13. I/O Devices

CS 4352 Operating Systems

Adopted from slides of Drs. Silberschatz, Huang, & Bryant

I/O Devices

- I/O is critical to computer system to interact with systems
  - Incredible variety of I/O devices
    - Storage
    - Transmission
    - Human-interface
- I/O 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

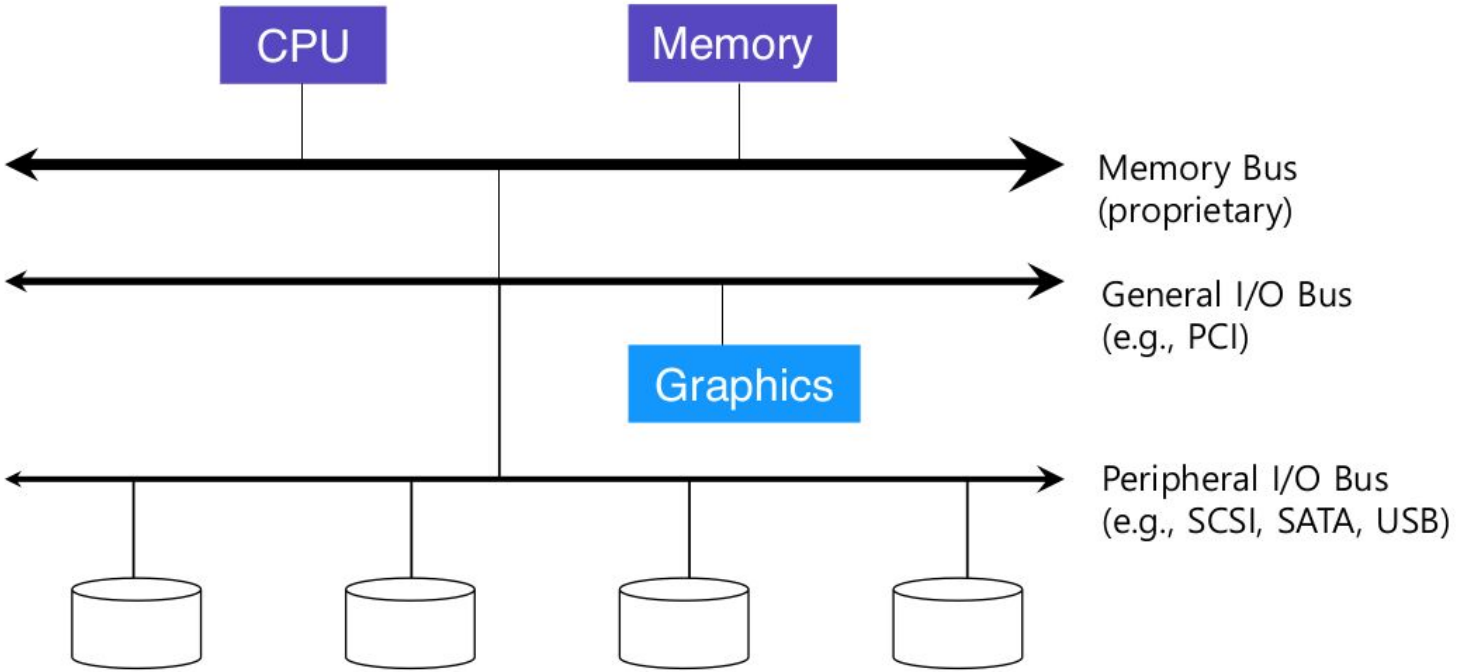
Characteristics of I/O Devices

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

Issues

- How should I/O be integrated into systems?
- What are the general mechanisms to operate them?
- How can we improve efficience?

