



Operating System Practice

Che-Wei Chang

chewei@mail.cgu.edu.tw

Department of Computer Science and Information
Engineering, Chang Gung University

Advanced Operating System Concepts

- Chapter 10: File System
- Chapter 11: Implementing File-Systems
- Chapter 12: Mass-Storage Structure
- ➡ ● Chapter 13: I/O Systems
- Chapter 14: System Protection
- Chapter 15: System Security



Study Items

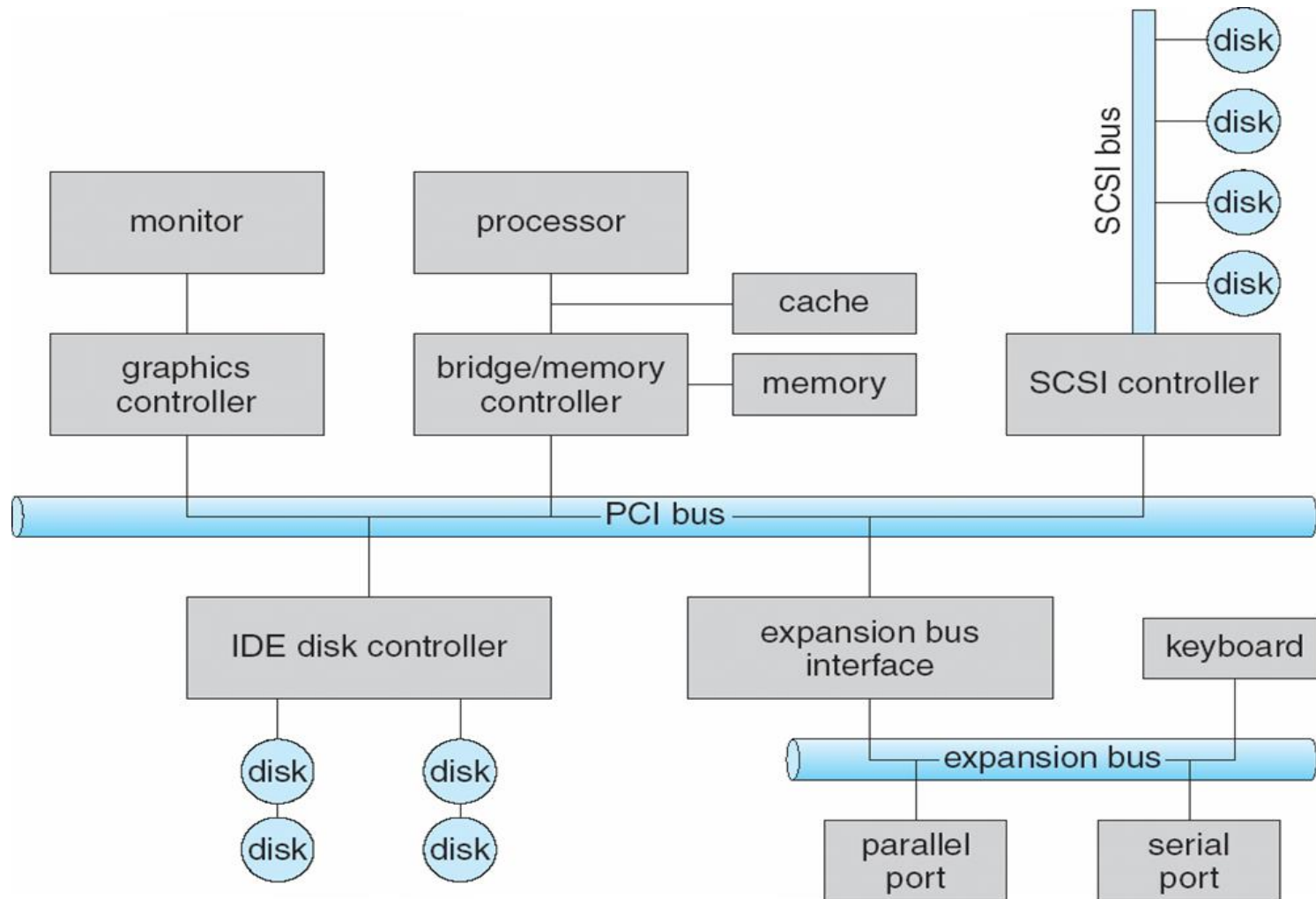
- ▶ I/O Hardware
- ▶ Application I/O Interface
- ▶ Kernel I/O Subsystem
- ▶ Transforming I/O Requests to Hardware Operations
- ▶ Performance



