OPERATING SYSTEMS

I /O Systems

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

Contents

Overview

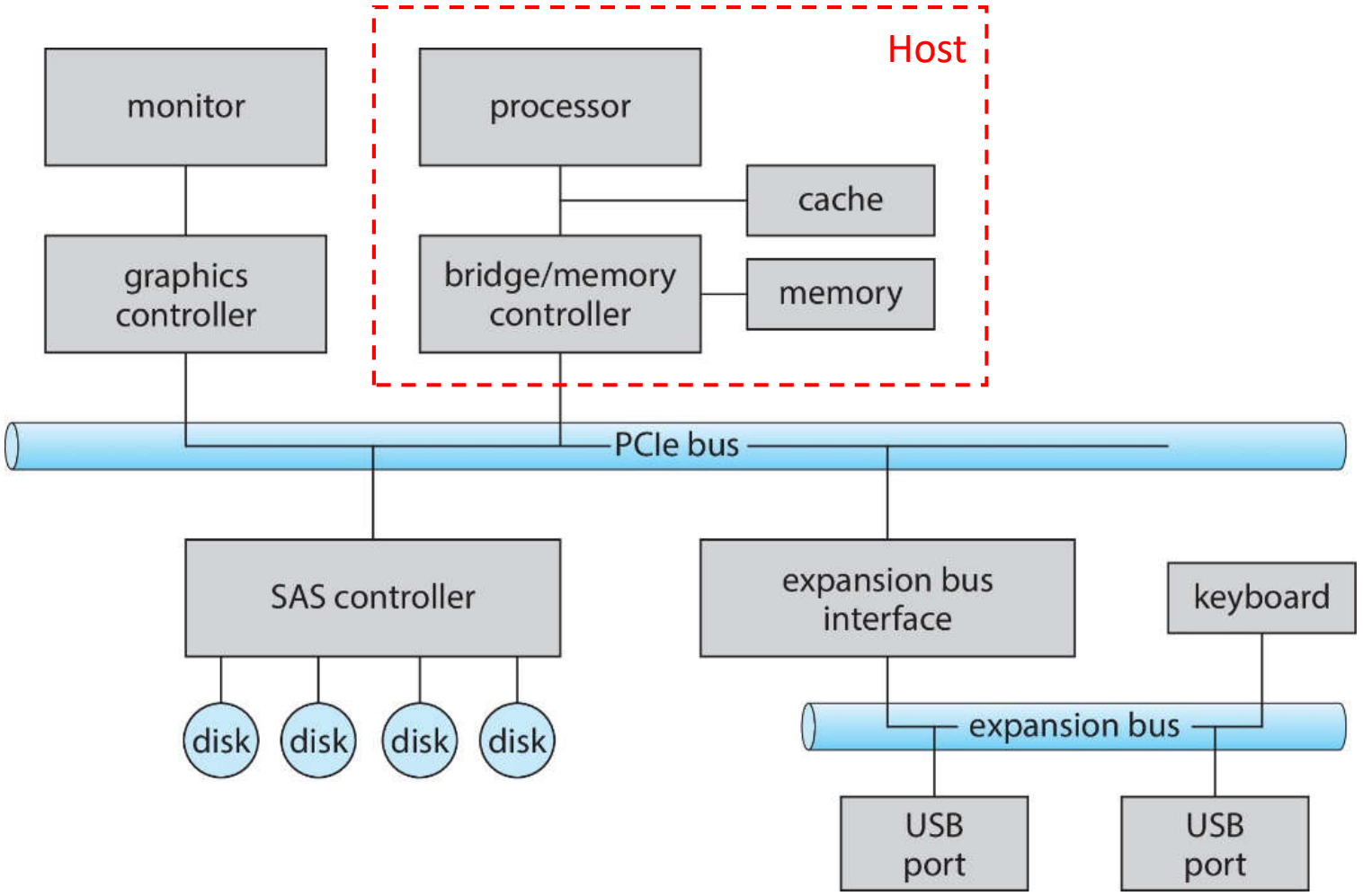
- The two main jobs of a computer are I/O and computing.
  - In many cases, the main job is I/O, and the computing or processing is merely incidental.
- The role of the operating system in computer I/O is to manage and control I/O devices and I/O operations. The control of devices connected to the computer is a major concern of OS designers.
  - I/O devices vary widely and varied methods are needed to control them.
  - These methods form the I/O subsystem of the kernel, which separates the rest of the kernel from the complexities of managing I/O devices.
- The basic I/O hardware elements, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices.
- Device-driver modules are used to encapsulate the details and oddities of different devices.
  - They present a uniform device access interface to the I/O subsystem.

I/O Hardware

- Incredible variety of I/O devices for:
  - Storage
  - Transmission
  - Human-interface
- A device communicates with a computer system (host) by sending signals over a cable or even through the air.
  - *Port* – A port is a connection point for the communication.
  - *Bus* – A bus is a common set of wires shared by devices with a rigidly defined protocol that specifies a set of messages that can be sent on the wires.
    - *Daisy Chain* (雏菊花链) or shared direct access
    - *PCI* (Peripheral Component Interconnect) bus common in PCs and servers
    - *PCIe* (PCI Express) bus
    - *Expansion Bus* connects relatively slow devices
  - *Controller* (Host Adapter) – A controller is a collection of electronics that can operate a port, a bus, or a device.
    - e.g., Disk controller and FC controller

I/O Hardware

- A Typical PC Bus Structure.



I/O Device-Control Registers

- I/O devices are controlled by I/O instructions. 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: read 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. These bits indicate states, such as ending of command, data ready, device error, etc.
  - control register: can be written by the host to start a command or to change the mode of a device.
    - For instance, full-duplex/half-duplex, parity checking, speed selection, etc.
- The data registers are typically 1 to 4 bytes in size. Some controllers have FIFO chips that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register. A FIFO chip can hold a small burst of data until the device or host is able to receive those data.

Memory Mapped I/O (MMIO)

- Memory-Mapping
  - The device-control registers are mapped into the address space of the processor (physical memory).
  - 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.
- Example: DriectDraw.
  - The graphics controller has I/O ports for basic control operations and a large memory-mapped region to hold screen contents. A thread sends output to the screen by writing data into the memory-mapped region. The controller generates the screen image based on the contents of this memory. This technique is simple to use. Moreover, writing millions of bytes to the graphics memory is faster than issuing millions of I/O instructions.
- Today, most I/O is performed by device controllers using memory-mapped I/O.

Device I/O Port

- 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)

Polling

- In polling (轮询) scheme, the host writes output through a port, coordinating with the controller by handshaking as follows:
  (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.
- Step (1) is busy-waiting or polling to wait for I/O from device
  - Reasonable if device is fast, but inefficient if device slow.
  - CPU switches to other tasks?
    - If a cycle was missed, data would be overwritten or lost.

