CISC 3320
I/O Subsystem

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

- These slides are a revision of the slides provided by the authors of the textbook via the publisher of the textbook

Outline

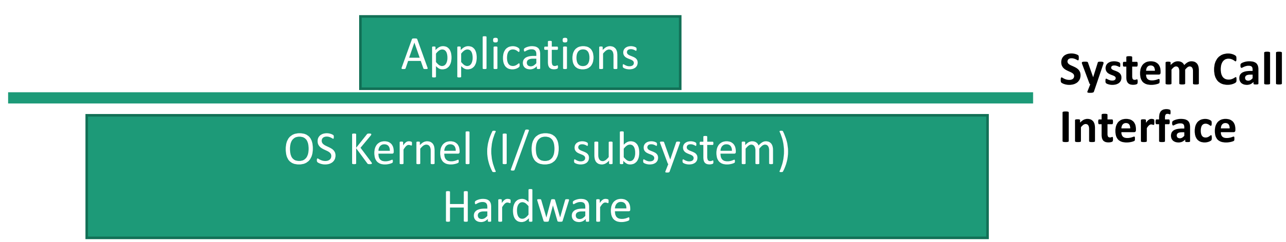
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- Performance

Application I/O Interface

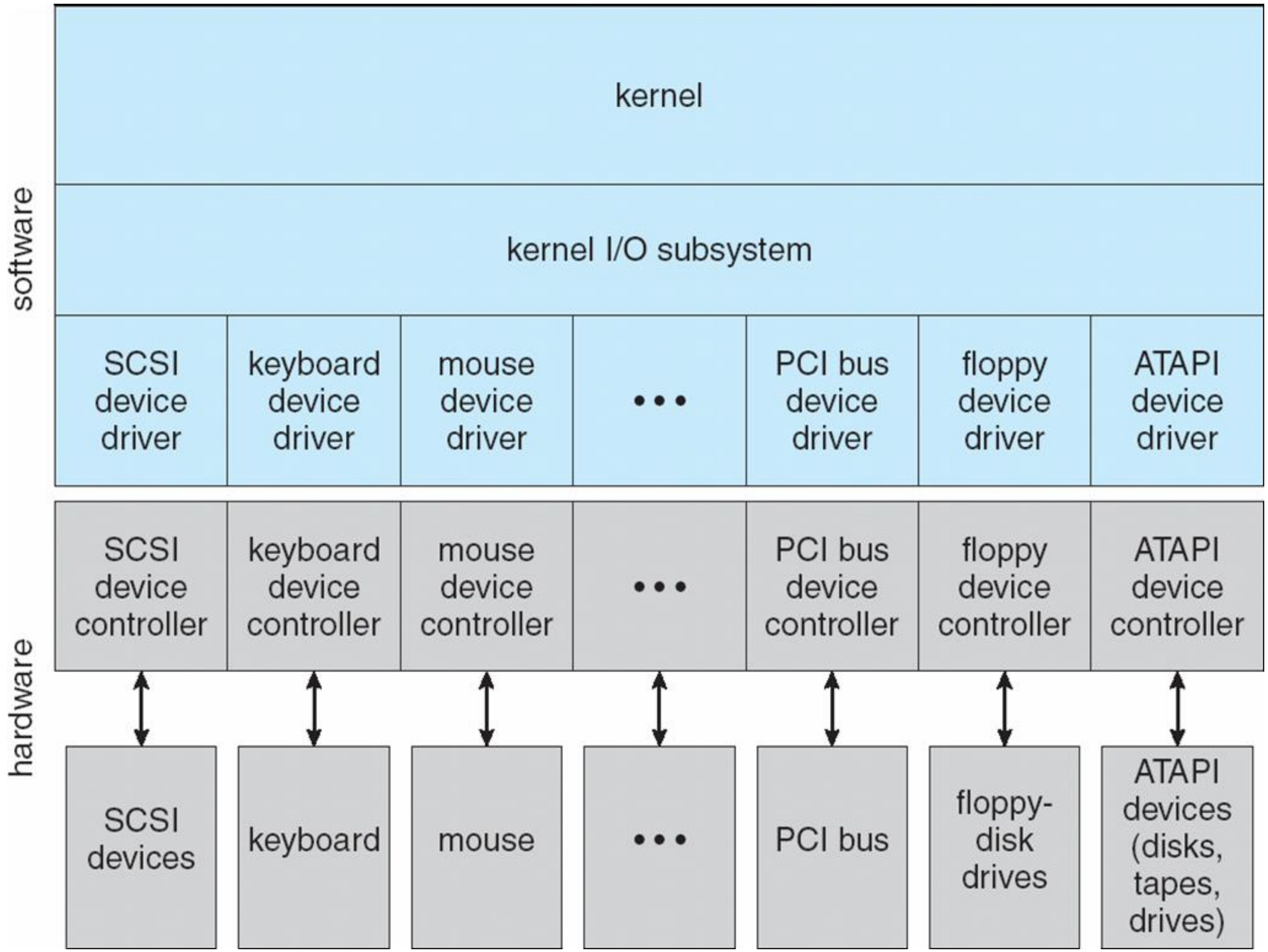
- I/O system calls encapsulate device behaviors in generic classes
- Examples: the programs are almost identical when you
  - write to a file, or
  - write to a terminal