Computer Organization

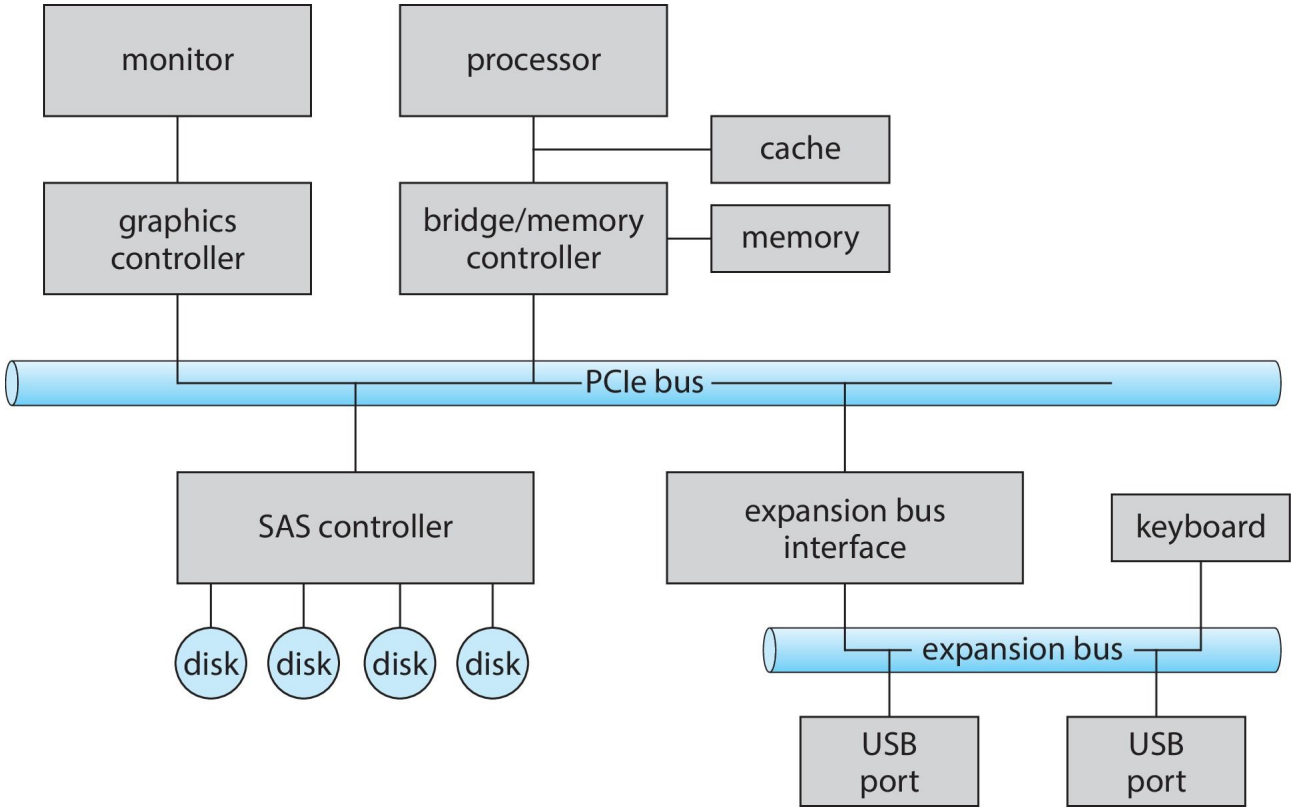


Buses

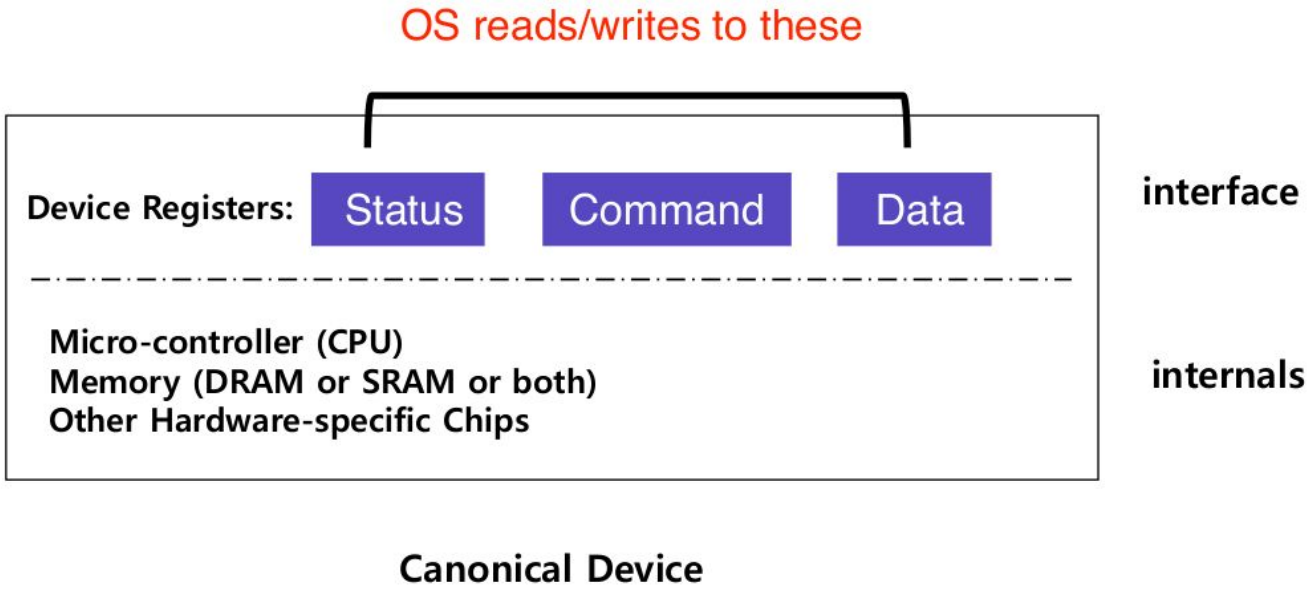
- Buses
  - Communication channels to enable information between CPU(s), RAM, and I/O devices