Interrupts

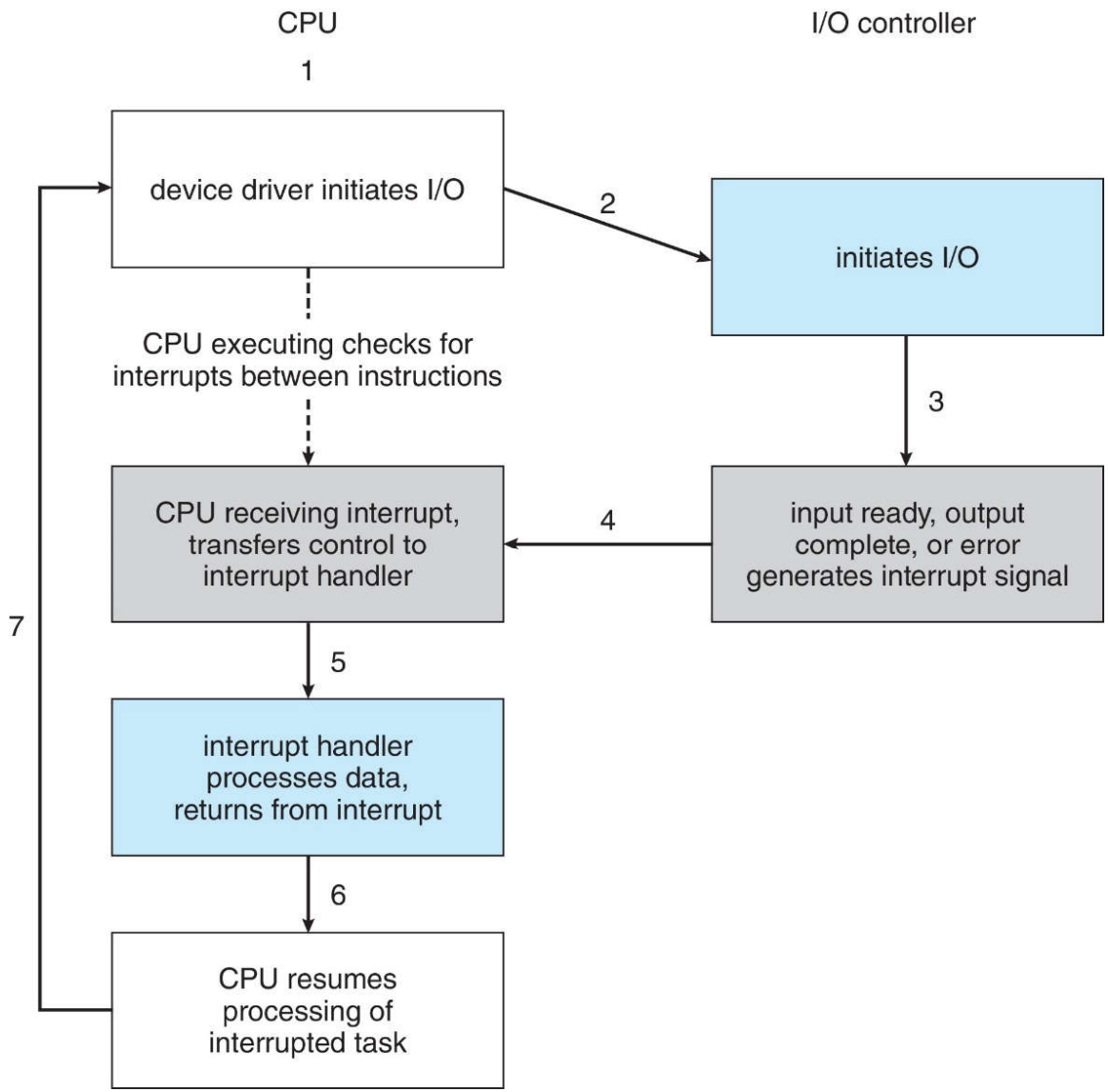
- Polling can happen in 3 instruction cycles.
  (1) Read a device register
  (2) logical-and to extract status bit
  (3) branch if not zero.
  - How to be more efficient if non-zero infrequently (always busy)?
- CPU has an *Interrupt-request line* triggered by I/O device.
  - The line is checked by processor after each instruction.
- *Interrupt handler routine* receives interrupts.
  - *Maskable* to ignore or delay some interrupts
- *Interrupt vector* is used 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.

Interrupts

- 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-Driven I/O Cycle.

Interrupts

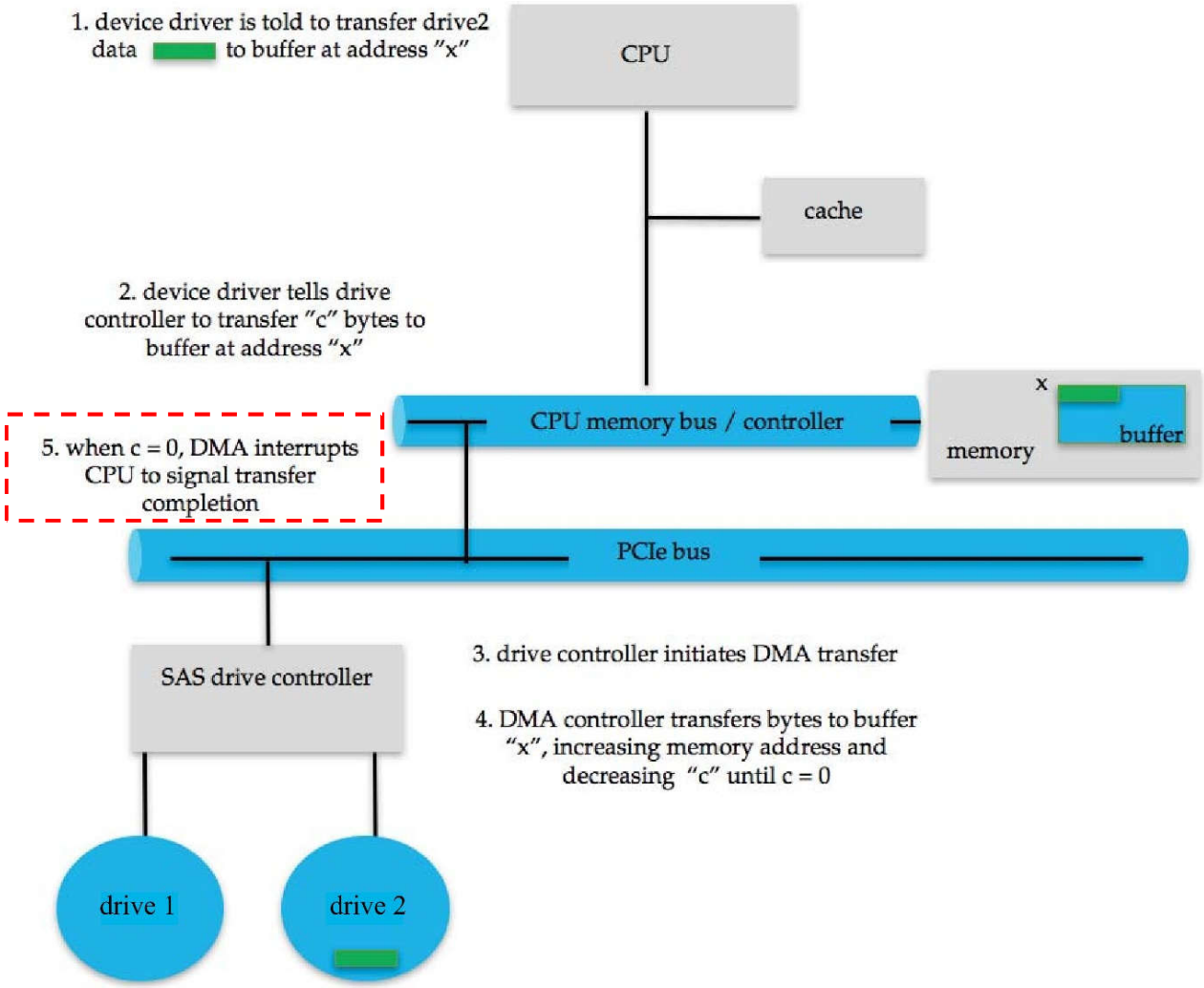
- Interrupt mechanism is also used for *exceptions.*
  - dividing by zero
  - accessing a protected or nonexistent memory address
  - attempting to execute a privileged instruction from user mode
  - hardware error
  - etc.
- A page fault is also an exception that raises an interrupt (No.14).
  - The interrupt suspends the current process and jumps to the page-fault handler in the kernel.
- Trap (software interrupt)
  - System call executes via *trap* to trigger kernel to execute request.
  - The trap is given a relatively low interrupt priority compared with those assigned to device interrupts.
- Multi-CPU systems can process interrupts concurrently.
  - If operating system designed to handle it.
- Question: Discuss the difference among interrupt, fault, trap and exception.

