# Silberschatz and Galvin
# Chapter 12

I/O Systems



## Topic overview

- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O requests to hardware operations
- Performance

*Topics in this chapter review and extend material discussed earlier*

# I/O hardware

- Conflicting trends in I/O devices:
  - Standardized software and hardware interfaces
  - Wide variety of hardware devices, some providing unique resources
- Device driver modules
  - Provide uniform device access interface to the I/O subsystem
  - Analogous to system calls, which provide a standard interface between application and operating system



# I/O hardware

- Common concepts
  - Port
    - connection point
  - Bus
    - common set of wires and protocol
    - daisy chain (A to B to C to computer) or shared direct access
  - Controller
    - operates port, bus, or a device
    - host adapter: separate circuit board that plugs into computer. Generally contains processor, microcode, some private memory

# I/O hardware

- Controller has one or more registers for data and control signals
- Processor communicates with controller by reading and writing these registers
  - Specified through use of I/O instructions
    - Direct I/O instructions: Device registers are separate; instructions transfer byte or word to I/O port address
    - Memory-mapped I/O: device control registers mapped into memory space of the processor (e.g., screen memory)



# I/O hardware

- I/O port registers
  - status
    - bits that are readable by host (e.g., current command has completed, byte ready to be read, device error has occurred)
  - control
    - written by host to start command or change device mode (e.g., full-duplex and half-duplex communications for serial device)
  - data-in
    - read to get input
  - data-out
    - written to send output

# I/O hardware: Polling

- Determines state of device
  - `command-ready` bit in control register
  - `busy` bit in status register
  - `error` bit in status register
- Busy-wait cycle to wait for I/O from device



# I/O hardware: Polling
# Example of writing output

- Host repeatedly reads **busy** bit until that bit becomes clear (*busy waiting* or *polling* here)
- Host sets **write** bit in **control** register and writes byte into **data-out** register
- Host sets **command-ready** bit in **control** register
- When controller detects **command-ready** bit, sets **busy** bit
- Controller reads **command** register and sees **write** bit. Reads **data-out** register to get the byte and performs I/O to the device
- Controller clears **command-ready**, **error** (command succeeded), and **busy** (controller finished)

# I/O hardware: interrupts

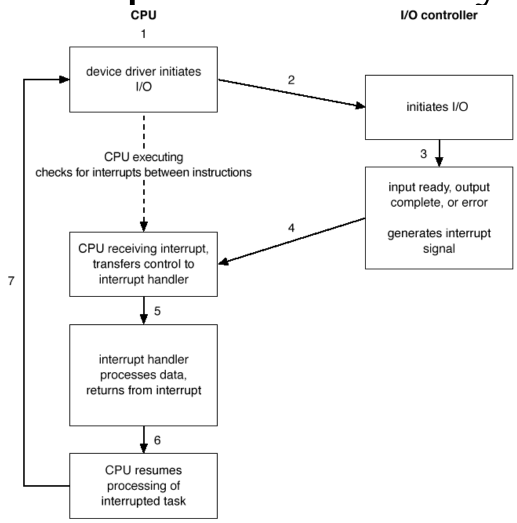
- CPU *Interrupt request line* triggered by I/O device (sensed after executing every instruction)
- Interrupt handler receives interrupts; **return from interrupt** instruction returns CPU to state prior to interrupt
- Terminology:
  - device controller *raises* interrupt
  - CPU *catches* interrupt and *dispatches* to the interrupt handler
  - Interrupt handler *clears* interrupt after servicing



# I/O hardware: interrupts

- CPUs have two interrupt request lines: maskable and nonmaskable
  - Maskable to ignore or delay some interrupts
- Interrupt vector (offset in table) to dispatch interrupt to correct handler
  - Based on priority: defers low-priority interupts to higher-priority ones
  - Some unmaskable
- Interrupt mechanism also used for exceptions (e.g., divide by zero)