Device Driver and I/O Subsystem

- Each OS has its own I/O subsystem structures and device driver frameworks
- Device independent
  - Device-driver layer hides differences among I/O controllers from the kernel
  - New devices talking already-implemented protocols need no extra work



A Kernel I/O Structure



Devices Vary

- Need to understand general characteristics to achieve device independent
- Characterize them along a couple of dimensions
  - Size of transfer: Character-stream or block
  - Access order: sequential or random access
  - Predictability and responsiveness: Synchronous and asynchronous
  - Shared or dedicated
  - Speed of operation, e.g., latency, seek time, transfer rate
  - Read-write, read only, or write only
  - Questions: what are example devices for each?

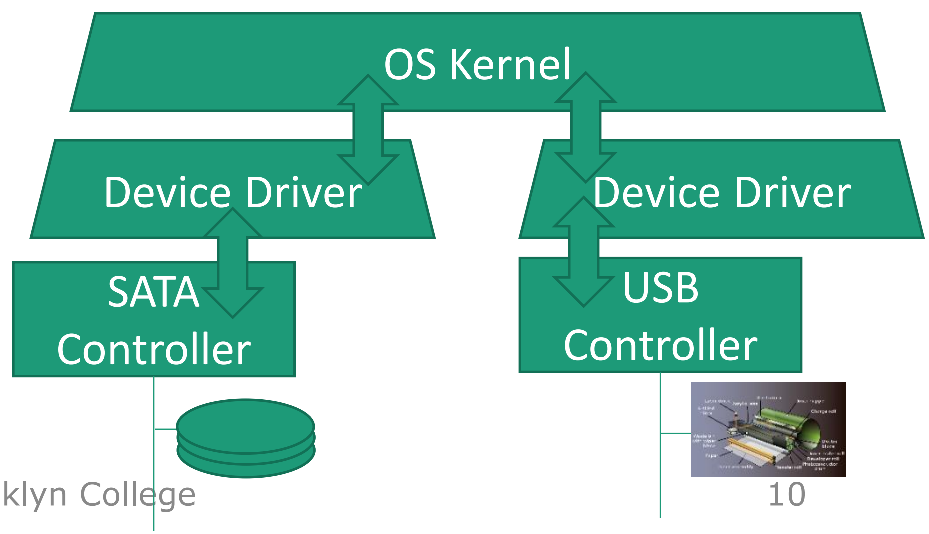
aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction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

Questions?

- I/O subsystem structure?
  - Application, kernel, I/O subsystem (device driver), device controller and hardware
  - Kernel provides I/O services to applications
- I/O system call examples?
- Different types of devices
  - Why do we need to characterize I/O devices? How do we characterize these devices?

Need to Categorizing I/O Devices

- Subtleties of devices handled by device drivers
- Applications and Kernel want to communicate with drivers in very limited but well defined fashion



Categorizing I/O Devices

- Broadly I/O devices can be grouped by the OS into
  - Block I/O
  - Character I/O (Stream)
  - Memory-mapped file access
  - Network sockets
- Thus, common functionalities can be defined.
- For direct manipulation of I/O device specific characteristics, usually via an escape / back door
  - Example: UNIX **`ioctl()`** call to send arbitrary bits to a device control register and data to device data register

Questions?

- The need to categorize different I/O devices

Block 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

Block Devices: Examples

- Naming
  - Examples on Linux
    - by label, by uuid, by id, and by path
    - Running examples
      - lsblk –f
      - ls /dev/disk/
- Read and write a block a time
- Essential behavior
  - read(), write()
  - For random-access block devices
    - seek()

Character Devices

- Character devices include keyboards, mice, serial ports
- Read and write a character a time
- Essential behavior
  - 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)