Direct Memory Access (DMA)

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

Direct Memory Access

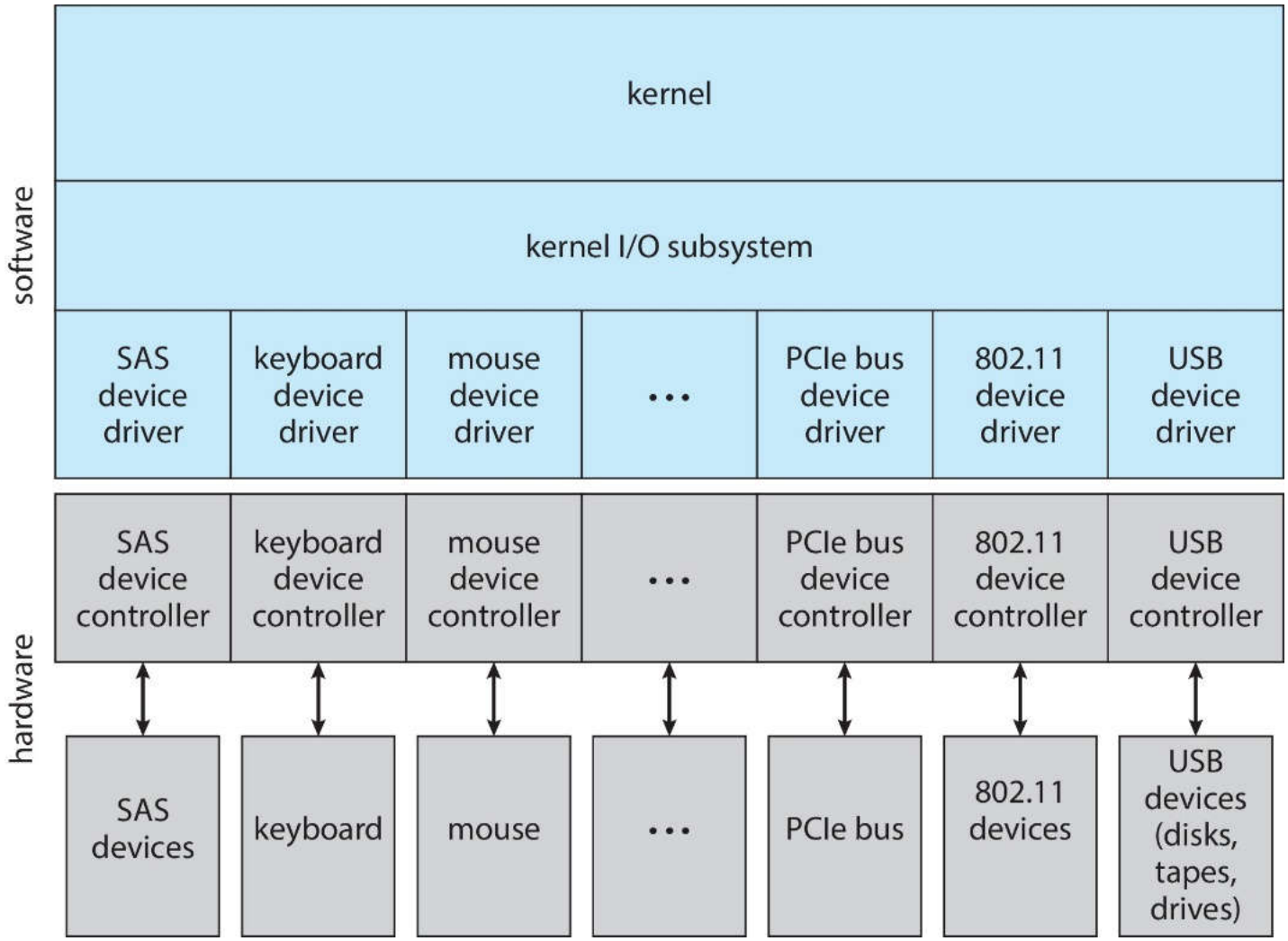
- Steps to Perform a DMA Transfer.



Application I/O Interface

- Each OS has its own I/O subsystem structures and device driver frameworks. The detailed differences in I/O devices are abstracted away by identifying a few general kinds. Each general kind is accessed through a standardized set of functions—an *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.
- Making the I/O subsystem independent of the hardware simplifies the job of the operating-system developer. It also benefits the hardware manufacturers.

Application I/O Interface



A Kernel I/O Structure

Application I/O Interface

- I/O Devices vary in many dimensions.
  - Data transfer mode
    - character-stream or block
  - Access mode
    - sequential or random-access
  - Transfer schedule
    - synchronous or asynchronous (or both)
  - Sharing
    - sharable or dedicated
  - Speed of operation
    - latency, seek time, transfer rate, delay, etc.
  - I/O direction
    - read-write, read only, or write only.

Application I/O Interface

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

Application I/O Interface

- Characteristics of I/O Devices
  - Contiguous Allocation
  - Subtleties of devices handled by device drivers (由设备驱动器处理设备的细节)
  - Broadly I/O devices can be grouped by the OS into (I/O 设备可以由操作系统大致分为):
    - Block I/O
    - Character I/O (Stream I/O)
    - Memory-mapped file access
    - Network sockets
  - For direct manipulation specific characteristics of I/O devices, usually an escape (or back door) is used to transparently passes arbitrary commands from an application to a device driver.
    - Unix ioctl() call is used to send arbitrary bits to a device control register and data to the 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 is 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

- Network device vary enough from block and character to have their 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

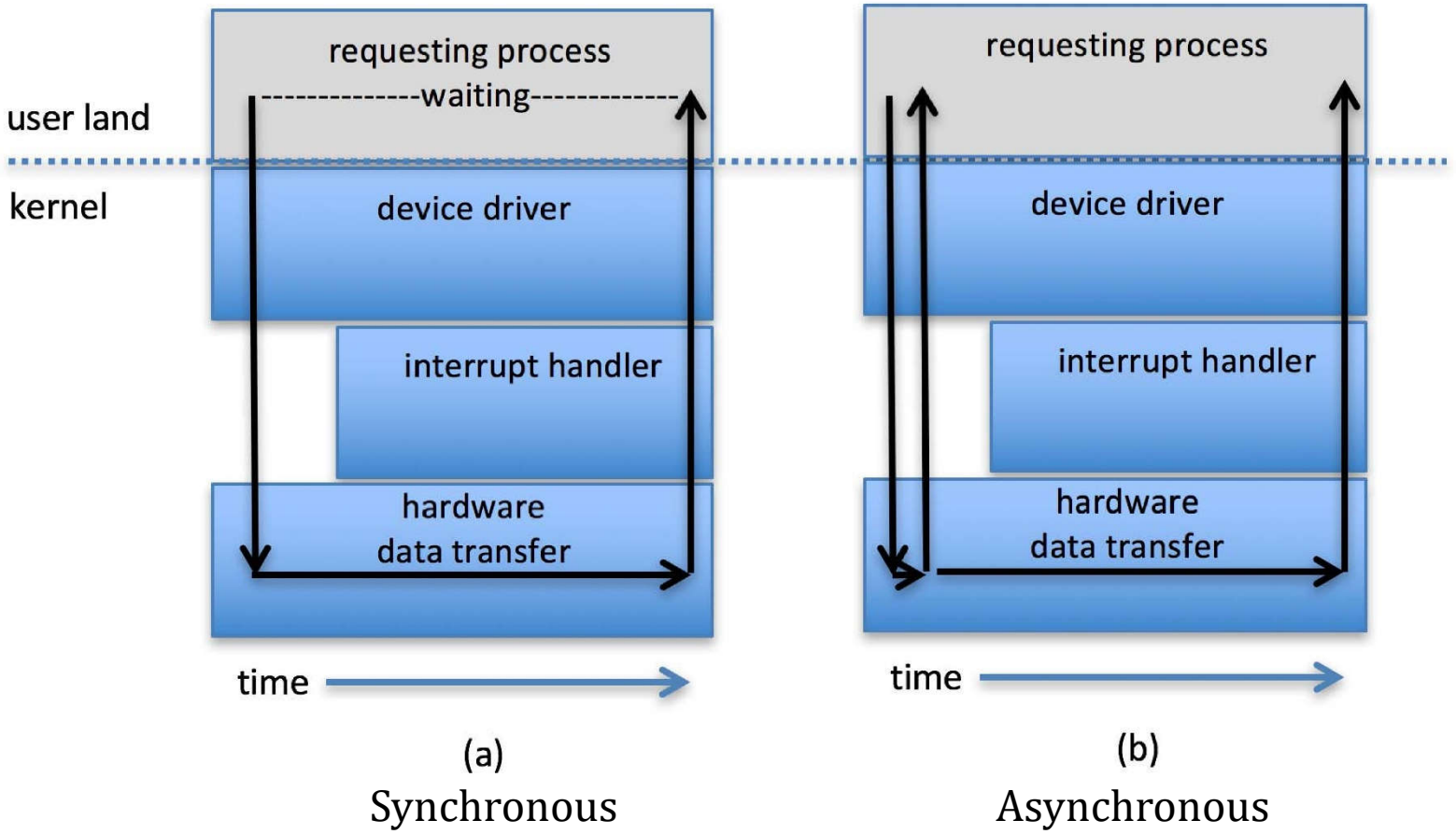
- Provide current time, elapsed time, timer
- Normal resolution about 1/60 second
  - Some systems provide higher-resolution timers.
- *Programmable interval timer* used for timings, periodic interrupts
- ioctl() on UNIX covers odd aspects of I/O such as clocks and timers.

