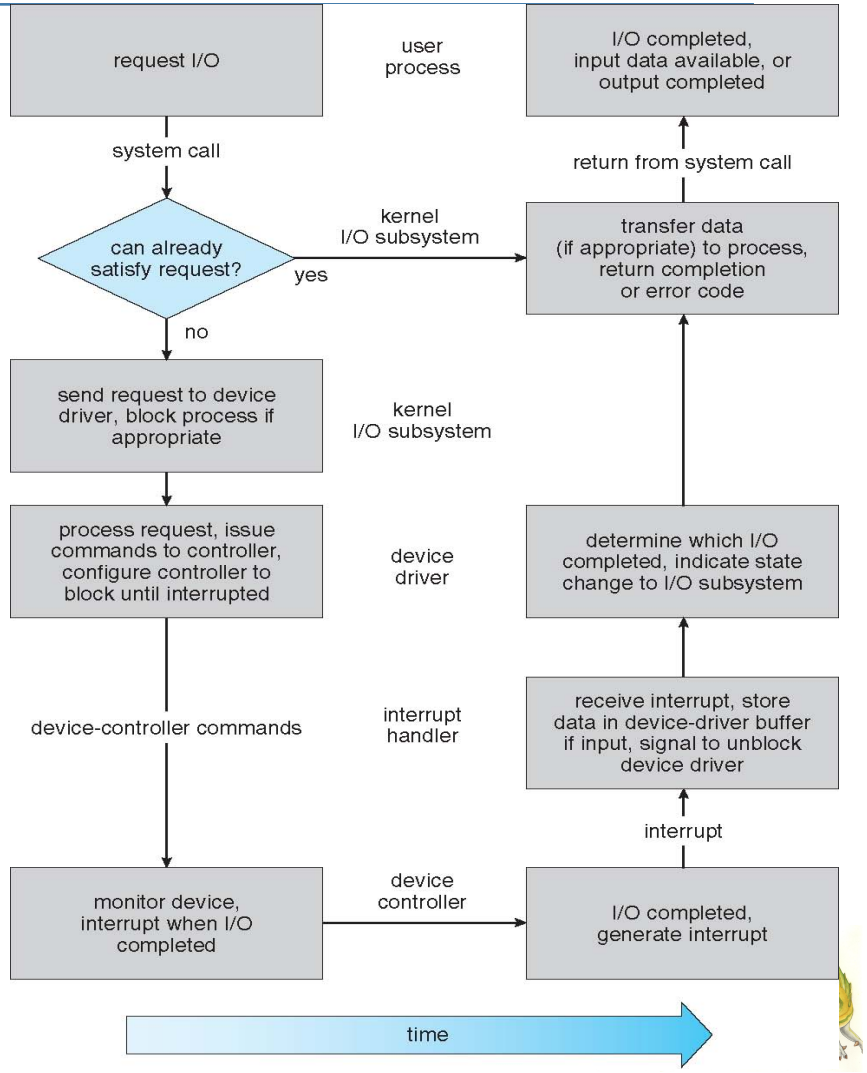
- Kernel keeps state info for I/O components, including open file tables, network connections, character device state, etc.
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks, etc.
- Some use object-oriented (UNIX) and message-passing (Windows) methods to implement I/O



