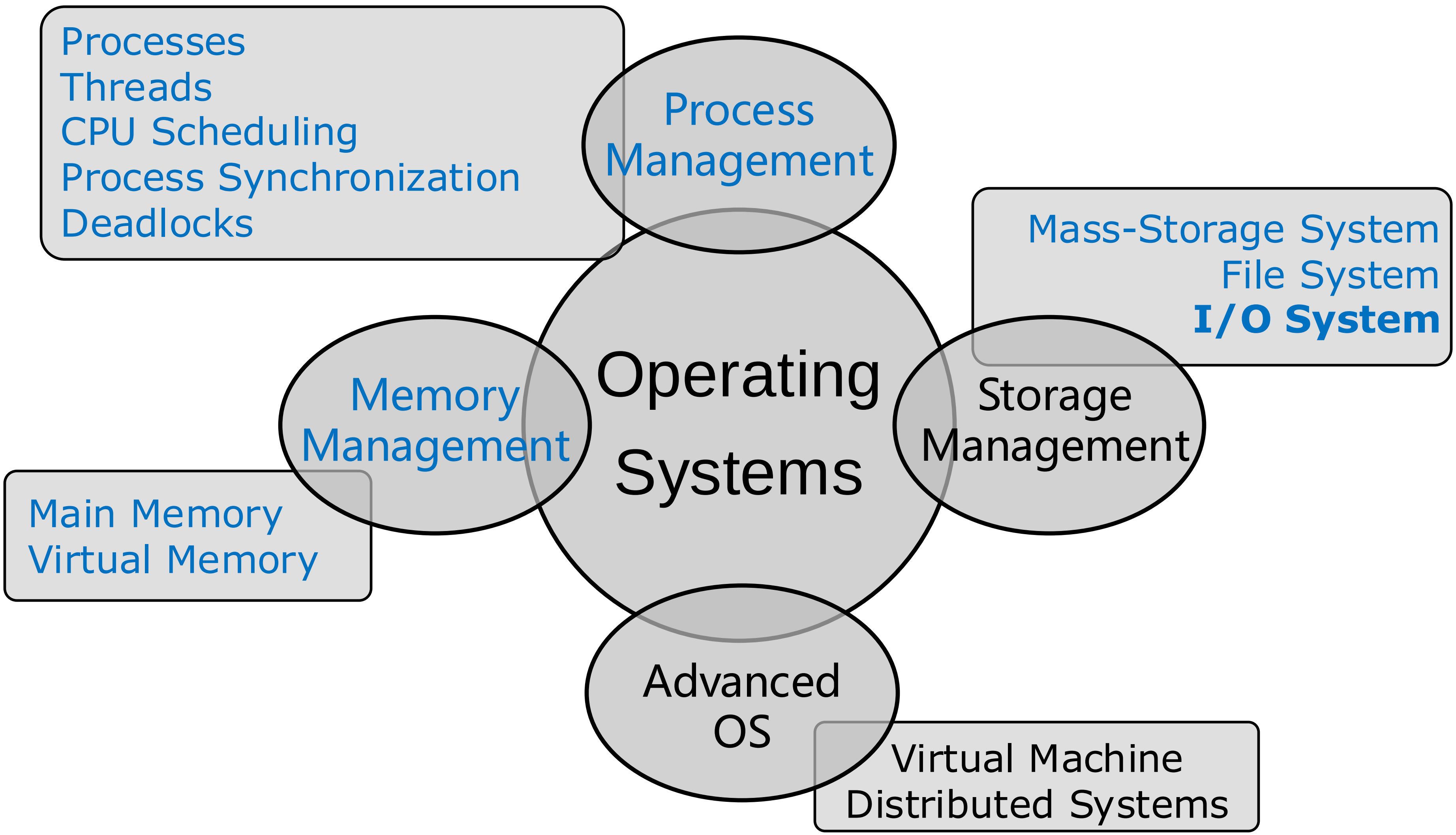
I/O Systems

Shengzhong Liu
Department of Computer Science and Engineering
Shanghai Jiao Tong University

Operating System Topics



Outline

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

Overview

- I/O management is a major component of OS design and operation
  - I/O devices vary greatly, and new types of devices frequently appear
  - Various methods to control them
  - Performance management

- **Ports**, **buses**, and **device controllers** connect to various devices

- **Device drivers (设备驱动)** encapsulate device details
  - Present uniform device-access interface to I/O subsystem