I/O Hardware

- ▶ Incredible Variety of I/O Devices
 - Storage
 - Transmission
 - Human-interface
- ▶ Common Concepts
 - **Port**: connection point for device
 - **Bus**: daisy chain or shared direct access
 - **Controller** (host adapter): electronics that operate port, bus, device

A Typical PC Bus Structure



Access to I/O Hardware

- ▶ Device registers which can be accessed by the host
 - The **data-in register** is read by the host to get **input**
 - The **data-out register** is written by the host to send **output**
 - The **status register** contains bits which indicate device **states**
 - The **control register** is written by the host to send **command**
- ▶ Methods to access devices with their addresses
 - **Direct I/O instructions**
 - **Memory-mapped I/O**
 - Device data and command registers mapped to 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)



Polling

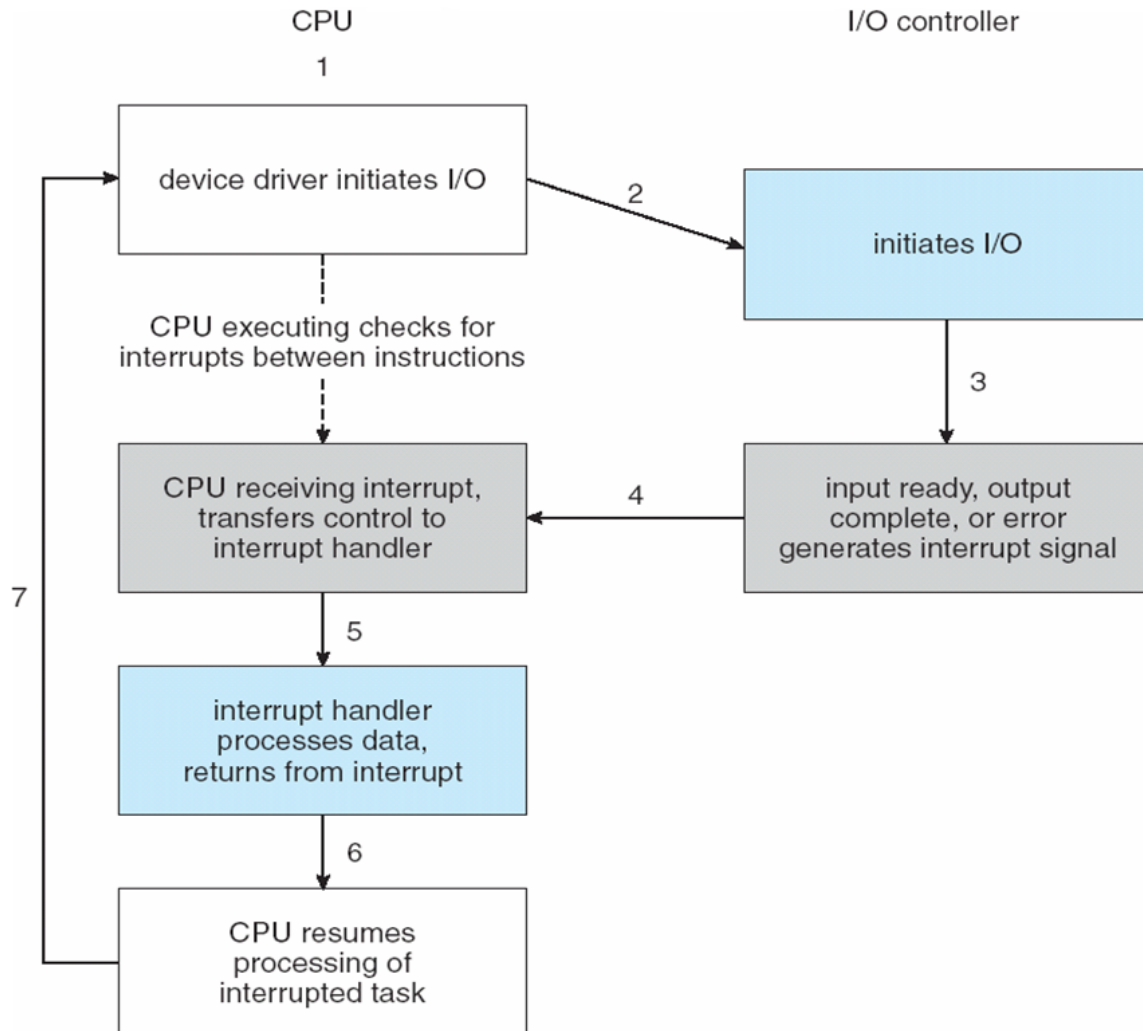
- ▶ An example of Polling I/O
 1. Read busy bit from status register until 0
 2. Host sets read or write bit and if write copies data into data-out register
 3. Host sets command-ready bit
 4. Controller sets busy bit, executes transfer
 5. Controller clears busy bit, error bit, command-ready bit when transfer done
- ▶ Step 1 is busy-wait cycle to wait for I/O from device
 - Reasonable if device is fast
 - But inefficient if device slow
 - CPU switches to other tasks?
 - Might miss some data