Object-oriented technique to call the proper function in UNIX

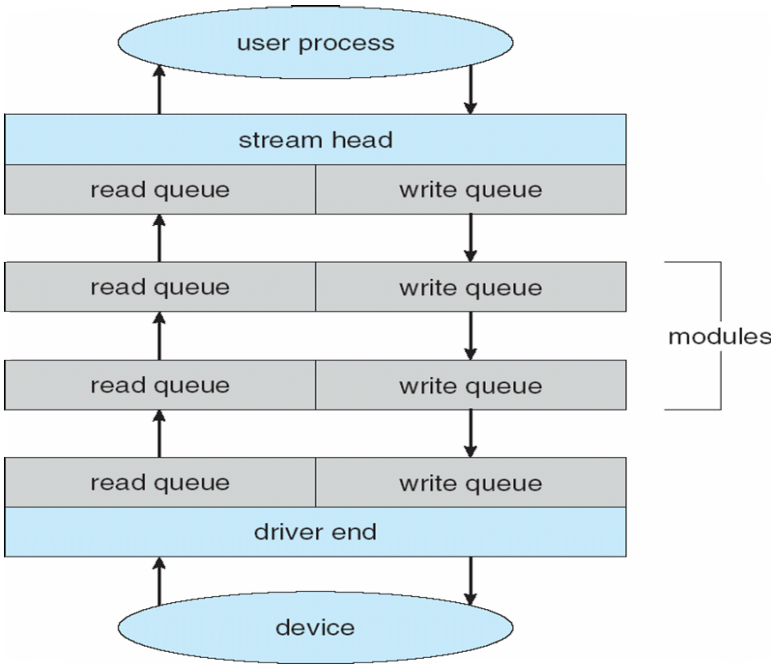
I/O Requests to Hardware Operations

- Tremendous number of CPU cycles for an I/O operation
- Consider reading a file from disk for a process:
  - Determine device holding file
    - C: in MS-DOS, mount table in UNIX
  - Translate name to device representation
  - Physically read data from disk into buffer
    - but first check if in cache
  - Make data available to requesting process
  - Return control to process
  - Figure to the right shows some of the tasks, but even there, it does not show all the steps in detail



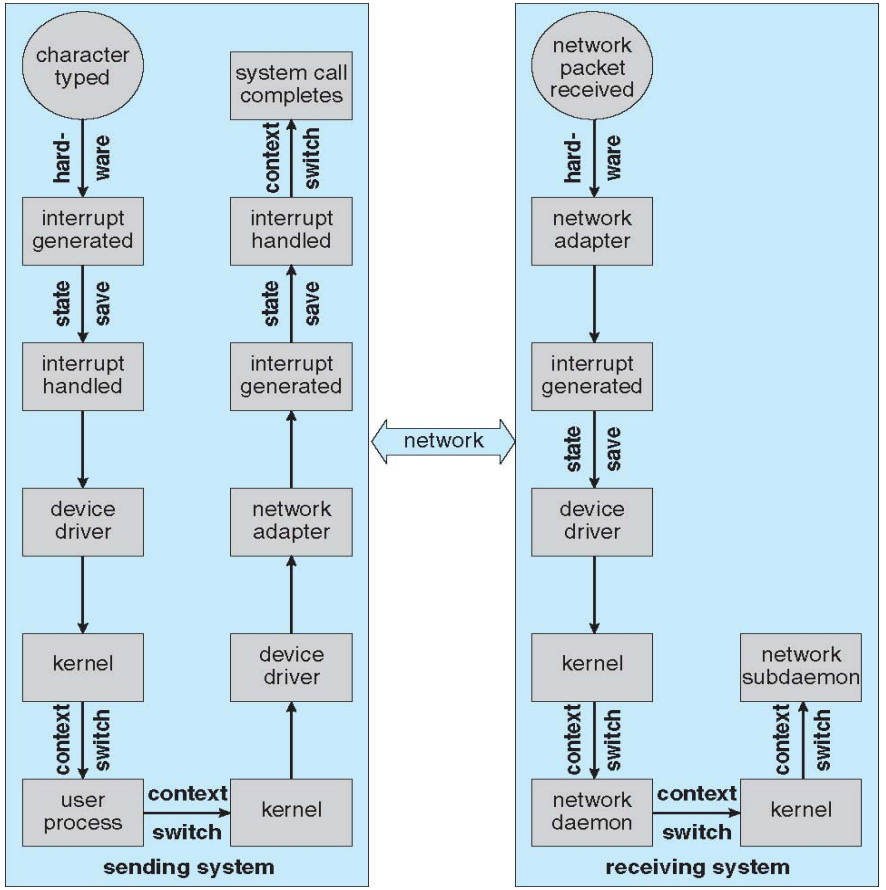
STREAMS (to read only)