I/O Hardware

I/O Device Types


Keyboard/Trackpad/Mouse


Touchscreen

Scanner


Microphone


Webcam


Monitor


Printer

Projector


Headphone


Plotter

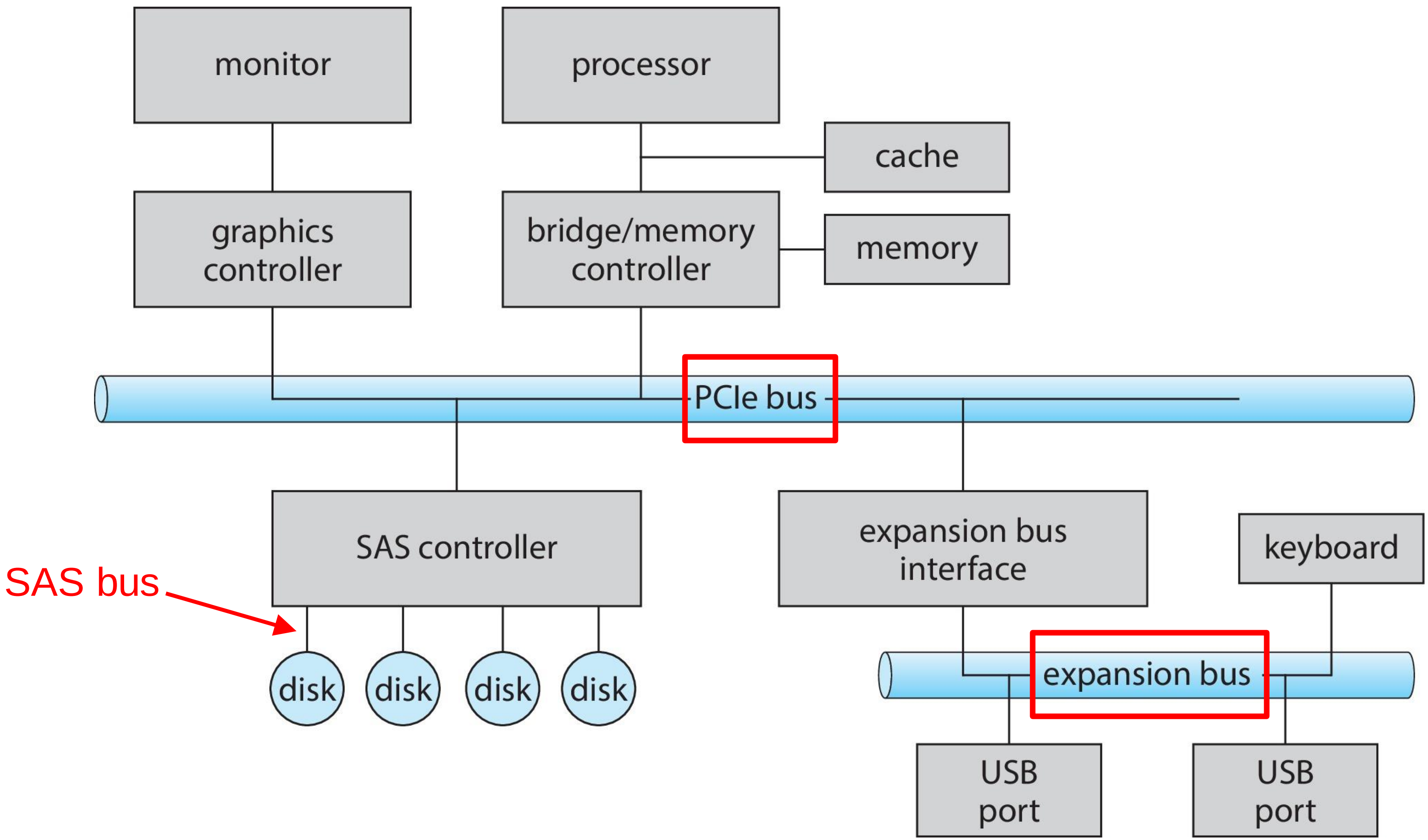

Joystick


AR/VR Headset

I/O Hardware

- Incredible variety of I/O devices, including three general categories:
  - **Storage**: disks, tapes
  - **Transmission**: network connections, Bluetooth
  - **Human-interface**: screen, keyboard, mouse, audio in and out
- Common concepts – signals from I/O devices interface with computer
  - **Port** – connection point for devices
  - **Bus** - daisy chain or shared direct access
    - PCI bus common in PCs and servers, PCI Express (PCIe)
    - Expansion bus connects relatively slow devices
    - Serial-attached SCSI (SAS) common disk interface
  - **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.
    - **Fibre channel** (**FC**) is complex controller, usually separate circuit board (**host-bus adapter**, **HBA**) plugging into bus

A Typical PC Bus Structure



Host-Controller Communication

- I/O instructions control devices
- The controller has one or more registers for data and control signals.
  - E.g., data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- Two ways of communication between the **host processor** and **controllers**:
  - **Option 1**: **Special I/O instructions**
    - Specify the transfer of a byte or a word to an I/O port address
  - **Option 2: Memory-mapped I/O**
    - Device-control registers are mapped into the processor address space
    - Especially for large address spaces (graphics)

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)

Host-Controller Communication: Polling (轮询)

- The host writes output through a port, handshaking with the controller.

- For each byte of I/O,
  1. The host repeatedly reads `busy bit` from the `status register` until 0
  2. Host sets `write bit` in the `command register`, and write byte into `data-out register`
  3. Host sets `command-ready bit`
  4. Controller notices the `command-ready bit`, and sets `busy bit`
  5. Controller reads the `command register`, reads the `data-out register`, and does the I/O to the device
  6. Controller clears `busy bit`, `error bit`, `command-ready bit` when transfer done

- Step 1 is **busy-wait** cycle (or **polling**) to wait for I/O from device
  - Reasonable if device is fast, but inefficient if device is slow
  - If the wait may be long, CPU should switch to other tasks
    - How does the host know when the controller become idle?

Host-Controller Communication: Interrupts

- Polling can happen in 3 instruction cycles
  - 1) Read status, 2) logical-and to extract status bit, 3) branch if not zero
  - How to be more efficient if non-zero infrequently?

- **Interrupt**: The hardware controller notifies the CPU when the device is ready for service
  1. **Raised** by the device controller by asserting a signal on the interrupt request
  2. **Caught** by the CPU
  3. **Dispatched** to the interrupt handler
  4. **Cleared** by the interrupt handler by serving the device

- **Interrupt vector**: saves memory addresses of interrupt handlers
  - Two types of interrupt:
    - **Non-maskable**: Signal error conditions, page faults, and debug requests
    - **Maskable**: Used for device-generated interrupts
  - Interrupt chaining if more than one device at the same interrupt number
    - Each element points to the head of an interrupt handler list

