Lecture Overview

- I/O devices
  - I/O hardware
  - Interrupts
  - Direct memory access
  - Device dimensions
  - Device drivers
  - Kernel I/O subsystem

I/O Device Issues

- The control of I/O devices is a major concern for OS designers
  - There are many different devices that vary greatly in function and speed
  - While software and hardware trends are towards standard interfaces, new devices are continually introduced and may not fit nicely in existing device categories

I/O Hardware

- A device communicates with the computer via a connection point, called a *port*
- If one or more devices use a common set of wire to communicate with the computer, this is a *bus*
  - A bus also requires a protocol for accessing it
  - Another form of a bus is a *daisy chain*
- A *controller* is some hardware that operates a port, bus, or device
  - Some controllers are simple, like a serial-port controller
  - Others are complex, like a SCSI-bus controller (it often uses a separate *host adapter* circuit board)

I/O Hardware

- How does the processor give commands and data to the controller
  - Controllers have one or more registers control and data signals, the processor writes to these registers
  - There are special I/O instructions to transfer data to an I/O port address over the bus
  - In some cases, it is possible to use memory-mapped I/O where the device registers are mapped into the address space of the processor
  - It is possible to use both techniques

Device I/O Port Locations

- As an example, a typical PC uses these I/O port locations

I/O address range (hex)	*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)

  - The graphics controller also uses memory-mapped I/O

I/O Device Coordination via Polling

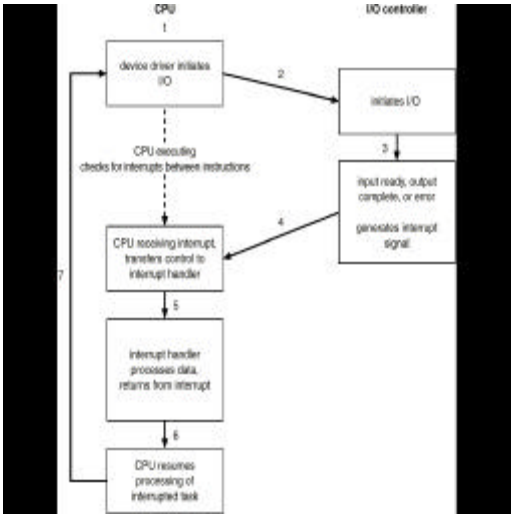
- Determines state of device using bit flags in the device controller, such as
  - `command-ready`, `busy`, `error`, `write`
- Busy-wait cycle to wait for I/O from device
  - Host repeatedly reads `busy` bit until it is clear
  - Host sets `write` bit and writes byte in `data-out` register
  - Host sets `command-ready` bit
  - Controller notices command, sets `busy` bit, does command
  - Clears `command-ready` and `busy` bits
  - Loop repeatedly
- This is busy-waiting or polling
  - Problematic when used with slow devices

I/O Device Coordination via Interrupts

- CPU hardware has a “wire” called the *interrupt request line*
  - The CPU interrupt request line is triggered by I/O devices
  - The CPU checks this line after every instruction execution
  - If line set, the CPU saves a small amount of state, then jumps to the *interrupt handler* routine
  - The interrupt handler determines cause of interrupt, performs necessary processing, and then returns the CPU to the execution state prior to the interrupt
- We say that a device controller *raises* an interrupt by *asserting* a signal on the interrupt request line; the CPU *catches* the interrupt and *dispatches* the interrupt handler which *clears* the interrupt

I/O Device Coordination via Interrupts

- Interrupt-driven I/O cycle



I/O Device Coordination via Interrupts

- Interrupt-driven I/O allows the CPU to respond to asynchronous I/O events while still doing other work
- Many sophisticated interrupt handling schemes include (via a hardware *interrupt controller*)
  - Ability to defer interrupt handling during critical processing
  - Efficient way to dispatch to proper handler
  - Multilevel interrupts based on priority
- Interrupts may be either *maskable* (i.e., it can be deferred during critical regions) or *non-maskable* (i.e., it cannot be deferred)
- The *interrupt vector* is held in a specific memory location and is a table of interrupt handlers
  - The index into the vector corresponds to the device that raised the interrupt

I/O Device Coordination via Interrupts

- A Pentium vector table

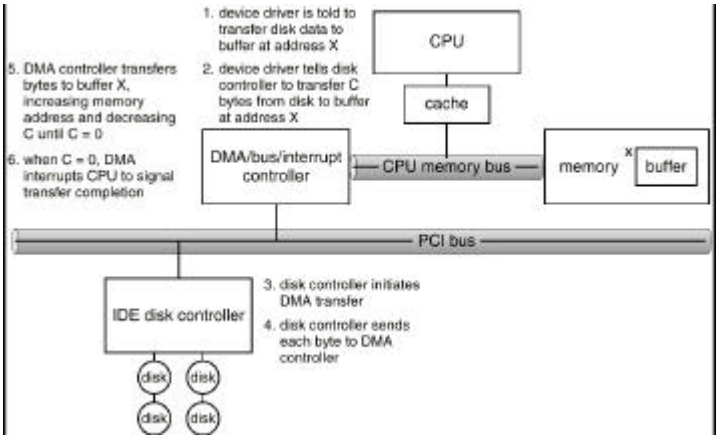
vector number	***description***	***vector number***	***description***
0	divide error	11	segment not present
1	debug exception	12	stack fault
2	null interrupt	13	general protection
3	breakpoint	14	page fault
4	[overflow]	15	(reserved)
5	range exception	16	floating-point error
6	invalid opcode	17	alignment check
7	device not available	18	machine check
8	double fault	19-31	(reserved)
9	(reserved)	32-255	maskable interrupts
10	invalid TSS		