- **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



Performance

- I/O a major factor in system performance:
  - Demands CPU to execute device driver code, kernel I/O code, schedule processes
  - Context switches due to interrupts stress CPU
  - Data copying loads memory bus
  - Network traffic especially stressful, since each interrupt must go from one machine to the other one through the network, and often returns to “echo” the proper result
- OS can handle thousands of interrupts per second, but interrupts are expensive tasks
  - changes the state
  - executes interrupt handler
  - restores state
- Programmed I/O can be more efficient than interrupt-driven I/O



Improving Performance

- Reduce number of context switches
- Reduce data copying in memory while passing between device and application
- Reduce interrupts by using large transfers, smart controllers, polling (minimize busy waiting)
- Use DMA controllers to offload data copying from CPU
- Use smarter hardware devices to be concurrent with CPU and bus operations
- Balance CPU, memory, bus, and I/O performance for highest throughput between each other
- Move user-mode processes / daemons to kernel threads

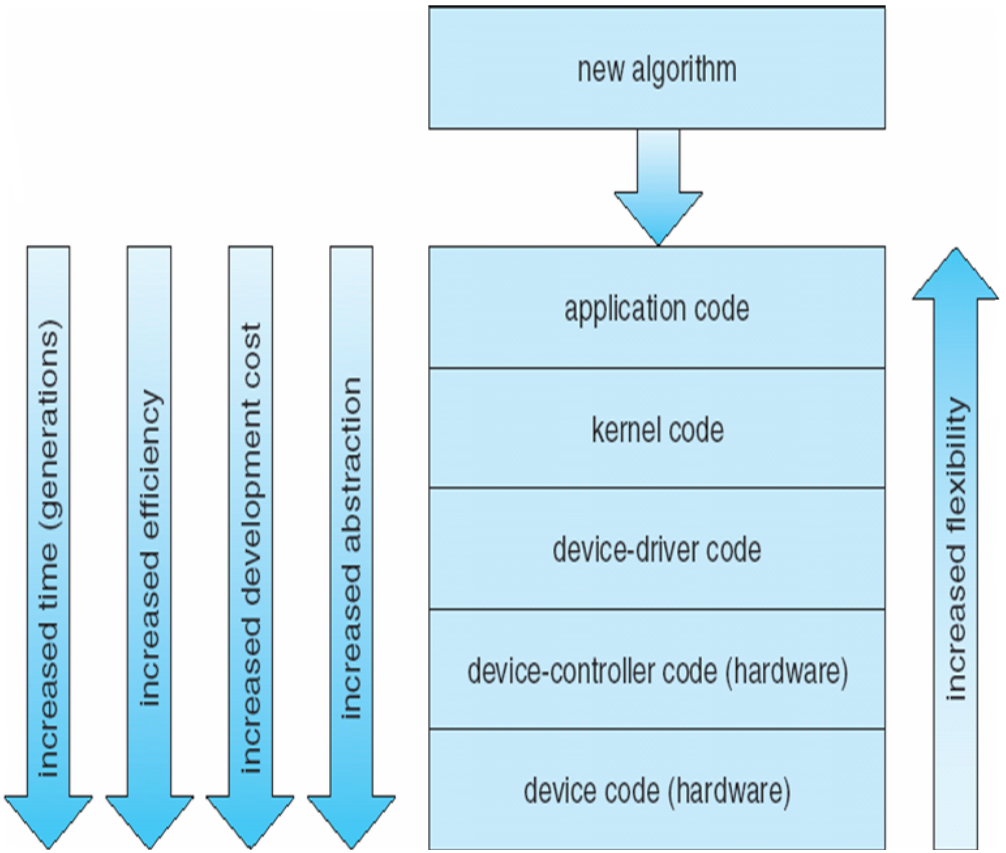
Device-Functionality Progression

Where to implement I/O functionality?

At application level:
+ flexible
+ bugs will not crash system
+ no need to reboot/reload drivers
- inefficient
  - context switches
  - does not exploit kernel data structures and functionalities

At kernel level:
+ improves tested performance
- more challenging in complex OS code
- must be thoroughly debugged

At hardware level (device or controller):
+ highest performance
- cost of improvements/fixing bugs
- requires more development time
- less flexible



End of Chapter 13