Frequent Interrupts in OS

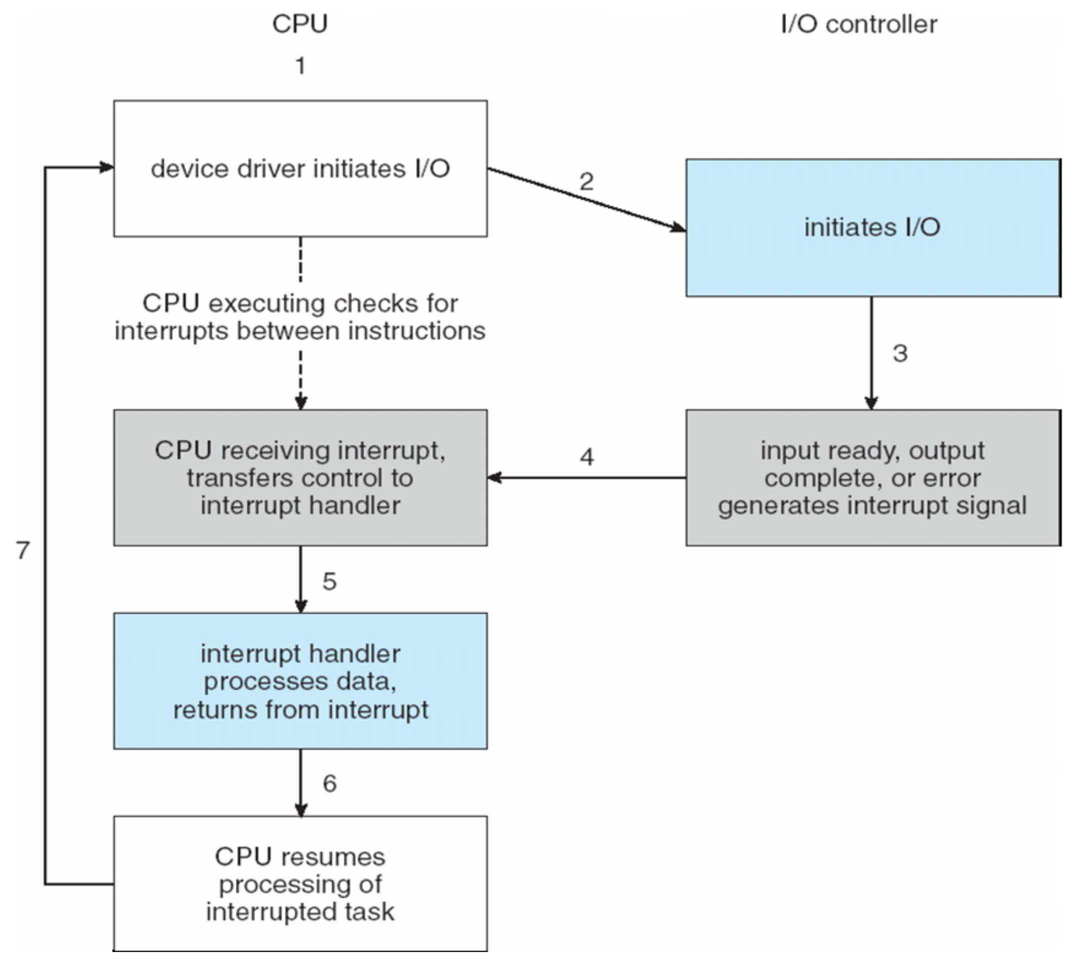
- **Stressing interrupt management** because
  - Single-user systems manage hundreds of interrupts per second
  - Servers can have hundreds of thousands of interrupts per second
- For example, a quiet macOS desktop generated 23,000 interrupts over 10 seconds

```
Fri Nov 25 13:55:59                                  0:00:10
                          SCHEDULER      INTERRUPTS
---------------------------------------------
total_samples                    13           22998

delays <  10 usecs               12           16243
delays <  20 usecs                1            5312
delays <  30 usecs                0             473
delays <  40 usecs                0             590
delays <  50 usecs                0              61
delays <  60 usecs                0             317
delays <  70 usecs                0               2
delays <  80 usecs                0               0
delays <  90 usecs                0               0
delays < 100 usecs                0               0
total  < 100 usecs               13           22998
```

Interrupt-Driven I/O Cycle

1. CPU senses **interrupt-request line** after executing every instruction
2. When CPU detects a controller has asserted a signal on the interrupt-request line, it performs a state save and jumps to the interrupt-handler
3. The **interrupt handler** determines the cause of the interrupt, performs the necessary processing
4. The interrupt handler performs a state restore, and returns the CPU to the execution state prior to the interrupt

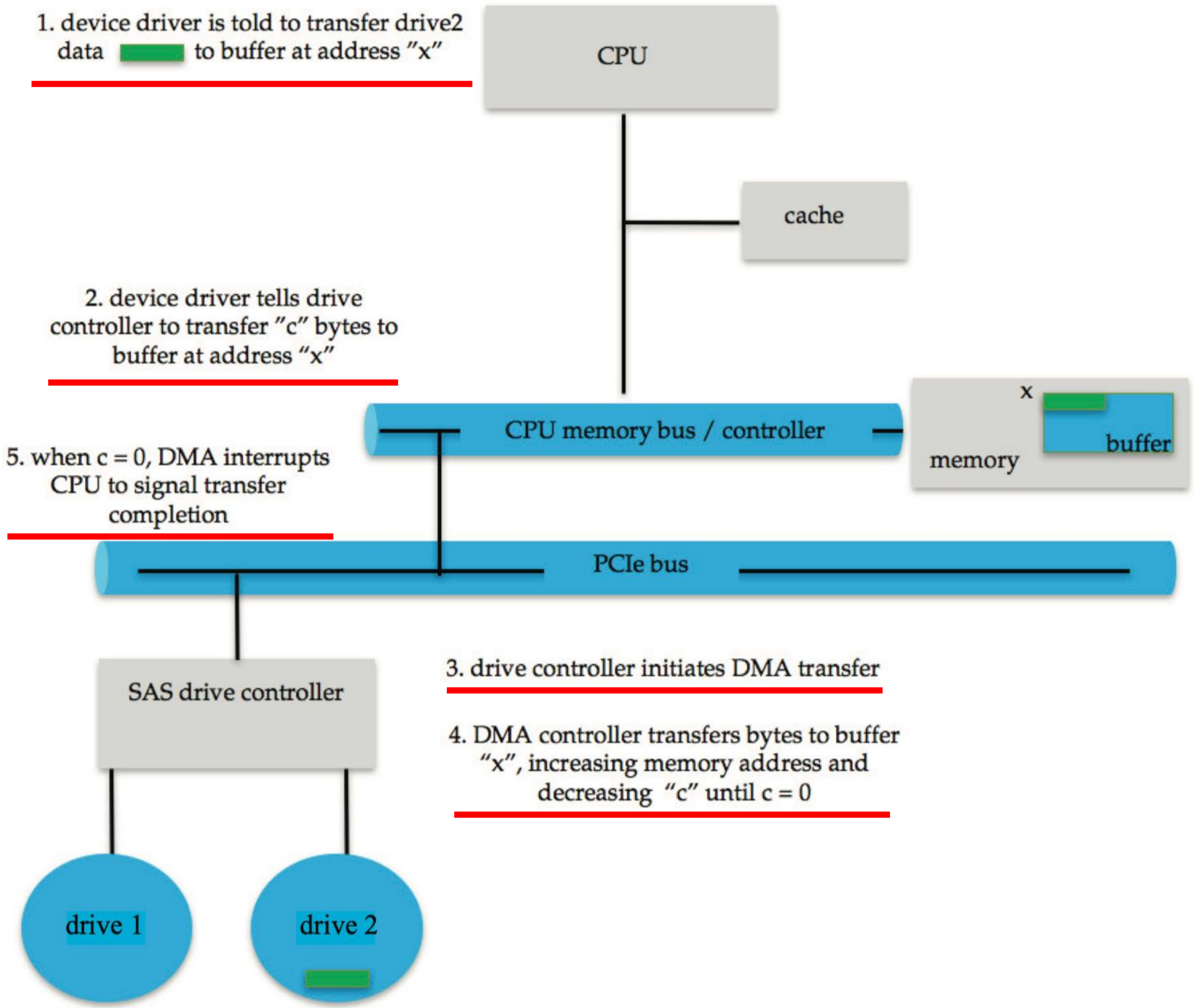


SHANGHAI JIAO TONG UNIVERSITY

Direct Memory Access (DMA)