# Interrupt-driven I/O cycle

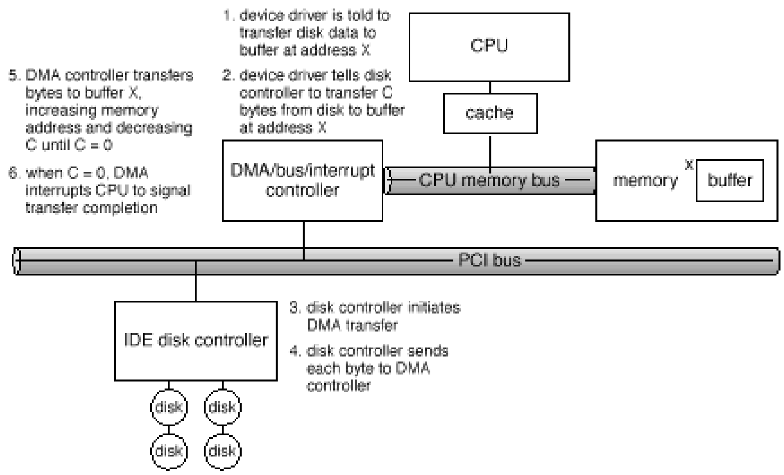





# Direct Memory Access

- Used to avoid *programmed I/O* for large data movement
  - programmed I/O: CPU transfers data to/from device one byte at a time, watching status bits, etc.
- Requires DMA controller
- Bypasses CPU to transfer data directly between I/O device and memory
  - DMA command block contains pointer to source of transfer, pointer to destination of transfer, number of bytes to be transferred
  - DMA controller manages transfer, communicating with device controller, while CPU carries out other work. Cycle stealing (DMA controller seizes memory bus) can slow down CPU.
  - DMA controller interrupts CPU at conclusion of transfer

# Steps in DMA transfer






# Application I/O interface

- Generalized device interfaces implemented by device drivers (for specific devices)
  - Abstraction, encapsulation, software layering
- Devices vary in many dimensions
  - Data transfer mode: character/block
  - Access method: sequential/random
  - Transfer schedule: synchronous/asynchronous
  - Sharing: sharable/dedicated
  - Speed of operation: latency/seek time/transfer rate/delay between operations
  - I/O direction: read/write/read-write

# Application I/O interface

- Major access conventions for device access
  - block I/O
  - character-stream I/O
  - memory-mapped file access
  - network sockets
- Escape or back-door system calls
  - transparently pass arbitrary commands to device driver
  - Unix **ioctl** (I/O ConTroL)



# Application I/O interface: Block and Character Devices

- Block devices include disk drives
  - Commands include `read`, `write`, `seek`
  - Raw I/O or file-system access (access device as a simple linear array of blocks)
  - Memory-mapped file access possible (operations are as if reading/writing to memory)
- Character devices include keyboards, mice, serial ports
  - Commands include `get`, `put` (character at a time)
  - Libraries layered on top allow line editing

# Application I/O interface: Network devices

- Varying enough from block (read-write-seek) and character (get-put) to have own interface
- Unix and Windows/NT include *socket* interface
  - Applications can create sockets, connect local socket to remote address, listen for remote applications to connect to local socket, send and receive packets over the connection
  - Separates network protocol from network operations
  - Includes `select` functionality; which sockets have a packet waiting and which have room to accept a packet to be set
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)



# Application I/O interface: Clocks and timers

- Provide current time, elapsed time, timer to trigger operation *X* at time *T*
- *programmable interval timer* used for timings, periodic interrupts
  - waits for specified time and then generates an interrupt (once or many times)
- **ioctl** (on UNIX) covers odd aspects of I/O such as clocks and timers

# Application I/O interface: Blocking and nonblocking 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
- Asynchronous - process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed either by setting a variable, with a software interrupt, with a callback routine, etc.