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

Nonblocking and Asynchronous I/O

- Two I/O Methods - Synchronous and Asynchronous.



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 and either reads from a source to that vector or writes from that vector to a destination.
- This scatter-gather method may be better than multiple individual I/O system calls.
  - It can decrease context switching and system call overhead.
  - Some versions provide atomicity.
    - assuring that all the I/O is done without interruption
    - avoiding corruption of data if other threads are also performing I/O involving those buffers.

Kernel I/O Subsystem

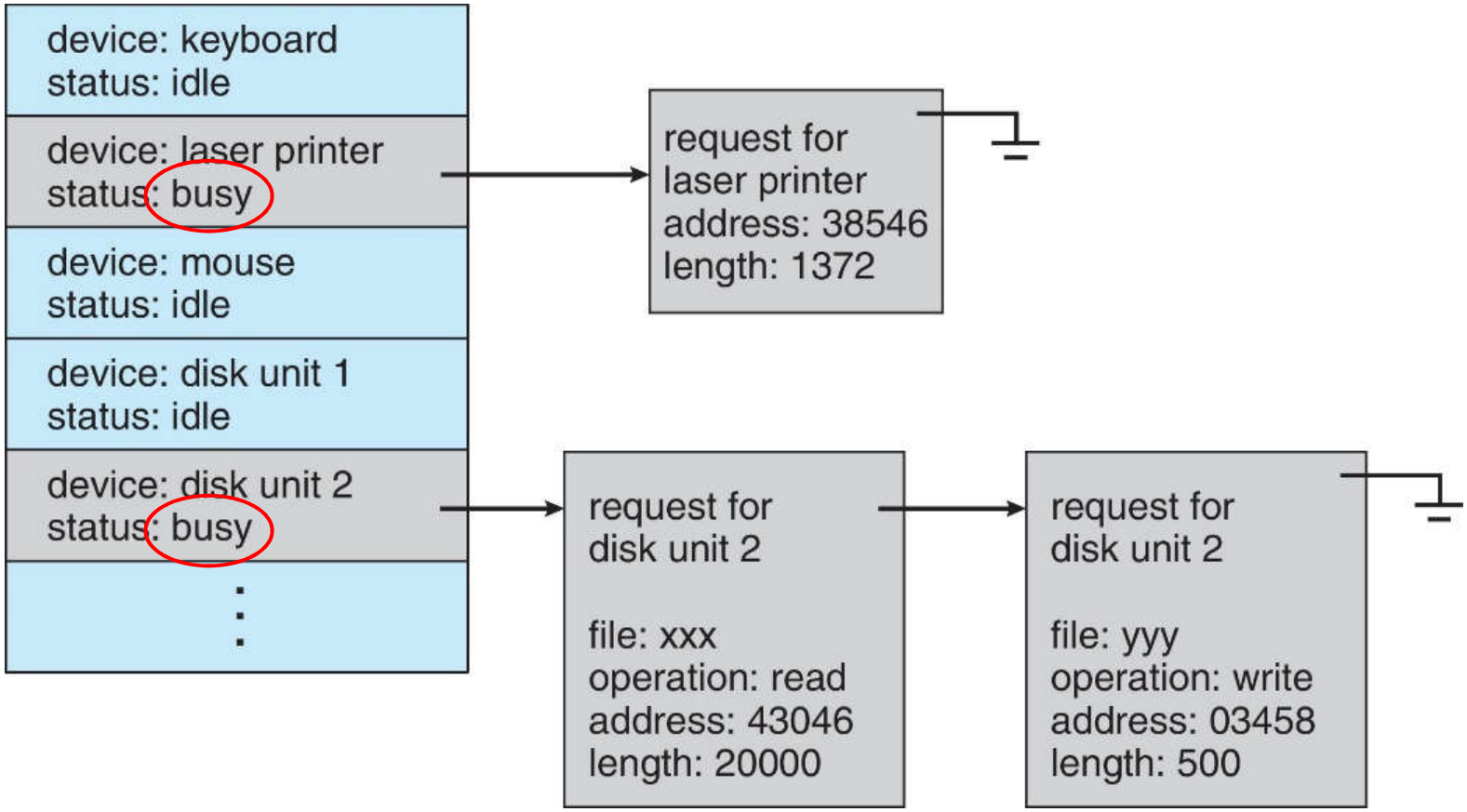
- kernel's I/O subsystem provides several services which are built on the hardware and device driver infrastructure.
  - Scheduling
  - Buffering
  - Caching
  - SPOOLing
  - Device reservation
  - Error handling.
- The I/O subsystem is also responsible for protecting itself from errant processes and malicious users.

I/O Scheduling

- The I/O scheduler rearranges the order of a wait queue of requests for each device.
  - to improve overall system performance
  - to share device access fairly among processes
  - to reduce the average waiting time for I/O to complete.
- *Device-status table*
  - Device-status table contains an entry for each I/O device indicating the device's type, address, and state.
    - A device is in one of the three status
      - not functioning
      - Idle
      - busy.
    - If the device is busy with a request, the information of the request will be declared.
  - The *kernel* manages this table, supports asynchronous I/O, keeps track all the I/O requests at the same time, scheduling I/O operations.

I/O Scheduling

- Device-status Table.