Interrupts

- ▶ CPU Interrupt-request line triggered by I/O device
 - Checked by processor after each instruction
- ▶ Interrupt handler receives interrupts
 - Masked to ignore or delay some interrupts
- ▶ Interrupt vector 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

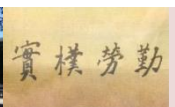


Interrupt-Driven I/O Cycle

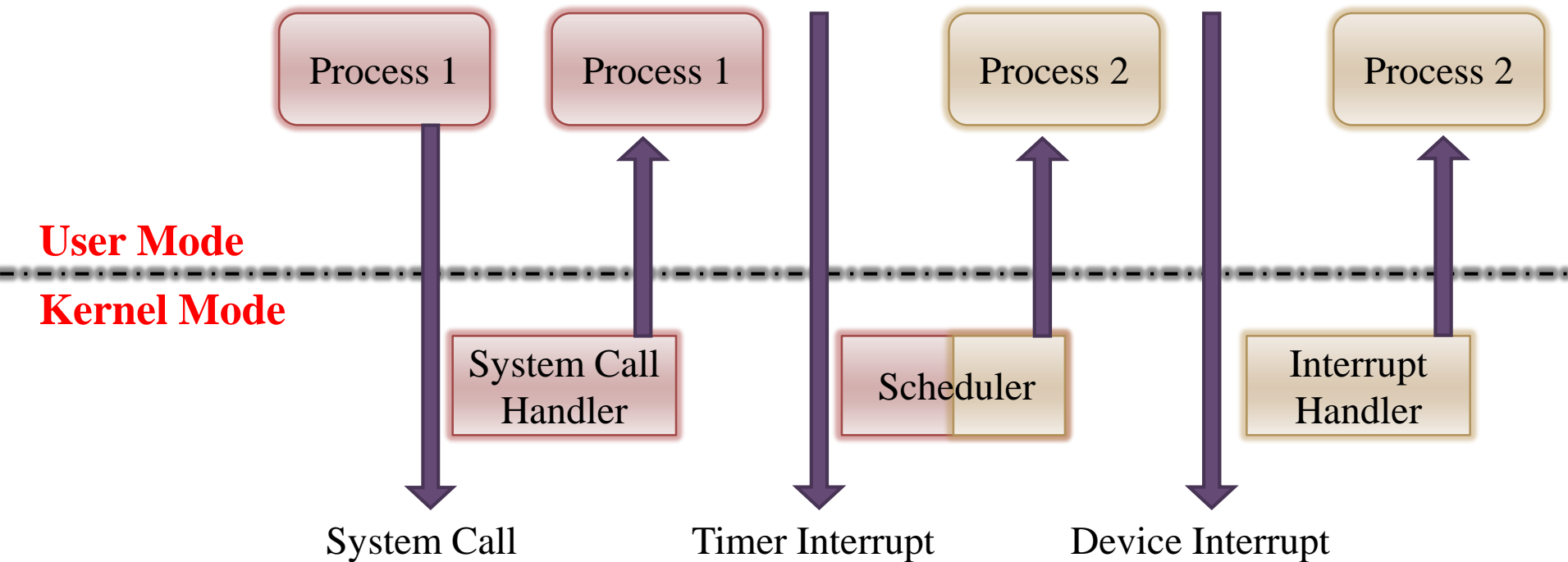


Interrupt Usage

- ▶ Interrupt vector table is used to identify which device sent out the interrupt
 - When multiple devices share a interrupt number, the handlers are checked one by one
- ▶ Interrupt mechanism also used for exceptions
 - Terminate process, crash system due to hardware error
 - Page fault executes when memory access error
 - System call executes via trap to trigger kernel to execute request
- ▶ Multi-CPU systems can process interrupts concurrently
 - If operating system designed to handle it