- Used to avoid **programmed I/O** (one byte at a time) for host-controller communication with large data movement
  - It requires a **DMA** controller
- **Idea**: Bypasses CPU to transfer data directly between I/O device and memory
- To initiate a DMA transfer, OS writes DMA command block into the memory
  - The command block contains the source and destination addresses of the transfer, as well as the number of bytes to transfer
  - CPU writes the location of the command block to the DMA controller, then works on other tasks
  - **DMA controller operates the memory bus directly without CPU involvement**
  - When done, the DMA controller interrupts the CPU to signal completion
- Version that is aware of virtual addresses can be even more efficient - **DVMA**

Six Step Process to Perform DMA Transfer



I/O Hardware Concepts

- The main concepts in I/O hardware:
  - A bus
  - A controller
  - An I/O port and its registers

- Handshaking between the **host** and a **device controller** through a polling loop or via interrupts

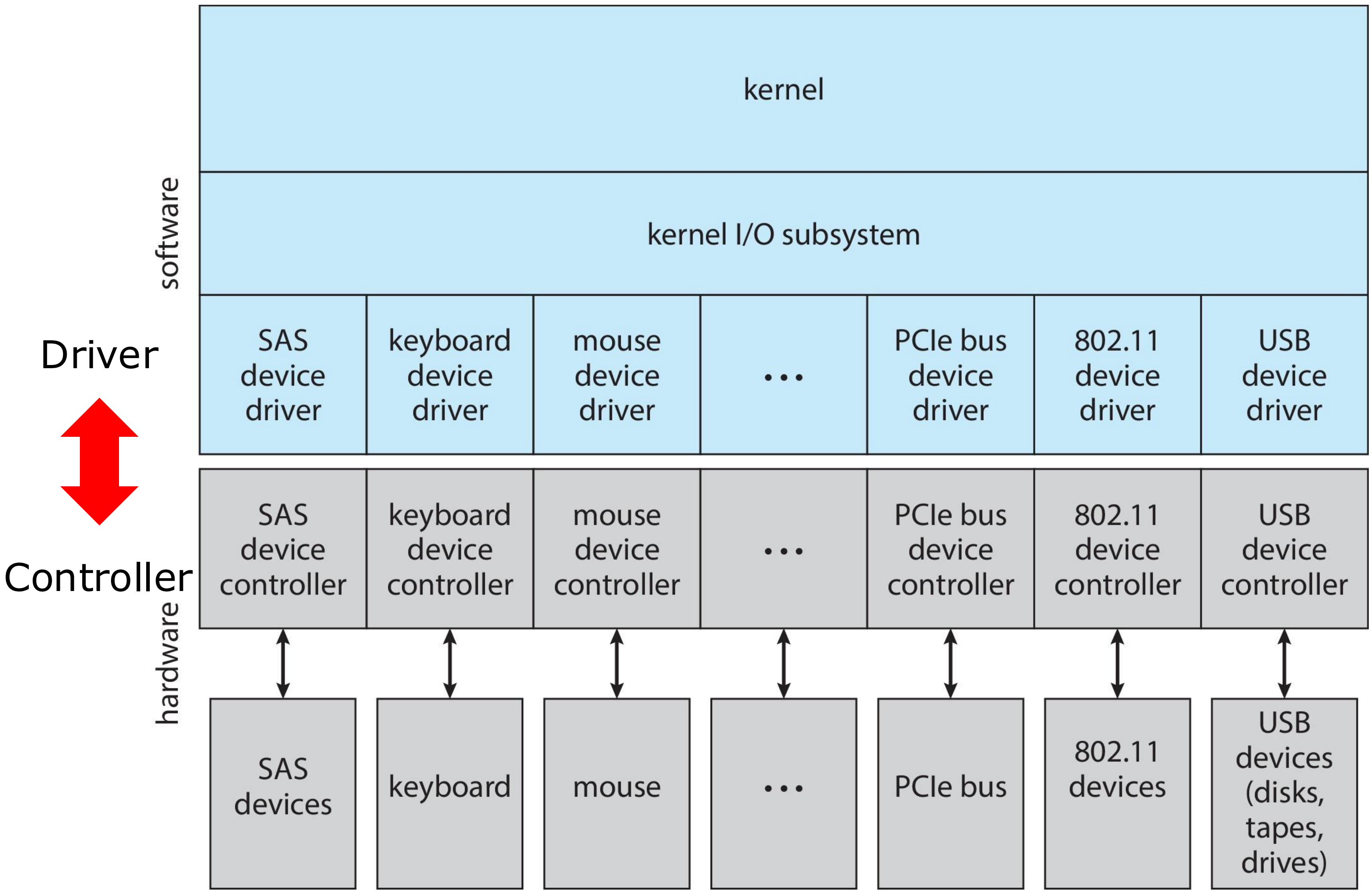
- Communication can be offloaded to a **DMA controller** for large data transfers

Application I/O Interface

Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
  - Device-driver layer hides **I/O controllers differences** from the **kernel**
  - New devices using existing protocols need no extra work
- Each OS has 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.)

- Differences 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 to transparently pass commands from application to device driver
  - Unix **`ioctl()`** **[I/O Control]** call to send arbitrary bits to a device control register and data to device data register

- UNIX and Linux use tuple of "**major**" and "**minor**" device numbers to identify type and instance of devices (here major 8 and instance number 0-4)

```
% ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda
brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1
brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2
brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3
```

I/O Devices: (1) Block and Character Devices

- The **block-device** interface captures all the aspects necessary for accessing disk drives and other block-oriented 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 can be on top of block-device drivers
    - File mapped to virtual memory and clusters brought via demand paging

- **Character devices** include keyboards, mouse, serial ports
  - Commands include **`get(), put()`**
  - Libraries layered on top allow line editing

I/O Devices: (2) Network Devices

- Performance differs from block and character to have their own interface
  - Facilitate distributed applications

- Linux, Unix, Windows , and many others include **socket** interface
  - Separates network protocol from network operation
  - Includes **`select()`** function
    - Returns which sockets have a packet waiting and which have room to accept new packet

- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

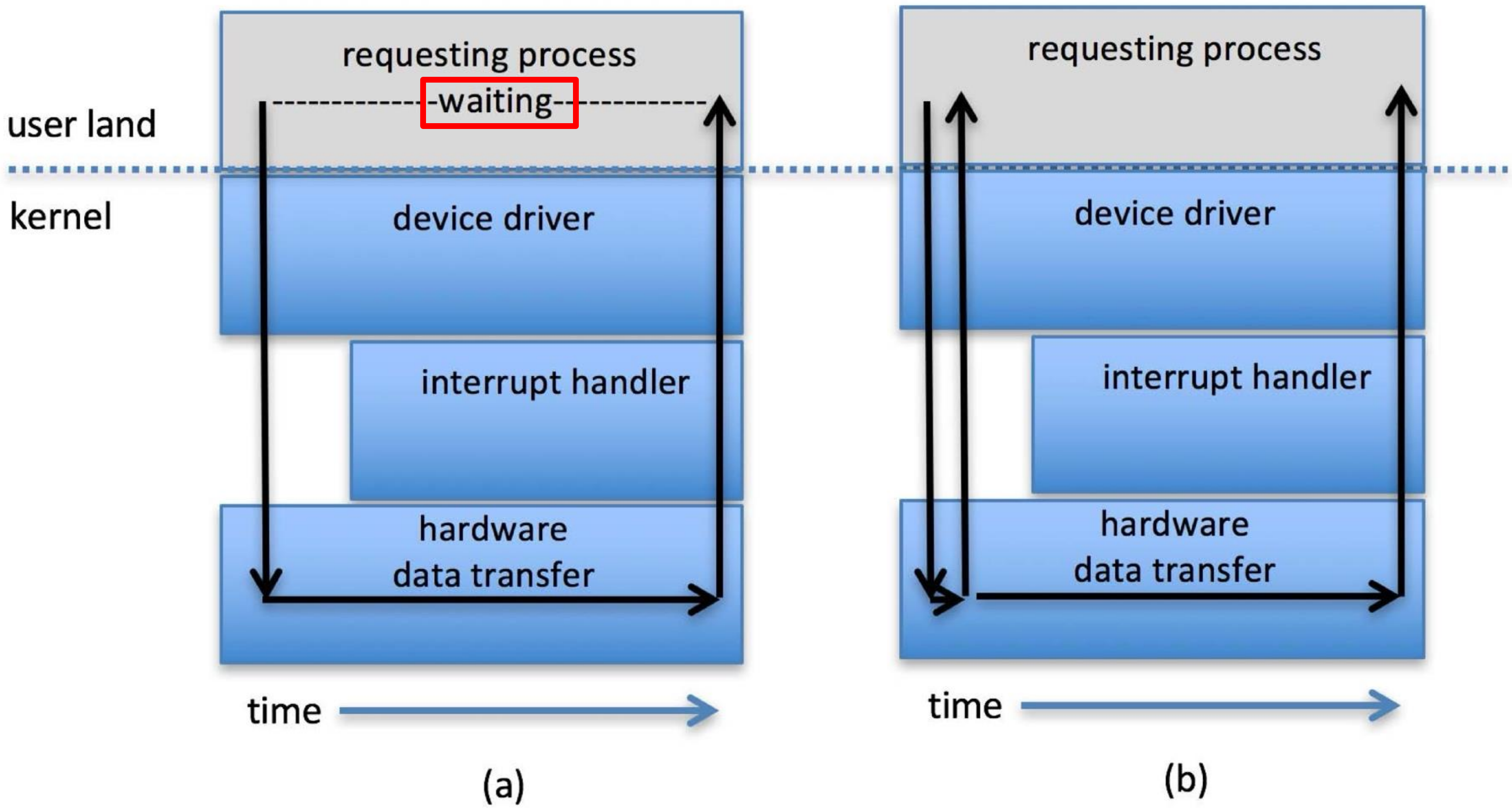
I/O Devices: (3) Clocks and Timers

- Provide current time, elapsed time, and timer to trigger operation
- Normal resolution about 1/60 second
  - Some systems provide higher-resolution timers
- **Programmable interval timer** used for timings, periodic interrupts
  - Can be set to wait a certain amount of time and then generate an interrupt
  - Disk I/O uses it to invoke periodic flushings of dirty cache buffers to disk
  - Network subsystem uses it to cancel operations that are proceeding too slowly because of network congestion or failures
- **`ioctl()`** (on UNIX) covers odd aspects of I/O such as clocks and timers

Blocking and Nonblocking I/O

- **Blocking I/O (阻塞)** - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- **Nonblocking I/O （非阻塞）** - I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading
  - Returns quickly with the count of bytes read or written
  - Example: `select()` to find if data ready then `read()` or `write()` to transfer
- **Asynchronous** - process runs while I/O executes
  - Returns immediately, without waiting for the I/O to complete
  - I/O subsystem signals process when I/O completed
  - Not exposed to user applications but contained within the OS operation