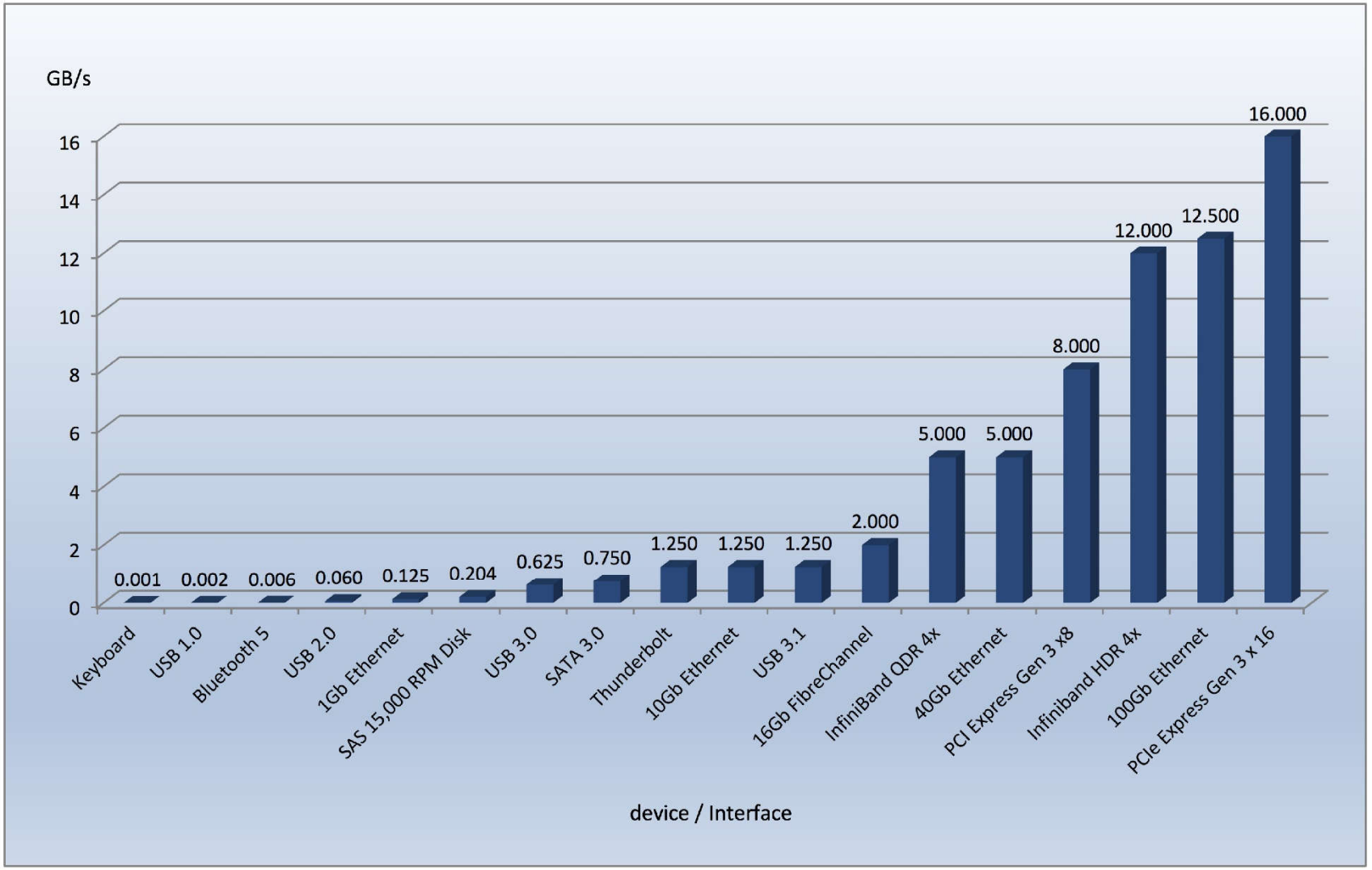


Buffering

- *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”
- *Copy semantics* - the OS guarantees that the version of the data written to disk is the version *at the time of* the application system call, independent of any subsequent changes in the application’s buffer.
  - For example, for the write() system call, the OS copies the application data into a kernel buffer before returning control to the application. The disk write is performed from the kernel buffer, so that subsequent changes to the application buffer have no effect.
- *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.

Buffering

- Common PC and data-center I/O device and interface speeds.



Caching

- A *cache* is a region of fast memory that holds copies of data.
  - Access to the cached copy is more efficient than access to the original.
- A buffer and a cache are different.
  - A buffer may hold the only existing copy of a data item, whereas a cache, by definition, holds a copy on faster storage of an item that resides elsewhere.
- *Caching and buffering* are distinct functions, but sometimes a region of memory can be used for both purposes.
  - When the kernel receives a file I/O request, the kernel first accesses the buffer cache to see whether that region of the file is already available in main memory. If it is, a physical disk I/O can be avoided or deferred.
  - Disk writes are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules.

Spooling and Device Reservation

- A *SPOOL* is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams.
  - Some devices, such as tape drives and printers, cannot usefully multiplex the I/O requests of multiple concurrent applications.
- *SPOOLing* is one way operating systems can coordinate such concurrent problem.
  - Each application's output is spooled to a separate secondary storage file.
  - When an application finishes printing, the SPOOLing system queues the corresponding spool file for output to the printer.
  - The OS provides a control interface that enables users and system administrators to display and maintain the spooling queue.
- *Device reservation* (设备预订) is another way to deal with concurrent device access by providing explicit facilities for coordination.
  - Applications should watch out for deadlock.

Error Handling

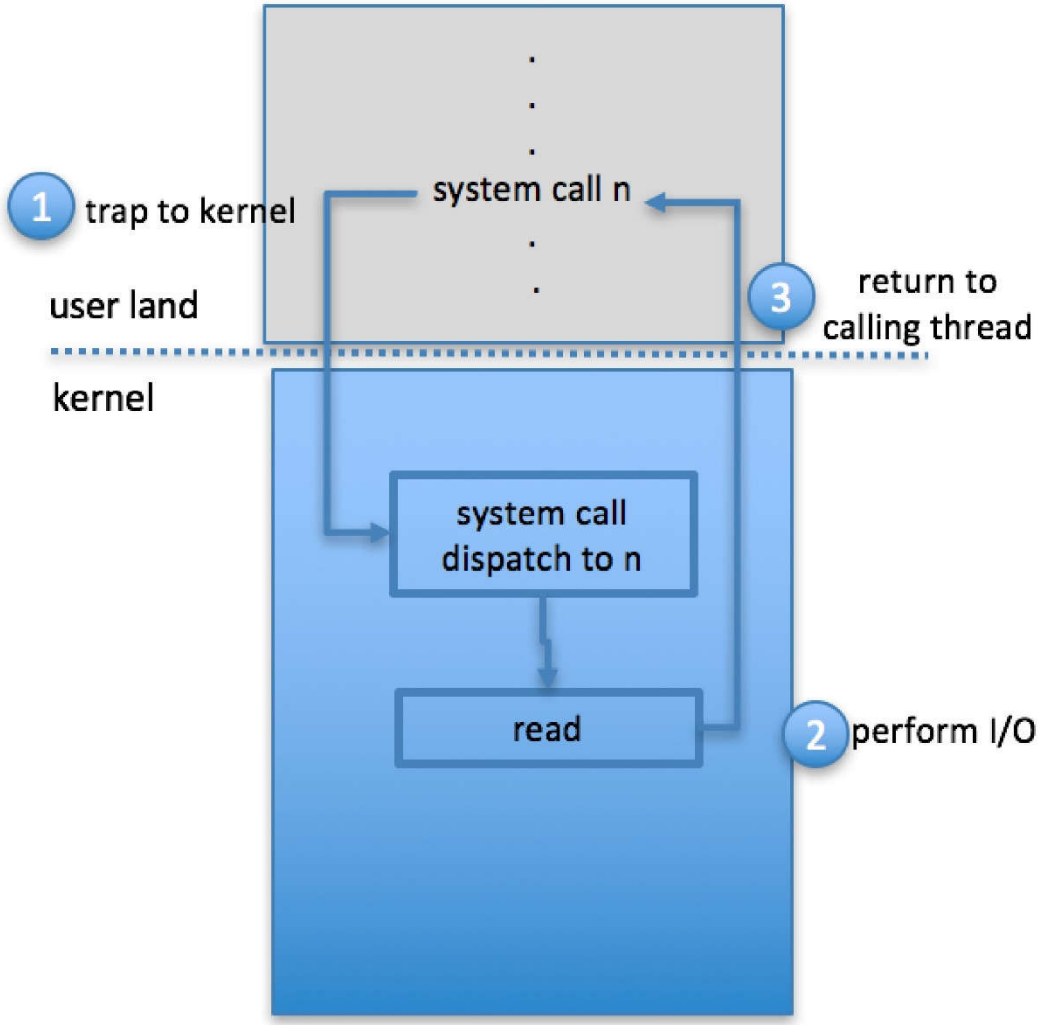
- Devices and I/O transfers can fail in many ways.
  - for transient (瞬时) reasons, as when a network becomes overloaded.
    - OS can often compensate effectively for transient failures by retrying the requests.
  - for permanent reasons, as when a disk controller becomes defective.
    - Unfortunately, if an important component experiences a permanent failure, the operating system is unlikely to recover.
- Generally an I/O system call will return information about the status of the call, signifying either success or failure.
  - E.g., errno in Linux.
- System error logs hold problem reports.

I/O Protection

- A user process may accidentally or purposely attempt to disrupt the normal operation of a system by attempting to issue illegal I/O instructions.
  - Errors are closely related to the issue of protection.
- *Protection*:
  - All I/O instructions are defined to be privileged instructions. A user program cannot issue I/O instructions directly, but execute a system call to request the OS to perform I/O on its behalf.
  - Any memory-mapped and I/O port memory locations must be protected from user access by the memory-protection system.

I/O Protection

- Use of a System Call to Perform I/O.