Transitions between User and Kernel Modes in Linux

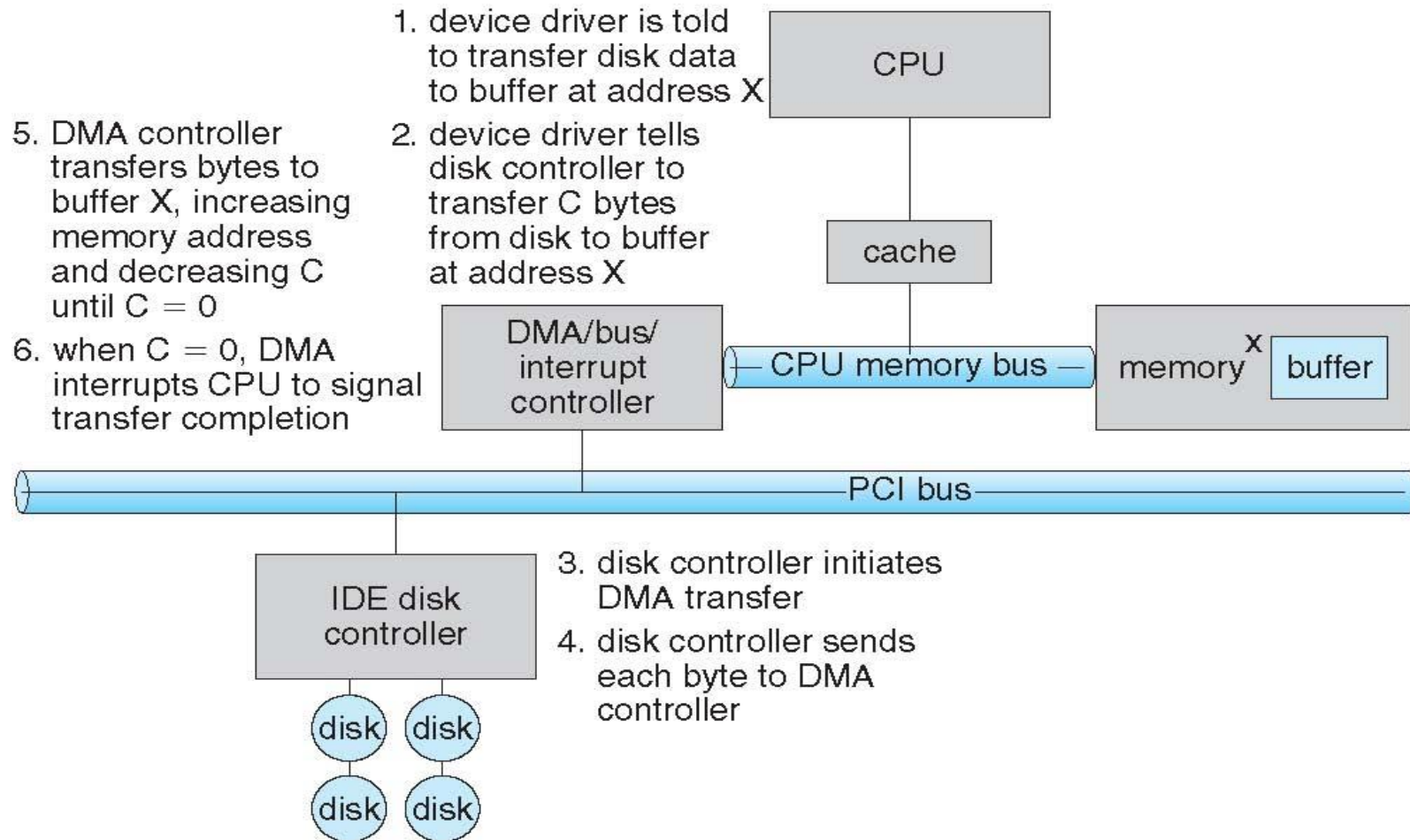


Direct Memory Access

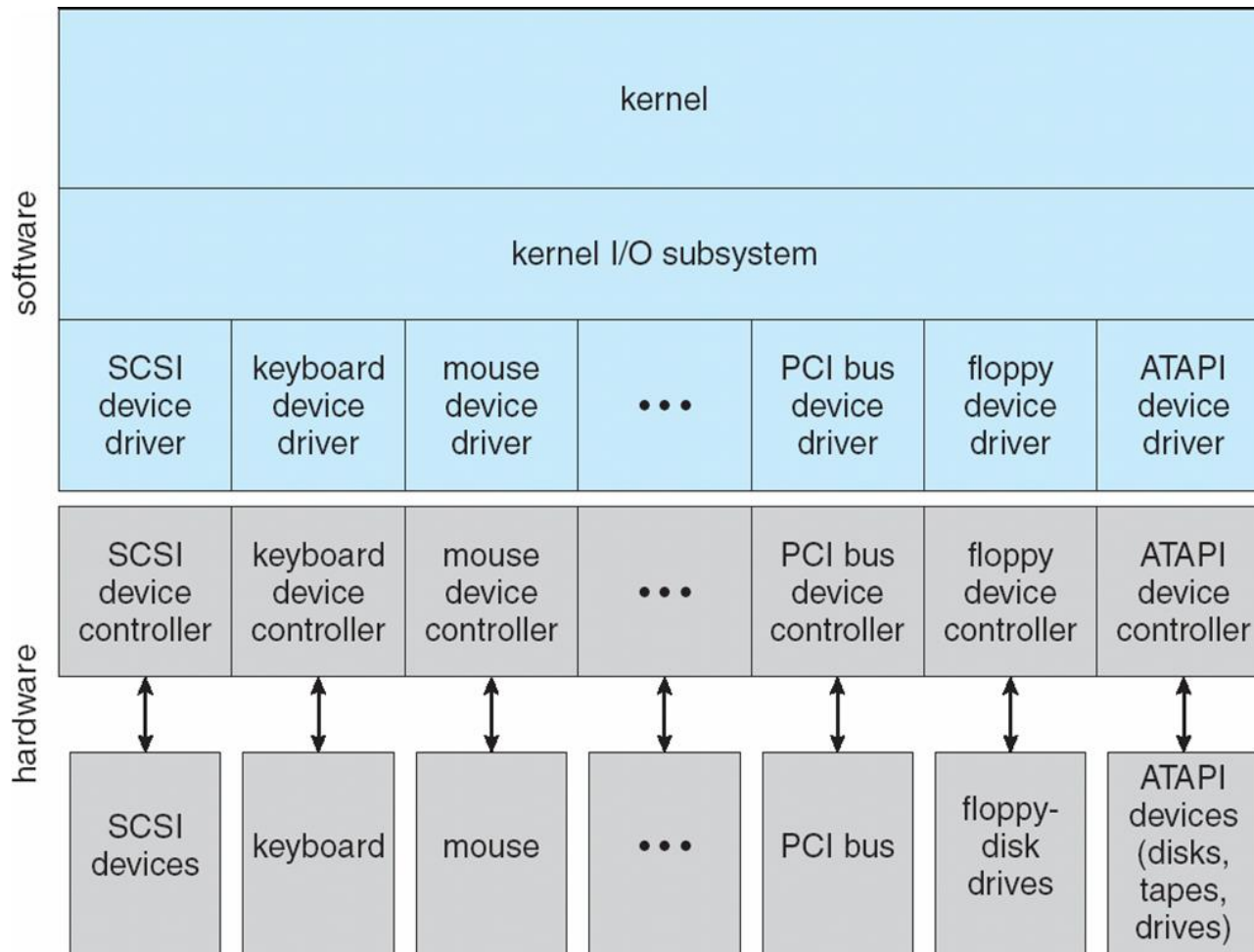
- ▶ Used to avoid programmed I/O (one or few bytes at a time) for large data movement
- ▶ Requires DMA controller
- ▶ Bypasses CPU to transfer data directly between I/O device and memory
- ▶ OS writes DMA command block into memory
 - Source and destination addresses
 - Read or write mode
 - Count of bytes
 - For each read/write:
 - Device ready → DMA-request
 - DMA controller complete → DMA-acknowledge
 - When done, interrupts to signal completion



DMA Transfer



Kernel I/O Structure



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
- ▶ Unix `ioctl()` call to send arbitrary bits to a device control register and data to device data register (called escape or back door)
 - Which device
 - Which command
 - The pointer to the data
- ▶ Device characteristics vary in many dimensions



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



Block and Character Devices

▶ Character Devices

- Sequential access
- Commands include `get()`, `put()`
- Examples include printer, sound board, terminal
- The same device may have both block and character oriented interfaces

▶ Block Devices

- Commands include `read`, `write`, `seek`
- Raw I/O, direct I/O, or file-system access
 - Raw I/O: no file system support, manage the device directly
 - Direct I/O: with file system support but without buffering and locking
- Block size is from 512B to 4KB
- For example, disks are commonly implemented as block devices



Network Devices

- ▶ Varying enough from block and character to have own interface
- ▶ Unix and Windows have **socket** (e.g., IP + port) interface
 - Separates network protocol from network operation
 - Includes `select()` functionality
 - `select()` returns that which sockets have data to be received and which sockets are available for sending data now
- ▶ Approaches vary widely