Synchronous vs. Asynchronous I/O



Synchronous I/O

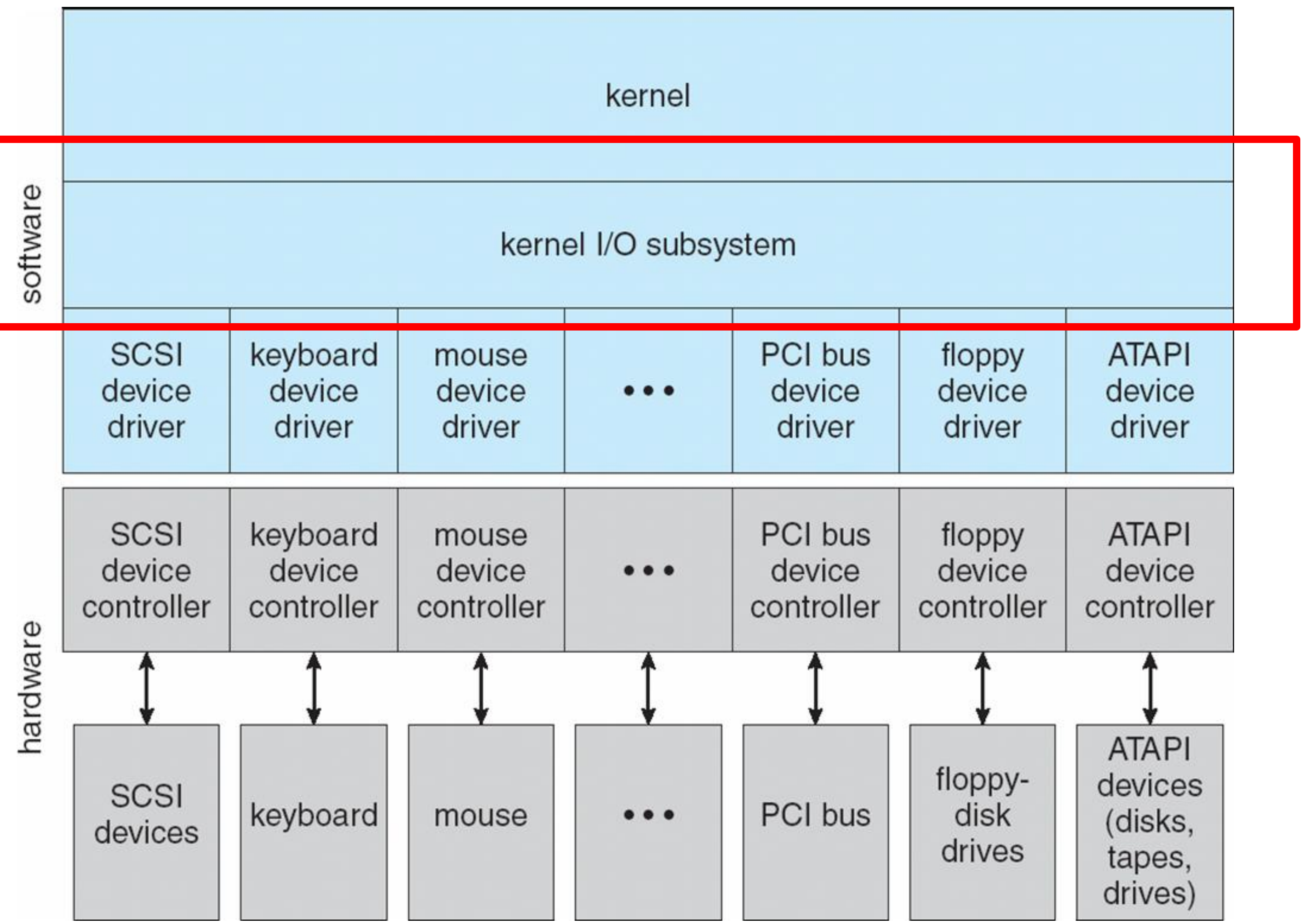
Asynchronous I/O

Vectored I/O

- **Vectored I/O (向量)** allows one system call to perform multiple I/O operations
- For example, Unix **`readve()`** accepts a vector of multiple buffers to read into or write from
- This scatter-gather method is better than multiple individual I/O calls
  - Decreases **context switching** and **system call** overhead
  - Some OS versions provide atomicity（**原子实现**）
    - No need to worry about multiple threads changing data as reads/writes occurring

Kernel I/O Subsystem

A Kernel I/O Structure



Kernel I/O Subsystem

- **I/O Scheduling** – Determine a good order to execute I/O requests
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
  - Some implement Quality of Service
- **Buffering** - Store data in memory while transferring between devices
  - To cope with device speed mismatch (e.g., network vs. disk)
  - To cope with device transfer size mismatch (e.g, different buffer sizes at sender and receiver)
  - To maintain "copy semantics"
- **Double buffering** – Two copies of the data
  - Decouples the producer of data from the consumer, thus relaxing timing requirements between them
  - Copy-on-write can be used for efficiency in some cases

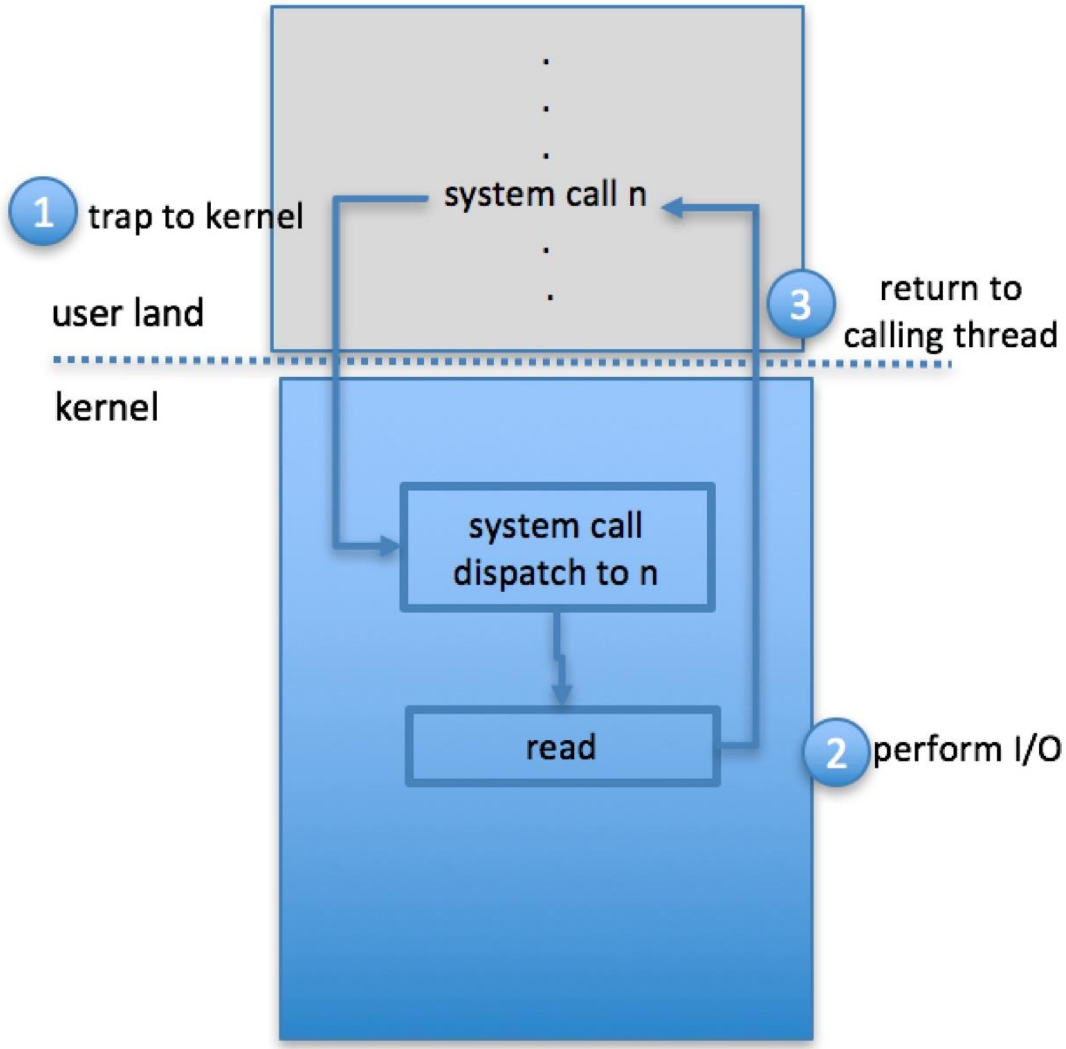
Kernel I/O Subsystem

- **Caching (缓存)** - faster memory region holding copy of data
  - Always just a copy
  - Key to performance
  - Sometimes combined with buffering
- **Spooling（假脱机）** - a buffer that holds 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
  - Enable a process to allocate an idle device and to deallocate that device when it is no longer needed
  - Watch out for deadlock