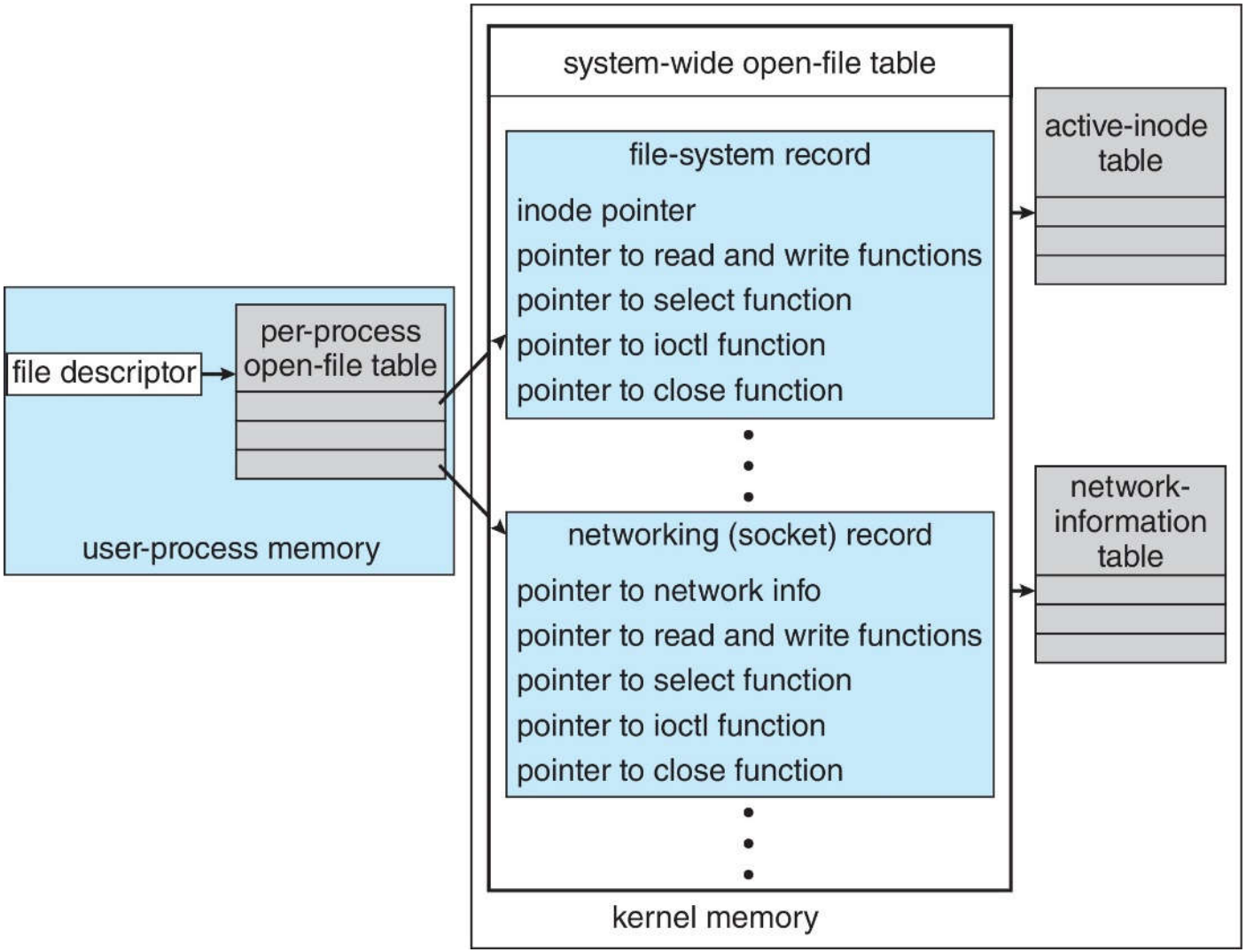


Kernel Data Structures

- The kernel keeps state information for I/O components, including open file tables, network connections, character device state, etc.
- Many complex data structures are used to track buffers, memory allocation, “dirty” blocks, etc.
- Some operating systems 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?

Kernel Data Structures

- UNIX I/O Kernel Structure.



Power Management

- Power management in general is based on device management.
  - At boot time, the firmware system analyzes the system hardware and creates a device tree in RAM to manage activities pertaining to devices, including hot-plug (热拔插), understanding and changing device states, and power management.
  - It is not strictly in the domain of I/O, but much is I/O related.
  - Operating systems can help management and improvement.
  - Mobile computing has power management as first class OS aspect.
- *Advanced Configuration and Power Interface (ACPI)*
  - ACPI, a set of firmware code, is an industry standard used by modern general-purpose computers to manage these aspects of hardware.
    - ACPI provides code that runs as routines callable by the kernel for device state discovery and management, device error management, and power management.
    - E.g., when the kernel needs to quiesce (静默) a device, it calls the device driver, which calls the ACPI routines, which then talk to the device.

Transforming I/O Requests to Hardware Operations

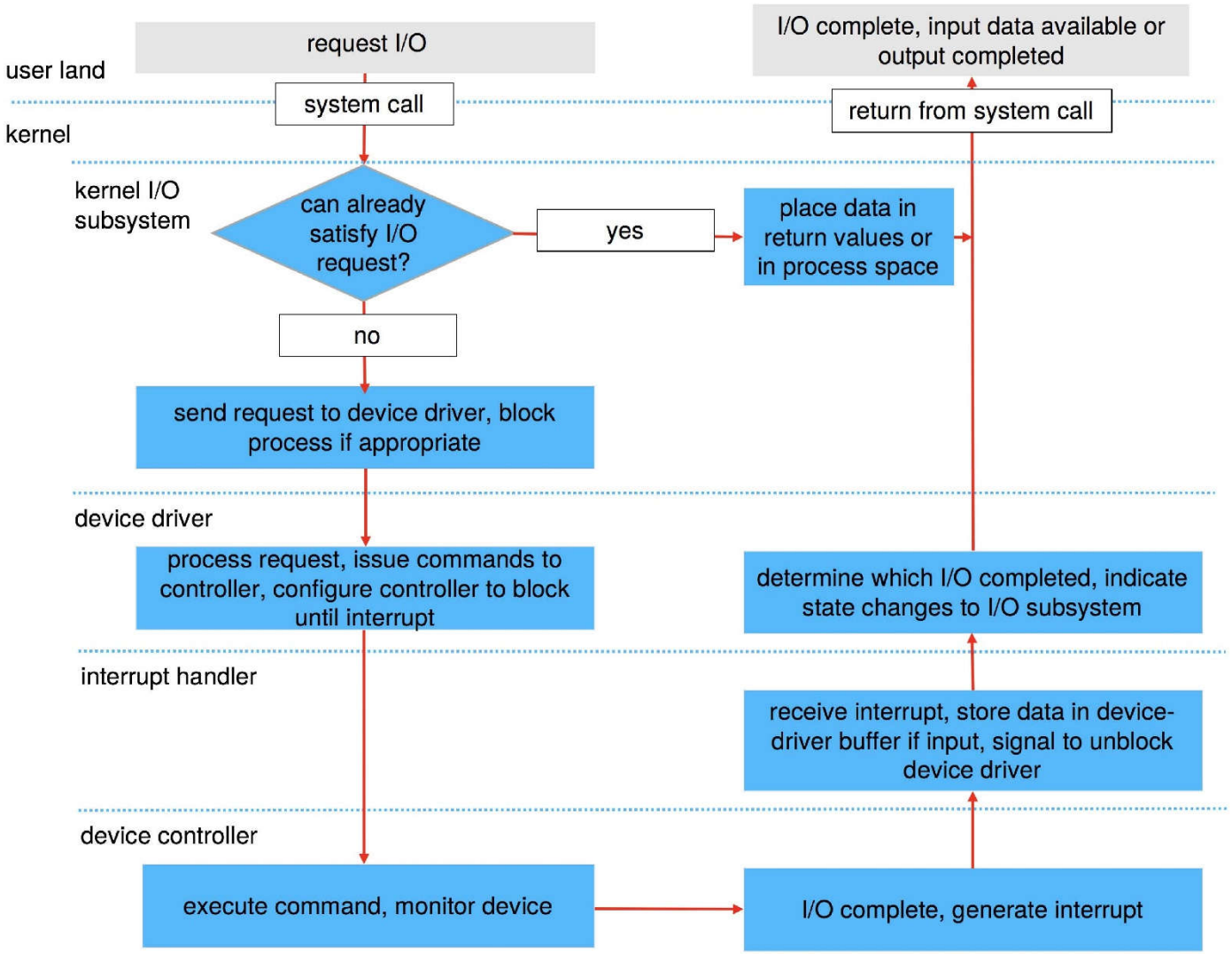
- Consider a blocking read request getting data from disk for a process.
  (1) A process issues a blocking read() system call to a file descriptor of a file that has been opened previously.
  (2) The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is completed.
  (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.
  (4) 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.
  (5) The device controller operates the device hardware to perform the data transfer.