Clocks and Timers

- ▶ Hardware clocks and timers provide three basic functions:
 - Give the current time
 - Give the elapsed time
 - Set a timer to trigger operation X at time T
- ▶ Some high-frequency counters do not generate interrupts, but they offers accurate measurements of time intervals

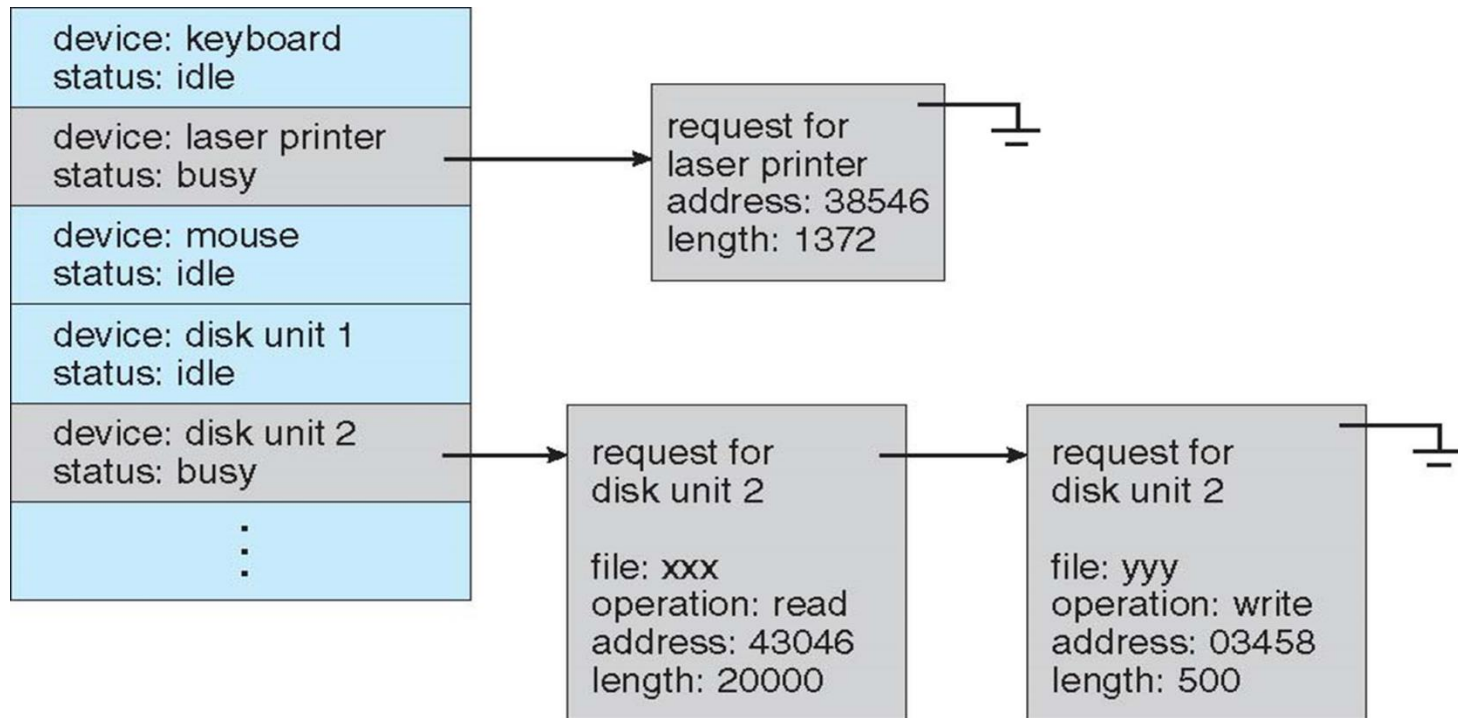


Blocking, Nonblocking, and Asynchronous I/O

- ▶ Blocking I/O
 - The execution of the application is suspended until the expected results are provided
- ▶ Nonblocking I/O
 - Nonblocking call returns quickly, with a return value that indicates how many bytes were transferred
- ▶ Asynchronous I/O
 - Asynchronous call returns immediately, without waiting for the I/O to complete
 - The completion of the I/O at some future time is communicated to the application



Device-Status Table



This information is used for the disk I/O scheduling in Chapter 12



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”
 - The data might be modified after the copy is issued and before it completes
- ▶ Double buffering – two copies of the data
 - It decouples the producer of data from the consumer, thus relaxing timing requirements between them