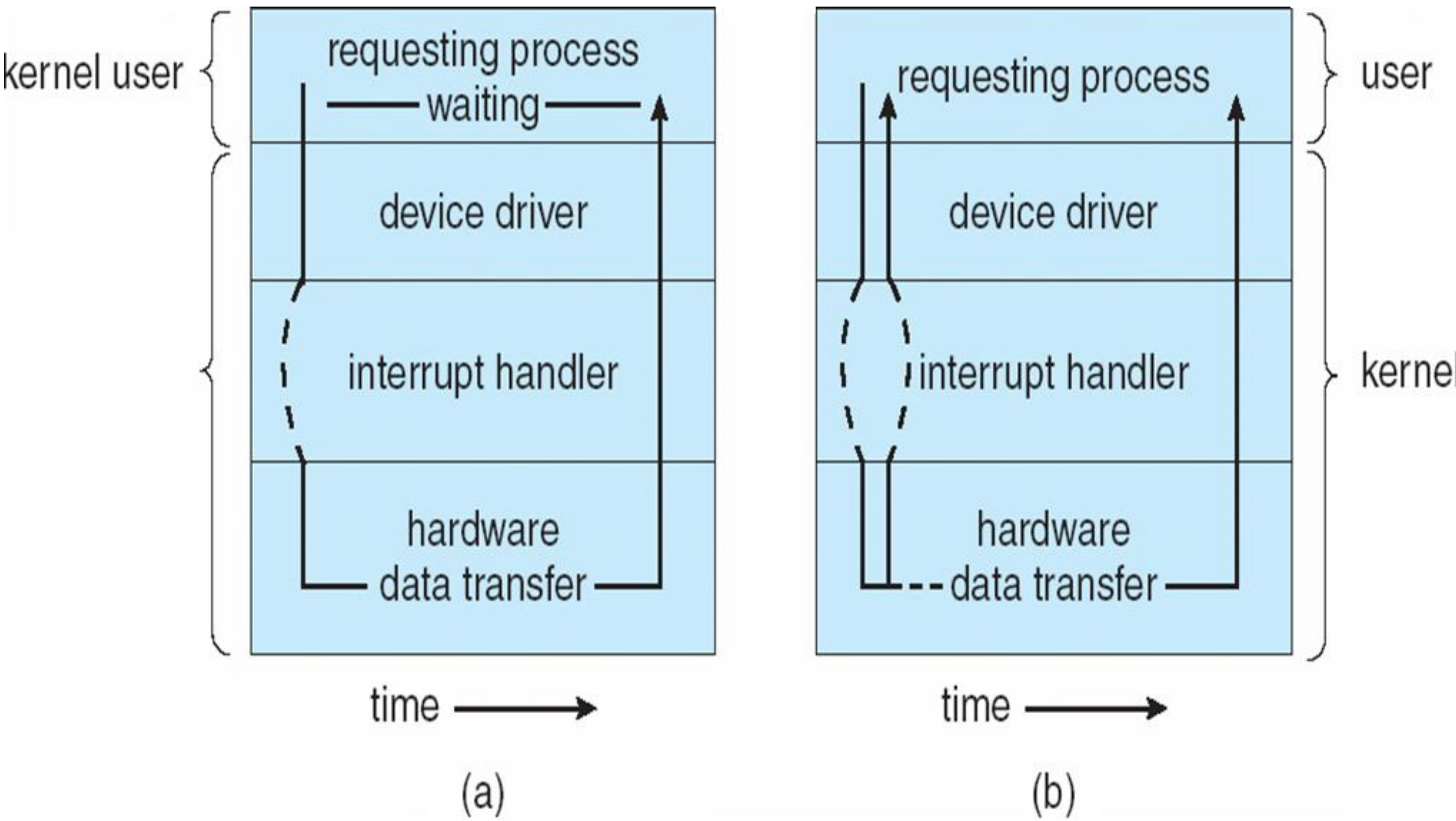
Clocks and Timers

- 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



Synchronous

Asynchronous

Vectored I/O

- Vectored I/O allows one system call to perform multiple I/O operations
- For example, Unix **`readve()`**/Linux **`readv()`** 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

Questions

- Different types of I/O devices and examples
- Related system calls?

Kernel I/O Subsystem: Scheduling

- Some I/O request ordering via per-device queue
- Some operating systems try fairness
- Some implement Quality Of Service (i.e. IPQOS)