Error Handling and I/O Protection

- **Error handling**: OS can recover from disk read, device unavailable, transient write failures
  - Retry a read or write, for example
  - More advanced systems (Solaris FMA, AIX):
    - Track error frequencies
    - Stop using devices 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
- **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, but not by users
    - Memory-mapped and I/O port memory locations must be protected from user access

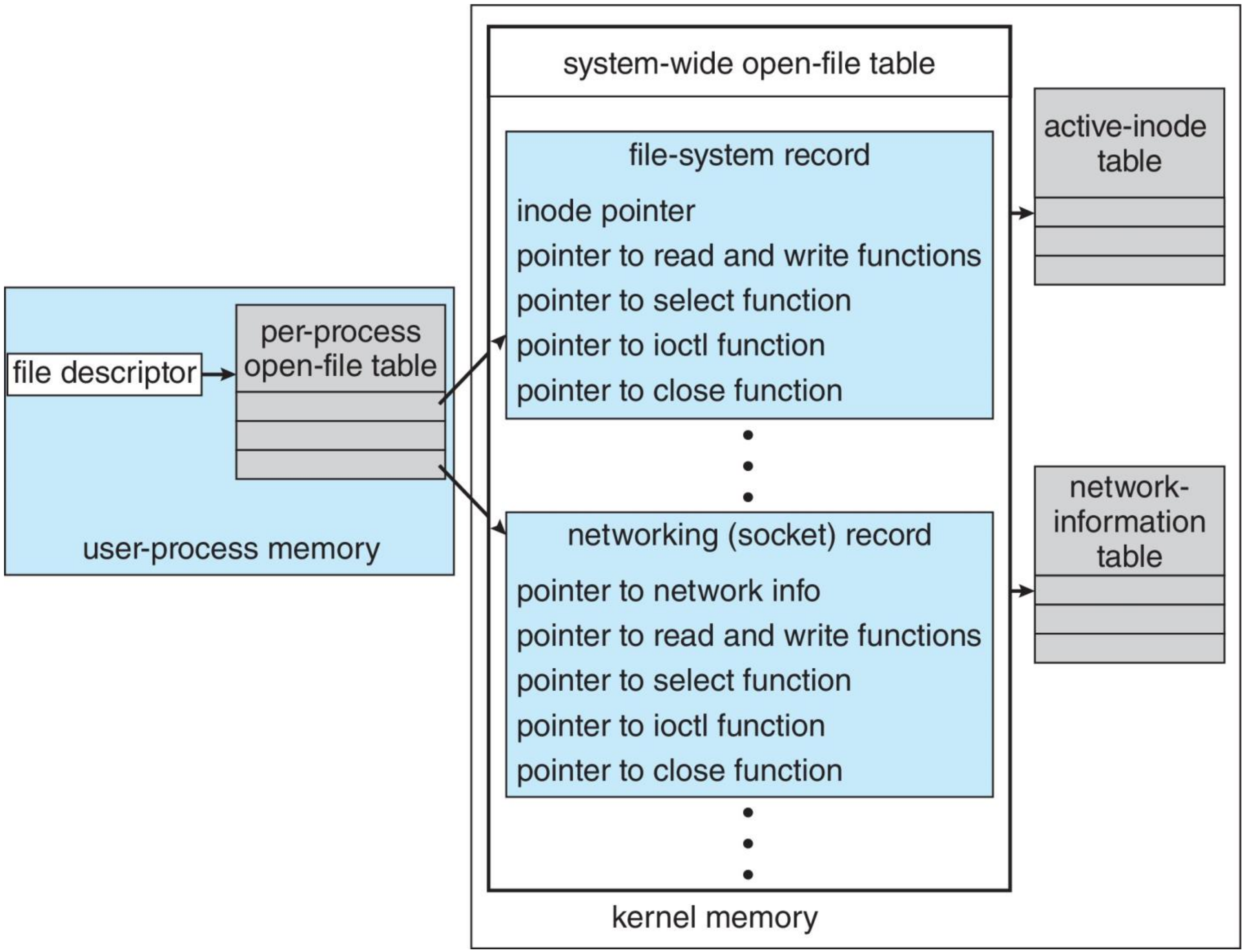
Use of a System Call to Perform I/O



Kernel Data Structures

- Kernel keeps state info for I/O components, including **open file tables**, **network connections**, **character device state**
- Many 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?
    - Add overhead but simplify the structure and design of I/O system and adds flexibility