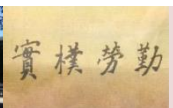
Producer → 

Consumer → 



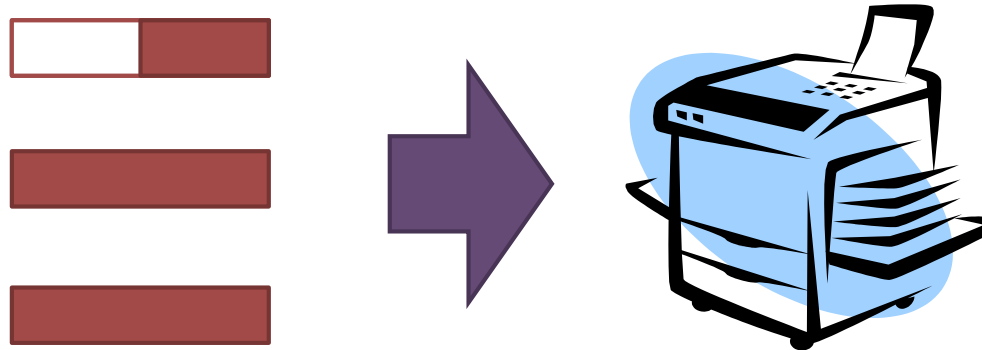
Caching

- ▶ 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
- ▶ The difference between a buffer and a cache is that 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



Spooling

- ▶ A spool is a buffer that holds multiple outputs for a device, such as a printer, that **cannot accept interleaved data streams**

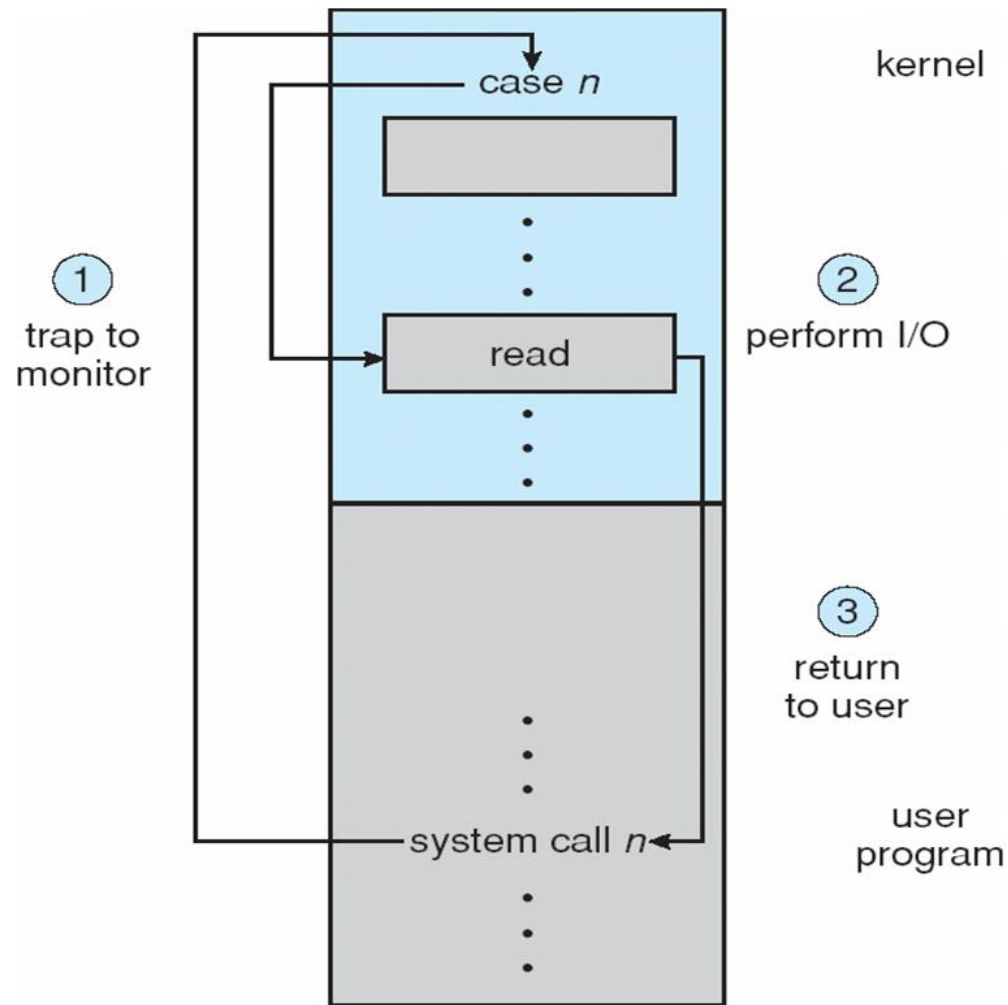


Error Handling

- ▶ OS can recover from disk read failures, device unavailable, transient write failures
 - Retry a read or write, for example
 - Some systems more advanced – SCSI
 - An additional sense code that states the category of failure
- ▶ Return an error number or code when I/O request fails
- ▶ 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