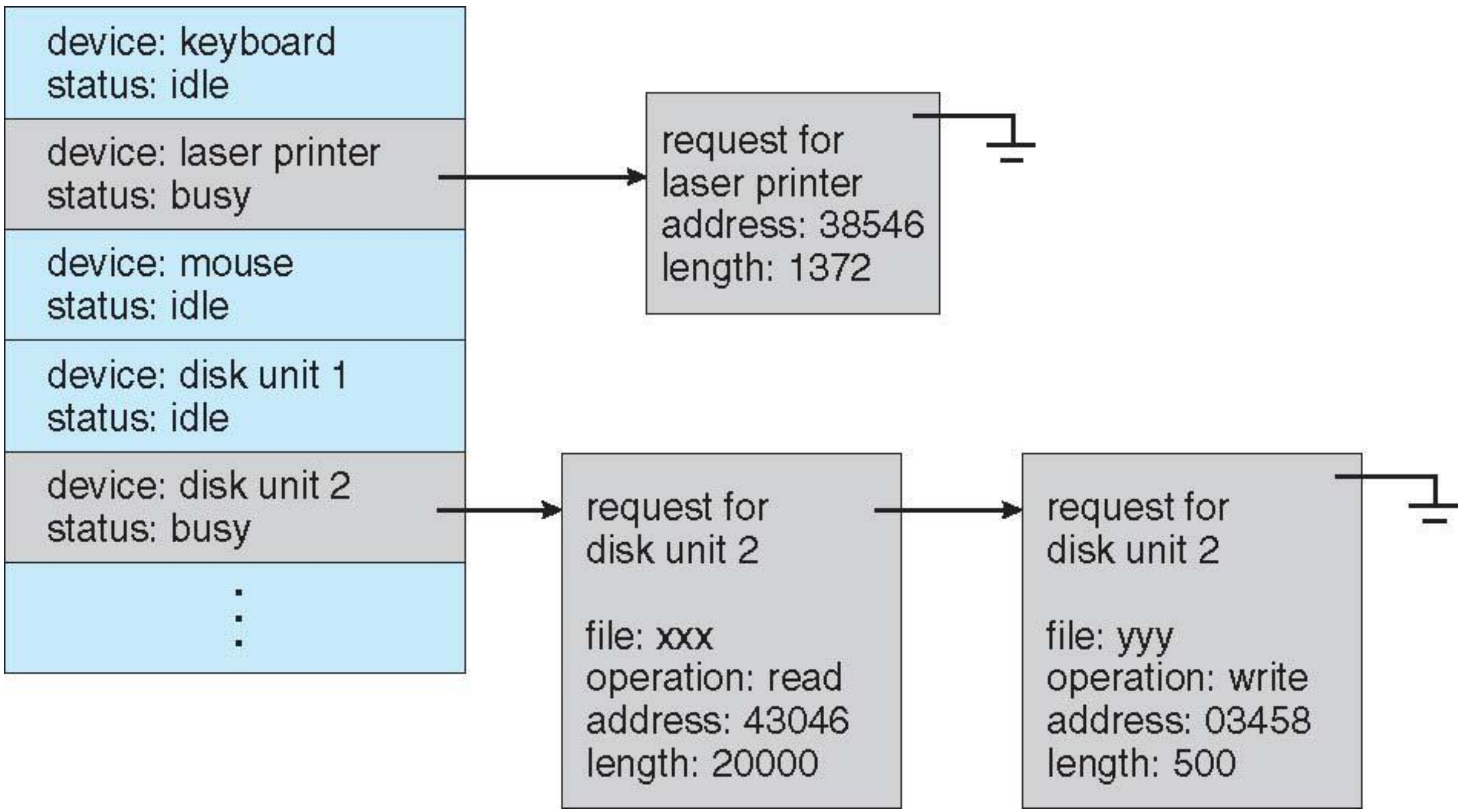
Kernel I/O Subsystem: Buffering

- 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

Questions

- I/O subsystem design issues
  - Scheduling
  - buffering

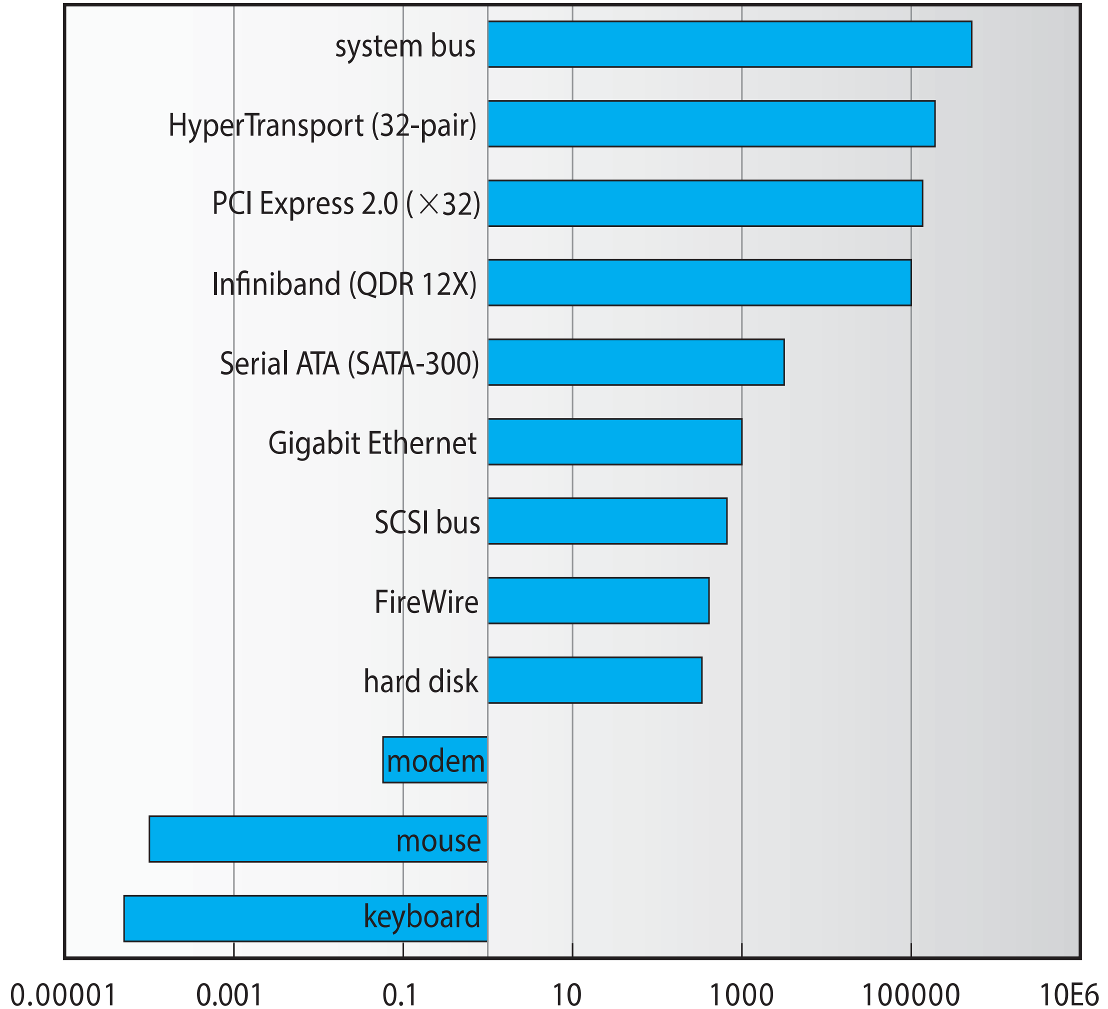
Device Status Table



Questions?

- Data structures in I/O subsystem

Sun Enterprise 6000 Device-Transfer Rates



Question for Designers

- How do we deal with (or take advantage of) the disparity of the vastly different transfer rates?
- What if a device can only fulfill one request at a time?