UNIX I/O Kernel Structure

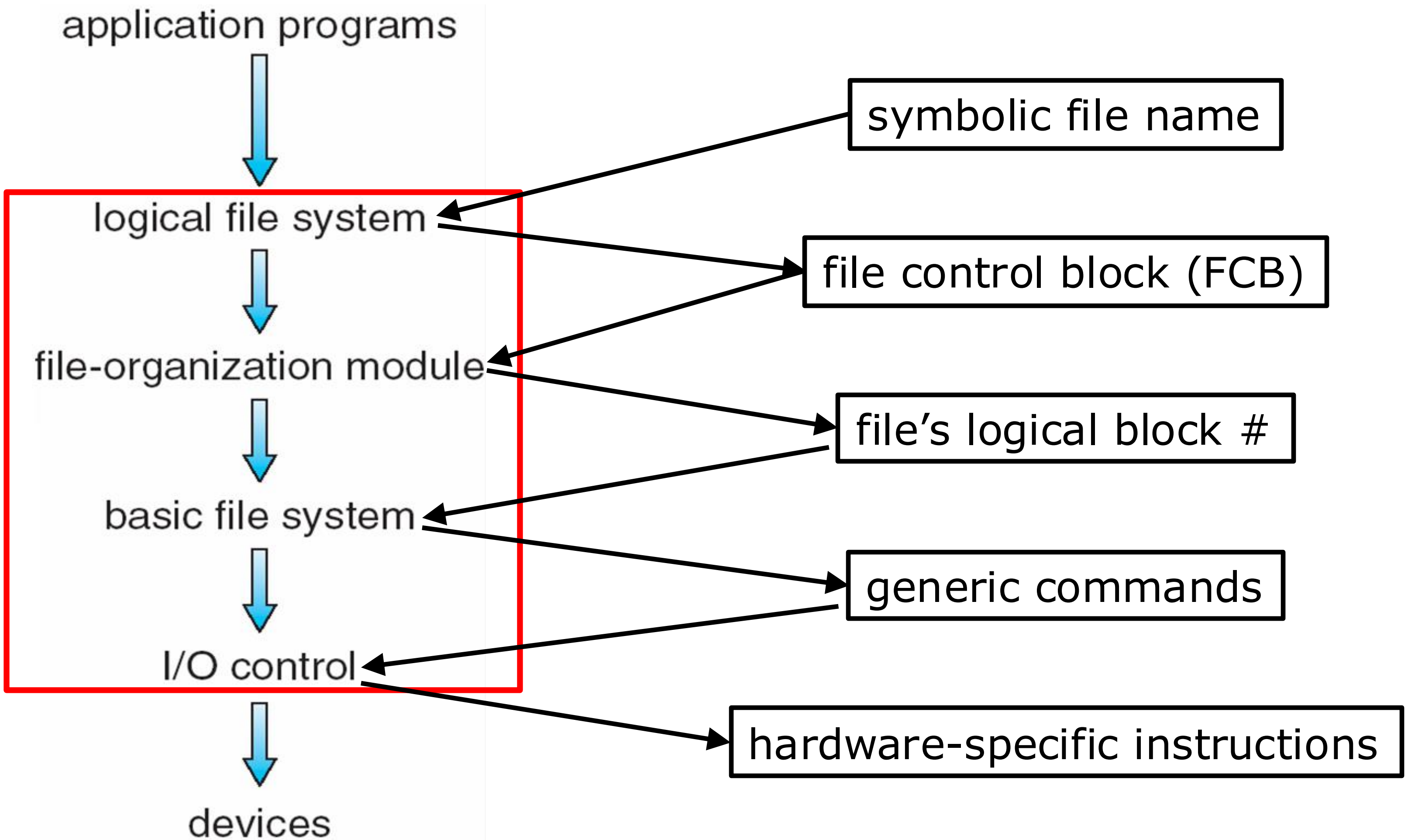


I/O Requests → Hardware Operations

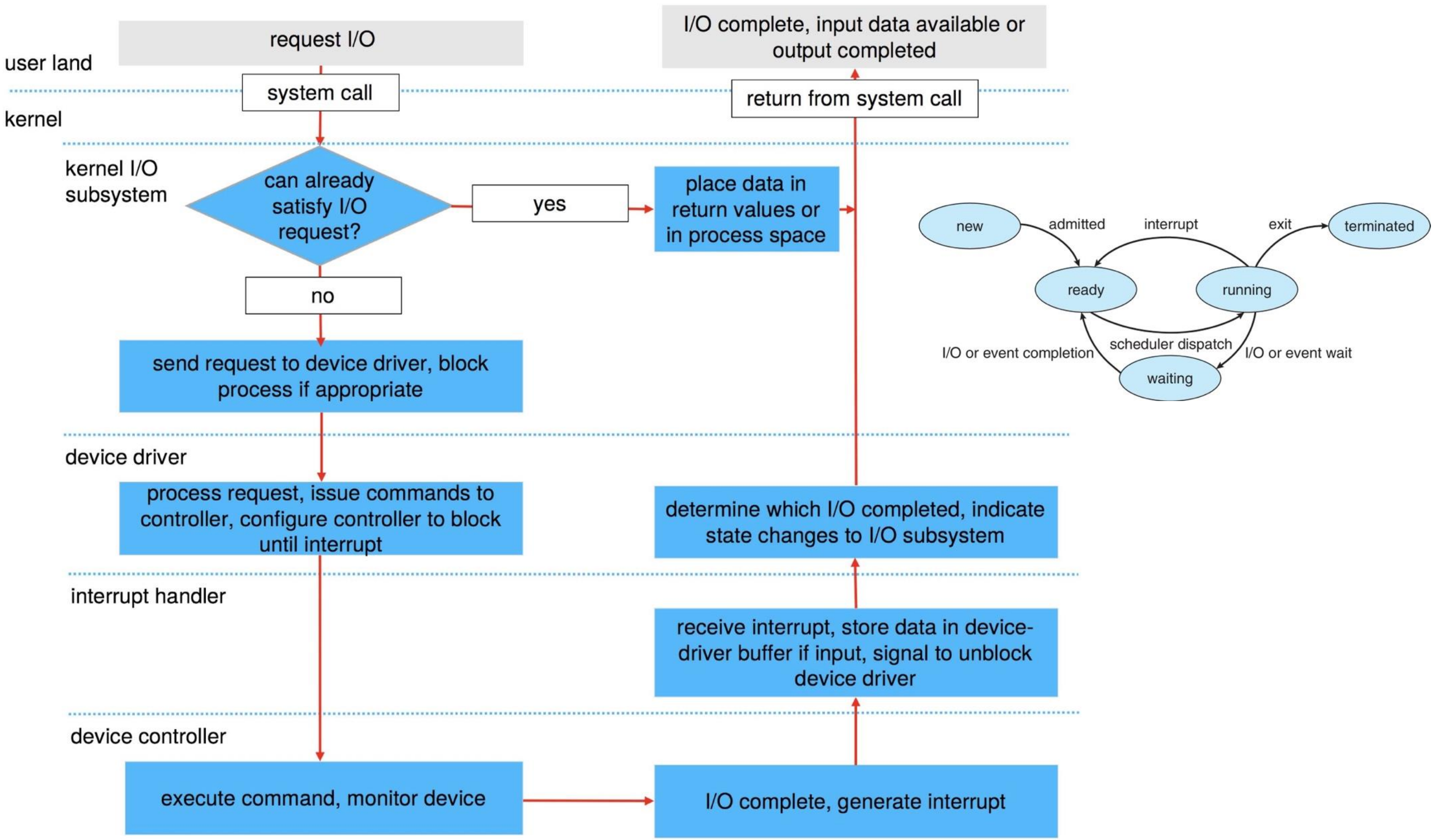
Transforming I/O Requests to Hardware Operations

- Consider **reading a file from disk** for a process:
  1. Determine the device holding the file
  2. **Translate name to device representation**
  3. Physically read data from disk into the buffer
  4. Make data available to the requesting process
  5. Return control to process

Recap: File System Layers and Operations



Life Cycle of An I/O Request



Summary

- The basic hardware elements involved in I/O are **ports**, **buses**, **device controllers**, and the **devices** themselves.
- Moving data between devices and main memory is performed by the CPU as **programmed I/O** or is **offloaded to a DMA controller**.
- The kernel module that controls a device is a **device driver**.
- The system call interface provided to applications is designed to handle several basic categories of hardware
  - Blocking vs. non-blocking vs. asynchronous calls
- Kernel's I/O subsystem provides numerous services: I/O scheduling, buffering, caching, spooling, device reservation, error handling, and translation.

Homework

- Reading
  - Chapter 12

- HW4 released today, due on **April 9, 23:59**!