# Kernel I/O subsystem

- Scheduling
  - Rearranging the order of service with goal of improving overall system performance (see Chapter 13)
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness

# Kernel I/O subsystem

- Buffering - store data in memory while transfering between devices
  - To cope with device speed mismatch
    - Example: double buffering; write one while transferring other
  - To cope with device transfer size mismatch
    - Example: fragmentation and reassembly of (relatively small-sized) network packets
  - To maintain "copy semantics"
    - Example: with DMA, what happens if an application changes the memory copy before a write completes? Here, application data is copied into a kernel buffer before returning control to application



# Kernel I/O subsystem

- Caching - fast memory holding copy of data
  - Always just a copy
  - Key to performance (see Chapter 17)

# Kernel I/O subsystem

- Spooling - holds output for a device
  - If device can serve only one request at a time
  - Example: Printing
- Device reservation - provides exclusive access to a device
  - System calls for allocation and deallocation
  - May be left up to application to watch out for deadlock



# Kernel I/O subsystem

- Error handling
  - OS can recover from disk read, device unavailable, transient write failures
    - Example: read retry, network resend, etc.
  - Permanent device failures require notification
    - Most return an error number or code when I/O request fails
    - System error logs hold problem reports
    - Example: Unix **errno** variable

# Kernel I/O subsystem

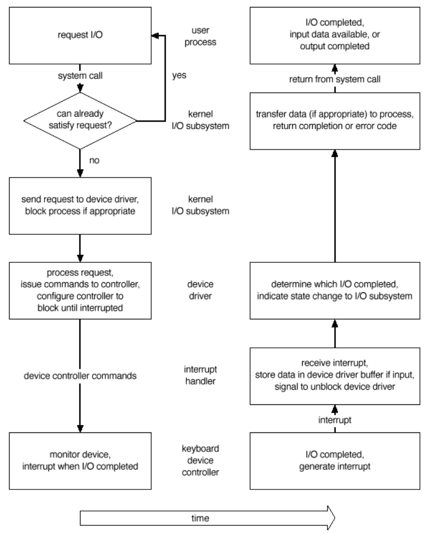
- Kernel data structures
  - Kernel keeps state info for I/O components, including open file tables, network connections, character device state
    - Many, many complex data structures to track buffers, memory allocation, ”dirty” blocks
    - Some use object-oriented methods and message passing to implement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



# Transforming I/O requests to hardware operations

- 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

# Blocking I/O request

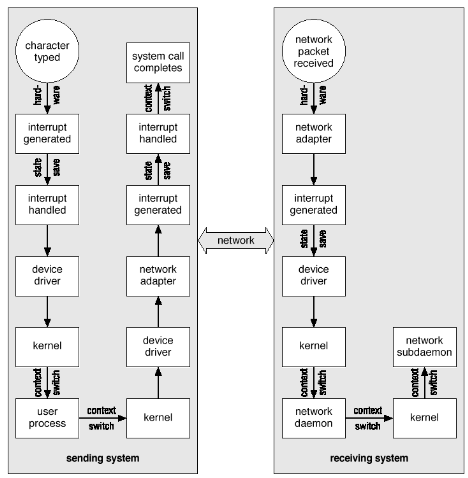





# Performance

- I/O a major factor in system performance
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts (switches necessary to execute the interrupt handler and to restore state)
  - Data copying
  - Network traffic especially stressful (see example on next slide)

# Performance: Intercomputer communications






# Improving performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Balance CPU, memory, bus, and I/O performance for highest throughput

# Implementation tradeoffs

- Application level implementation
  - more flexible, less likely to cause system crashes
  - inefficient because of context switch overhead, layers of abstraction
- Kernel implementation
  - can improve performance
  - more challenging to implement
  - greater debugging needed to avoid data corruption and system crashes
- Hardware implementation
  - highest performance
  - difficult and expensive to make further improvements or bug fixes
  - increased development time (months vs days)
  - decreased flexibility (e.g., can't necessarily take advantage of knowledge in the kernel)