Caching

- Faster device holding copy of data
  - Always just a copy
  - Key to performance
  - Sometimes combined with buffering

Spooling

- 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 (to be discussed in the class)

Questions?

- I/O Transfer rate?
- Concurrent access to I/O devices?

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

Questions?

- Dealing with errors?

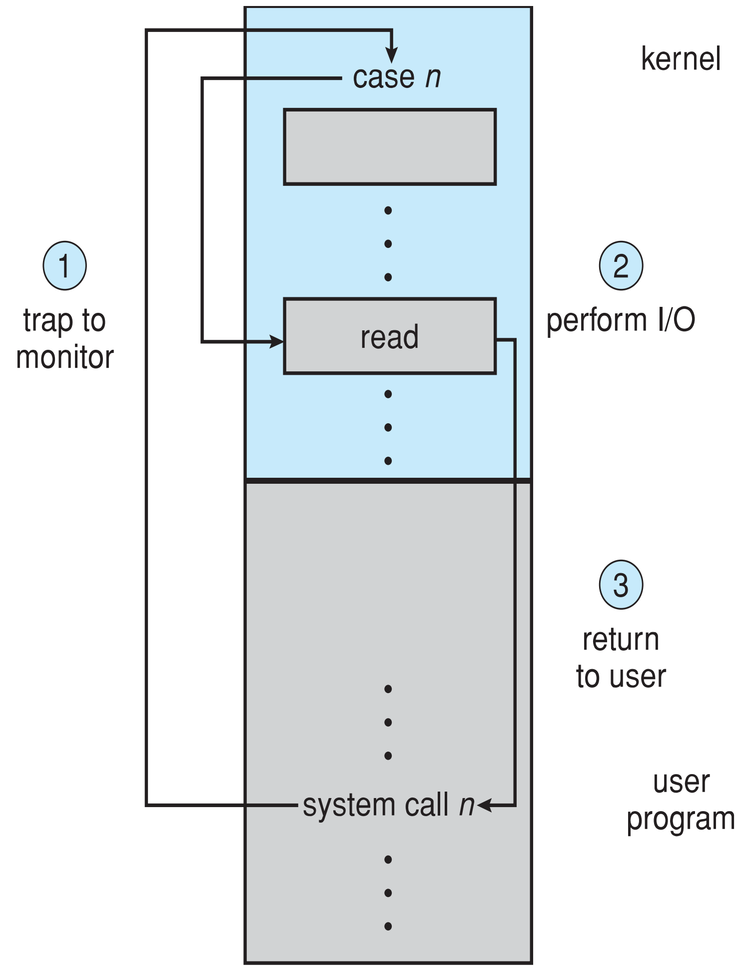
Need for I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions

I/O Protection

- 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



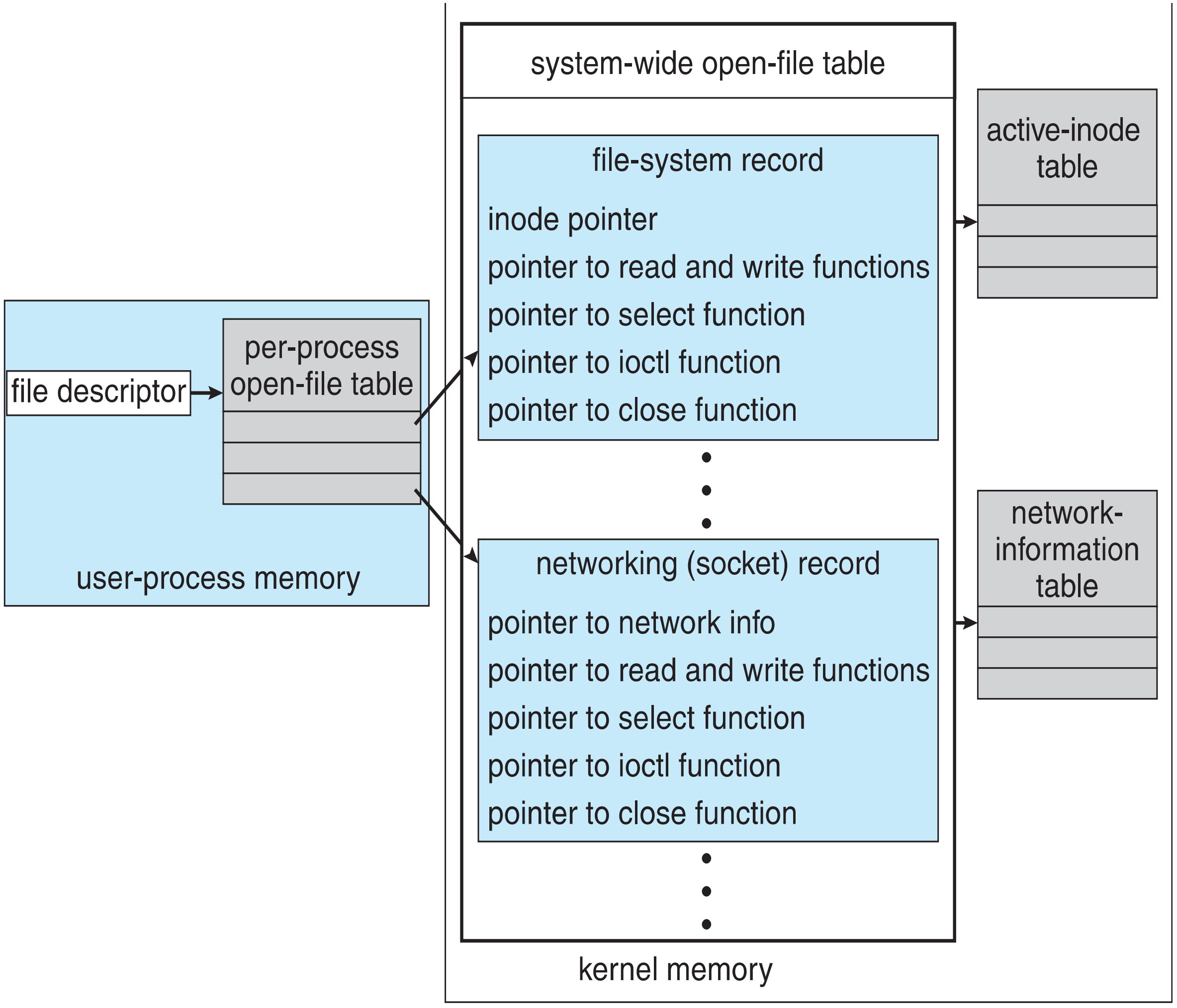
Questions?

- Design for I/O protection

Kernel Data Structures for I/O

- 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



Questions?