System Call for I/O



Build a System Call in Linux

- ▶ Set system call table
 - At `/arch/x86/kernel/` for older version, or at `/arch/x86/syscalls` for new version
 - Use assembly to map the name of a new system call to a number and the vector
- ▶ Set header file to define the system call number
 - For example, at `unistd.h`
 - Let the C code know the mapping information of the system call
- ▶ Define system call prototype
 - For example, at `syscall.h`
- ▶ Implement the system call
- ▶ Modify Makefile to compile the kernel with the changes

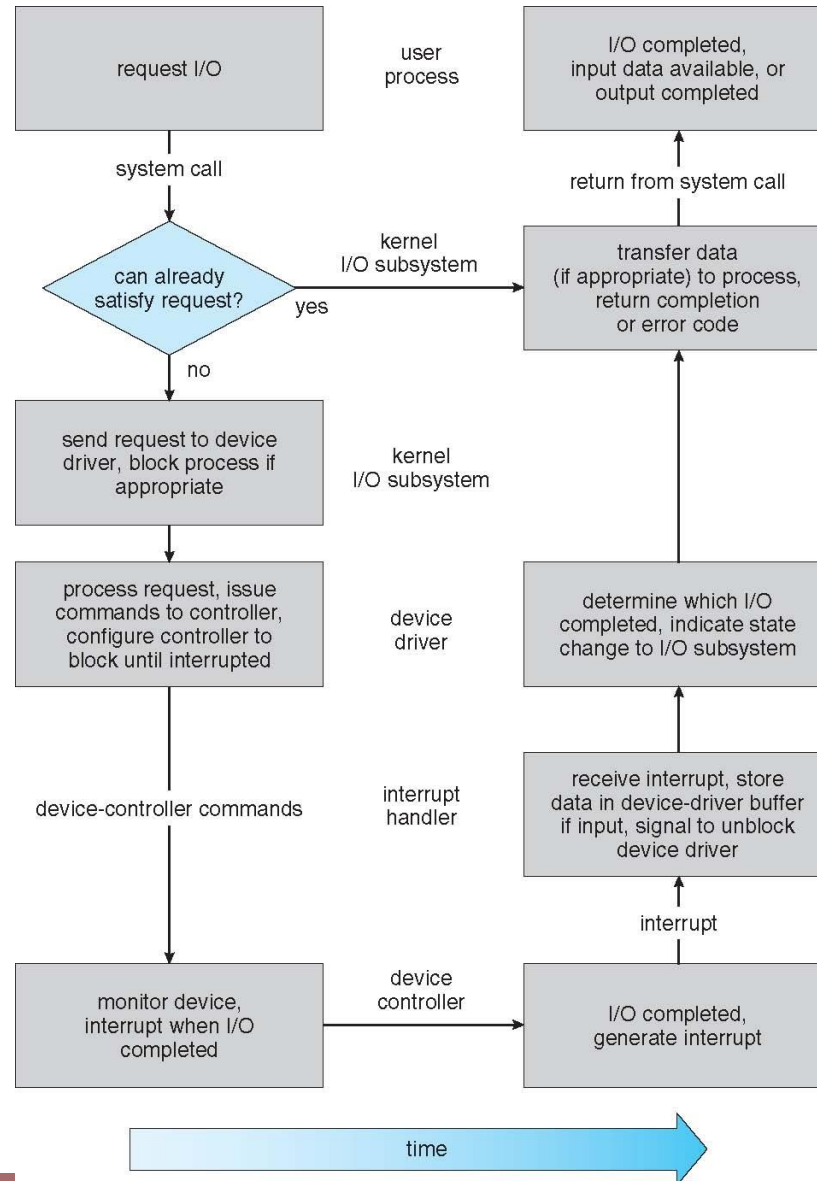


Transforming I/O Requests to Hardware Operations

- ▶ Consider reading a file from disk for a process:
 - Determine which device is holding the file
 - Mount table → which device (major number, minor number)
 - Translate name to device representation
 - File system → where is the file in the disk
 - Physically read data from disk into buffer
 - Make data available to requesting process
 - Return control to process
- ▶ Major number
 - Each device driver is identified by a unique major number
- ▶ Minor number
 - This uniquely identifies a particular instance of a device



Life Cycle of An I/O Request



I/O Performance

- ▶ I/O is 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
- ▶ Improving I/O 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