Transforming I/O Requests to Hardware Operations

- Consider a blocking read request getting data from disk for a process.
  (6) The driver may poll for status and data, or it may have set up 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.
  (7) The correct interrupt handler receives the interrupt via the interrupt vector table, stores any necessary data, signals 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 signals the kernel I/O subsystem that the request has been completed.
  (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.

Transforming I/O Requests to Hardware Operations

- The Life Cycle of an I/O Request.

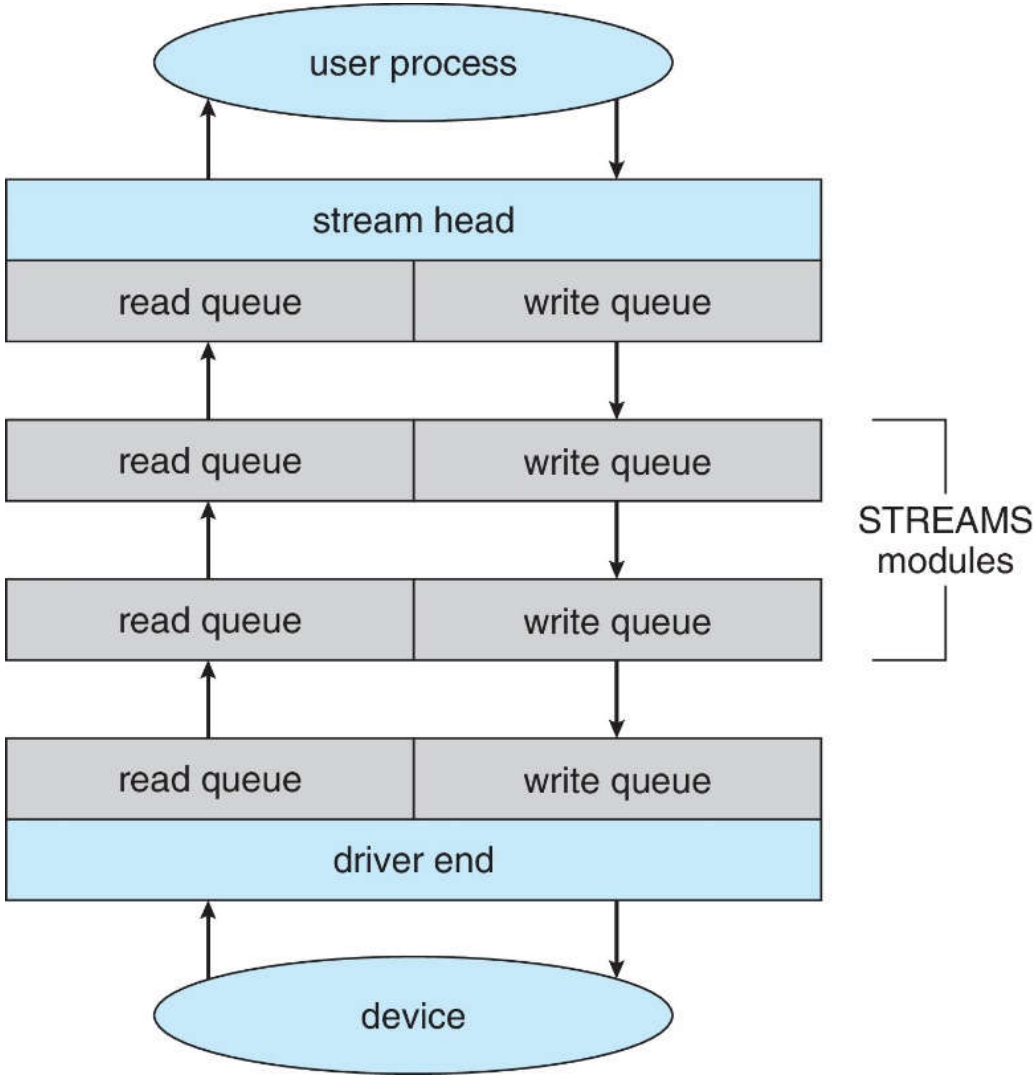


STREAMS

- *STREAMS* in Unix System V and beyond enables an application to assemble pipelines of driver code dynamically. A *STREAM* is a full-duplex communication channel between a user-level process and a device driver.
- A STREAM consists of:
  - a *stream head* that interfaces with the user process.
  - a *driver end* that controls the device; it must respond to interrupts, handle all incoming data.
  - zero or more *stream modules* (流模块) between the stream head and the driver end.
- Each stream module contains a *read queue* and a *write queue.*
  - Message passing is used to transfer data between queues.
  - A stream head may block if it is unable to copy a message to the next queue in line.
- Modules provide the functionality of STREAMS processing; they are *pushed* onto a stream by use of the ioctl() system call (in <stropts.h>).
  - For example, a process can open a USB device via a stream and can push on a module to the stream to handle input editing.

STREAMS

- The STREAMS Structure.



STREAMS

- Flow control
  - Messages are exchanged between queues in adjacent modules. A queue in one module may overflow an adjacent queue. To prevent this from occurring, a queue may support flow control to indicate available or busy.
  - Without flow control, a queue accepts all messages and immediately sends them on to the queue in the adjacent module without buffering them.
  - With flow control, a queue buffers messages and does not accept messages without sufficient buffer space. This process involves exchanges of control messages between queues in adjacent modules.
  - Drivers must support flow control as well.

STREAMS

- Read and Write through a STREAM
  - A user process writes data to a device using either the write() or putmsg() system call. The write() system call writes raw data to the stream, whereas putmsg() allows the user process to specify a message.
    - The stream head copies the data into a message and delivers it to the queue for the next module in line. This copying of messages continues until the message is copied to the driver end and hence the device.
  - Similarly, the user process reads data from the stream head using either the read() or getmsg() system call. If read() is used, the stream head gets a message from its adjacent queue and returns ordinary data (an unstructured byte stream) to the process. If getmsg() is used, a message is returned to the process.

STREAMS

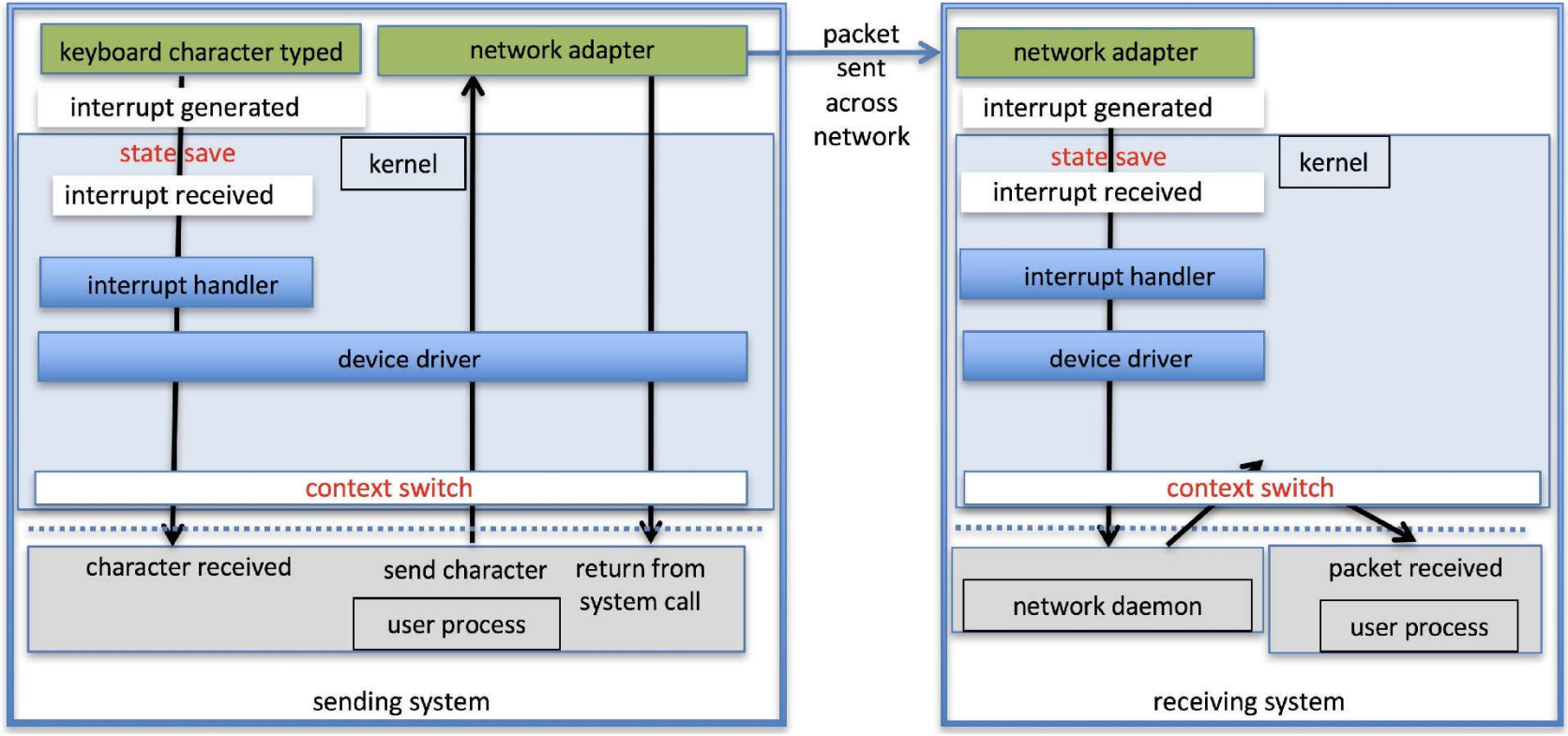
- Asynchronous and Synchronous Operations
  - STREAMS I/O is asynchronous (or nonblocking) except when the user process communicates with the stream head.
  - When writing to the stream, the user process will block, assuming the next queue uses flow control, until there is room to copy the message. Likewise, the user process will block when reading from the stream until data are available.