- Kernel data structures for I/O

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: Examples

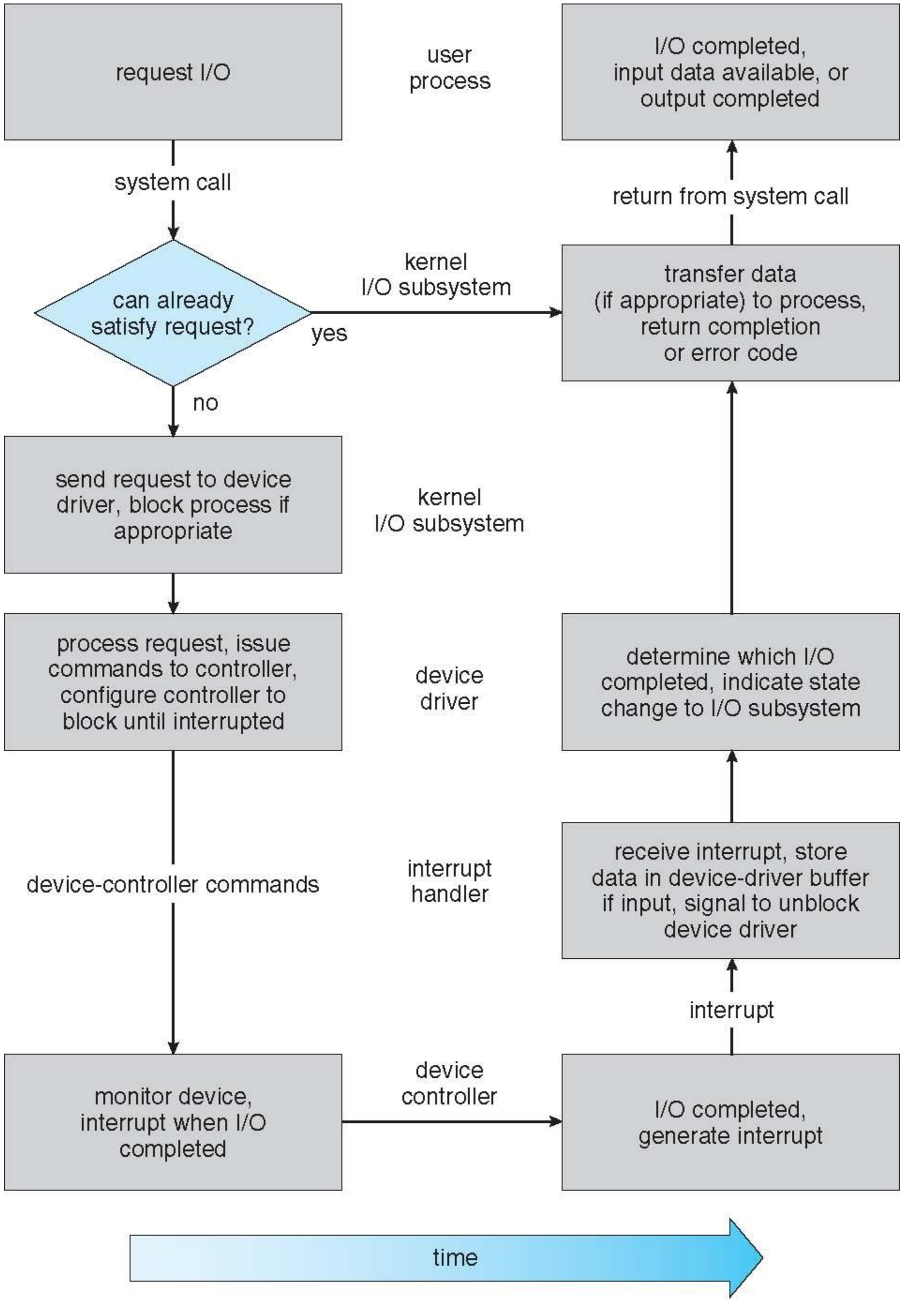
- For example, Android implements
  - Component-level power management
    - Understands relationship between components
    - Build device tree representing physical device topology
    - System bus -> I/O subsystem -> {flash, USB storage}
    - Device driver tracks state of device, whether in use
    - 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)

Questions?

- I/O and power management?

Life Cycle of An I/O Request

- 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


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

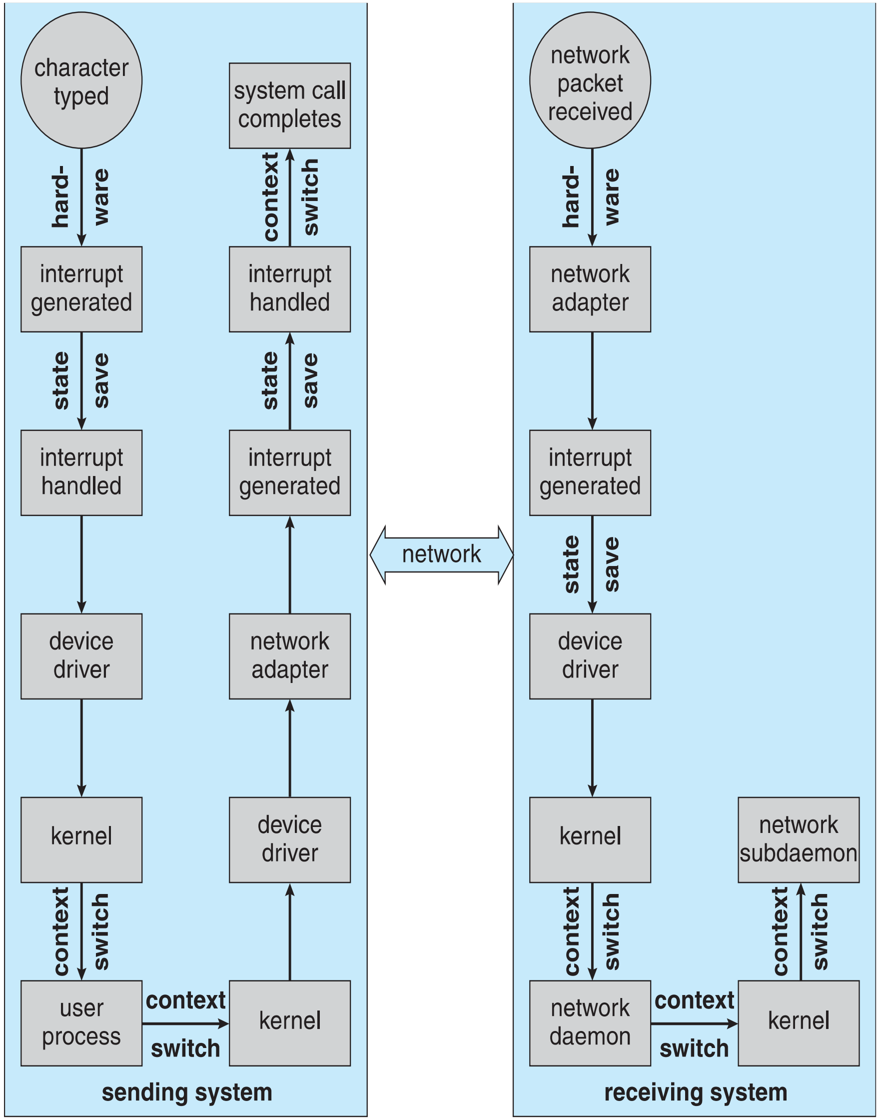
Questions?