  - For Linux, 32 to 47 are for IRQs and of the remaining, only vector number 128 (0x80) is used for system calls

Direct Memory Access (DMA)

- The term *programmed I/O* refers to using the CPU to monitor and process the low-level receiving of data from I/O devices
- DMA is used to avoid programmed I/O for large data movement
- Requires DMA controller
- Bypasses CPU to transfer data directly between I/O device and memory
  - This does lead to memory *cycle stealing* from the CPU, but on average it is not problematic
  - DMA can be implemented to use physical or virtual addresses

DMA Transfer



Main I/O Concepts Review

- A bus
- A controller
- An I/O port and its registers
- Handshaking relationship between the host and a device controller
- Execution of handshaking in a polling loop or via interrupts
- Off-loading this work to a DMA controller for large transfers

Device Dimensions

- Character stream or block
- Sequential or random access
- Synchronous or asynchronous
- Sharable or dedicated
- Speed of operation
- Read/write, read only, or write only

Device Dimensions

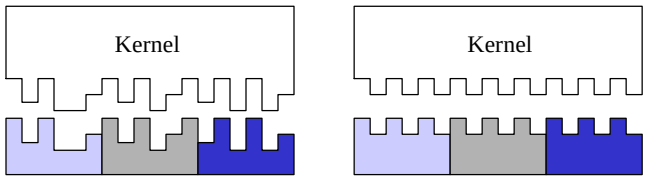
- Character stream device
  - Commands include `get`, `put`
  - Libraries layered on top allow line editing
- Block device
  - Commands include `read`, `write`, `seek`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Network device
  - Different enough have own interface
  - Unix and Windows/NT include socket interface
    - Separates network protocol from network operation
    - Includes `select` functionality
  - Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

Device Dimensions

- Clocks and timers
  - Provide current time, elapsed time, timer
  - Programmable interval time used for timings, periodic interrupts
- `ioctl` (on UNIX) covers odd aspects of I/O such as clocks and timers
  - Essentially an *escape* or *back-door* system call to directly access device driver functionality

Application I/O Interface

- Details I/O device usage differs among devices
- The OS abstracts away differences by defining a few general kinds of I/O devices
  - Essentially, it defines interfaces
  - An implementation of one of these interfaces is called a *device driver*, which encapsulates a specific device's details
  - Benefits both the OS writer and the device manufacturer



Specific devices Devices using driver interface

Blocking and Non-blocking I/O

- *Blocking* - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- *Non-blocking* - I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multithreading
  - 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

Kernel I/O Subsystem

- Scheduling
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
- 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”

Kernel I/O Subsystem

- Caching - fast memory holding copy of data
  - Always just a copy
  - Key to performance
- Spooling - hold output for a device
  - If device can serve only one request at a time (e.g., printing)
- Device reservation - provides exclusive access to a device
  - System calls for allocation and deallocation
  - Watch out for deadlock

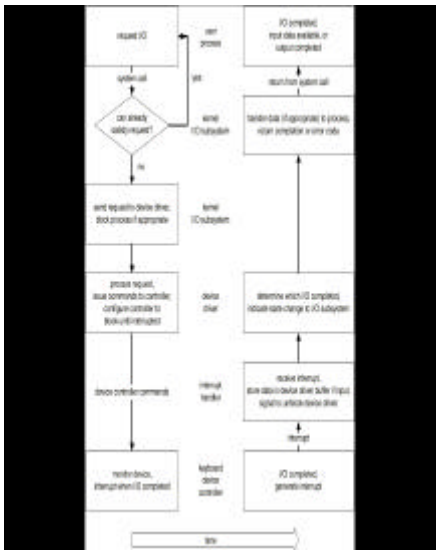
Kernel I/O Subsystem

- Error handling
  - OS can recover from disk read, device unavailable, transient write failures
  - Most return an error number or code when I/O request fails
  - System error logs hold problem reports
- Kernel data structures
  - Kernel keeps information about 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

I/O Request Life Cycle

- 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

I/O Request Life Cycle



I/O 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
- To improve 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

I/O Performance

- Performance also relates to where device support is implemented
  - Application code is the slowest, but offers many advantages when building a new an untested device
  - Kernel code is faster because it reduces context switching and provides access to internal structures, but the device must be stable
  - Hardware code is for optimal performance, but device must be really stable because it is difficult to change