Performance

- I/O is a major factor in system performance.
  - I/O places heavy demands on the CPU to execute device-driver code and to schedule processes fairly and efficiently as they are blocked and unblocked. The resulting context switches stress the CPU and its hardware caches.
  - I/O also exposes any inefficiencies in the interrupt-handling mechanisms in the kernel. Interrupt handling is a relatively expensive task. Each interrupt causes the system to perform a state change, to execute the interrupt handler, and then to restore state.
  - I/O loads down the memory bus during data copies between controllers and physical memory and again during copies between kernel buffers and application data space. Coping gracefully with all these demands is one of the major concerns of a computer architect.
- Network traffic can also cause a high context-switch rate. Consider the inter-computer communications.

Performance

- Inter-computer Communications.

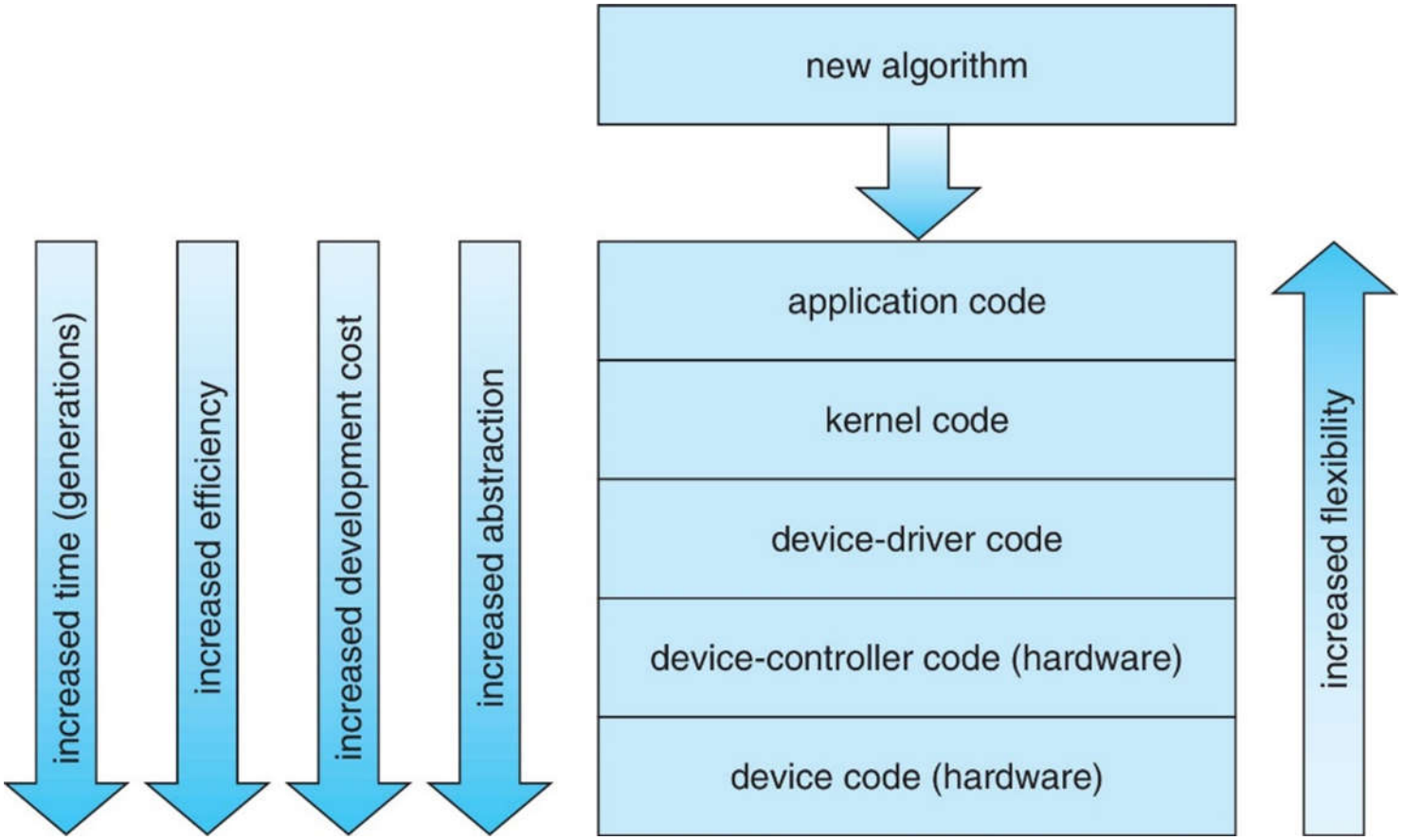


Performance

- Improving Performance
  - Some systems use separate *front-end processors* for terminal I/O to reduce the interrupt burden on the main CPU.
    - Terminal concentrator
    - I/O channel.
  - We can employ several principles to improve the efficiency of I/O.
    - Reduce the number of context switches.
    - Reduce the number of times that data must be copied in memory while passing between device and application.
    - Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy waiting minimized).
    - Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
    - Move processing primitives into hardware, to allow their operation in device controllers to be concurrent with CPU and bus operation.
    - Balance CPU, memory subsystem, bus, and I/O performance, for an overload in any one area will cause idleness in others.

Performance

- Device-Functionality Progression
  - Where should the I/O functionality be implemented—in the device hardware, in the device driver, or in application software?



Performance

- Device-Functionality Progression.
  - Initially, Let I/O algorithms be implemented at the application level.
    - Application code is flexible and application bugs are unlikely to cause system crashes.
    - Device drivers need not to reboot or reload after every change to the code.
    - It can be inefficient.
      - overhead of context switches
      - application cannot take advantage of internal kernel data structures and kernel functionality.
  - An in-kernel implementation can improve performance, but the development effort is more challenging.
  - The highest performance may be obtained through a specialized implementation in hardware, either in the device or in the controller. It is difficult and expensive, with the increased development time and the decreased flexibility.

Performance

- I/O performance of Storage
  - Over time, as with other aspects of computing, I/O devices have been increasing in speed.
    - Nonvolatile memory devices are growing in popularity and in the variety of devices available.
    - The speed of NVM devices varies from high to extraordinary, with next-generation devices nearing the speed of DRAM.
  - These developments are increasing pressure on I/O subsystems as well as operating system algorithms to take advantage of the read/write speeds now available.

Performance

- I/O performance of Storage.
  - CPU Cache/SRAM, DIMM(Dual inline memory model) DRAM, NVDIMM, PCIe/NVMe NVM, PCIe/NVMe SSD, SAS(Serial attached SCSI) SSD, SAS HDD, SATA(Serial Advanced Technology Attachment) HDD, …