- I/O request operation from beginning to end (life cycle)?

I/O and 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

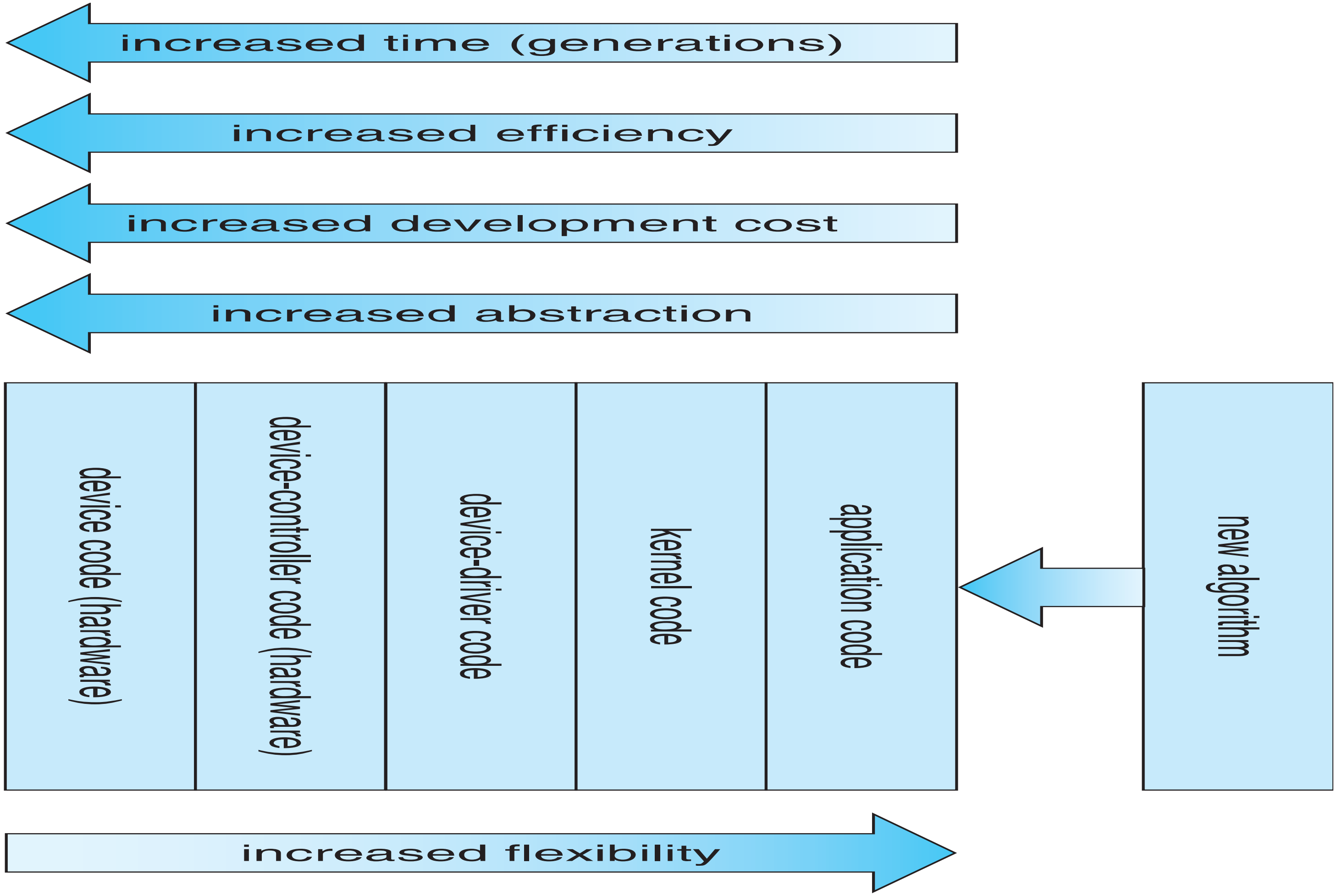
IPC Consideration



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



Design Consideration: Access Right

- A design consideration
  - What kind of access right should we give to device drivers?
  - Unrestricted
    - Kernel mode
    - Relatively easier to design, can affect the others
  - Restricted
    - User mode
    - More difficult to design, isolated from the others

Design Consideration: Loading Device Drivers

- Relink the kernel with the new deriver
  - Require reboot
- Add to the kernel an entry indicating a new driver is needed
  - Load the driver during reboot
- Install and run the device driver on the fly
  - Hot-pluggable

Questions?

- System performance and I/O
- Device-function progression
- Access right
- Loading device drivers