- I/O buses
  - Channels that connect a CPU to I/O devices.
  - I/O bus is connected to I/O device by three hardware components: I/O ports, interfaces and device controllers.

I/O Bus Example: PCIe



Canonical I/O Device



Hardware Interface of Canonical Device

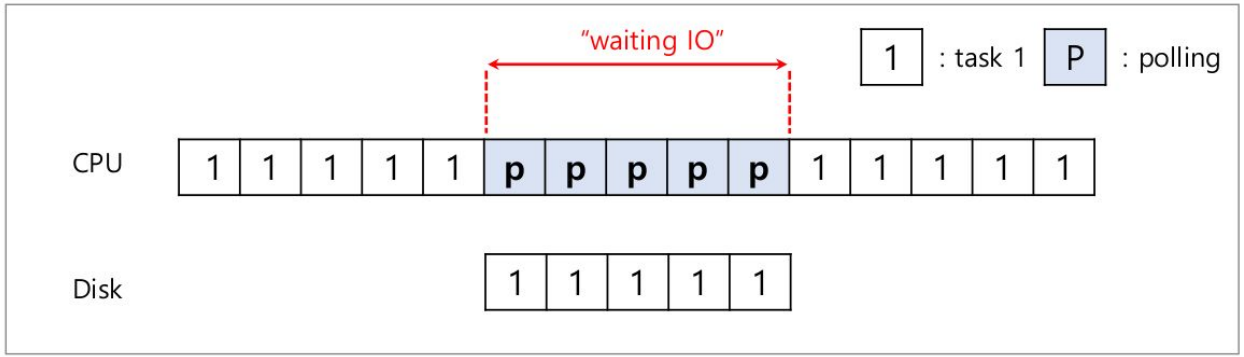
- Status registers
  - See the current status of the device
- Control registers
  - Tell the device to perform a certain task
- Data registers
  - Pass data to the device, or get data from the device
- By reading or writing the above three registers, the OS controls device behavior

Typical Interaction Example

```
while ( STATUS == BUSY)
    ; //wait until device is not busy
write data to data register
write command to command register
    Doing so starts the device and executes the command
while ( STATUS == BUSY)
    ; //wait until device is done with your request
```

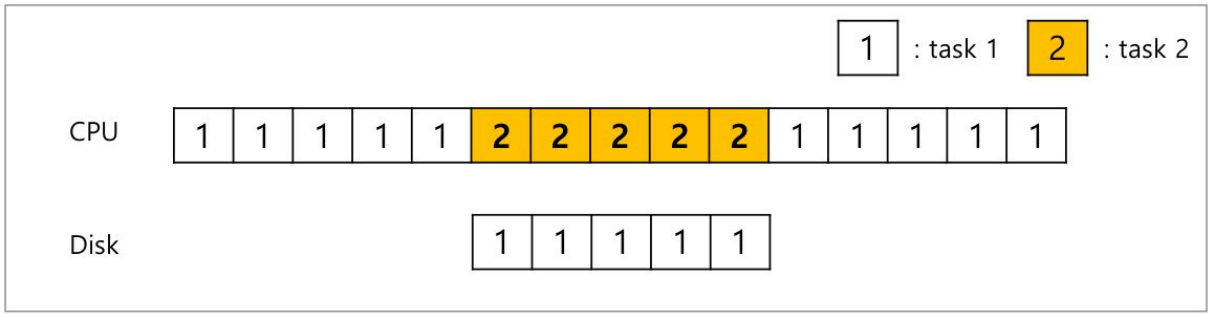
Polling

- OS waits until the device is ready by repeatedly reading the status register
  - Simple and working
  - However, it wastes CPU time just waiting for the device
    - Switching to another ready process is better utilizing the CPU



Interrupts

- Put the I/O request process to sleep and context switch to another
- When the device is finished, notify the OS by interrupt
  - It allows CPU to be properly utilized



x86 Processor Interrupt-Vector Table

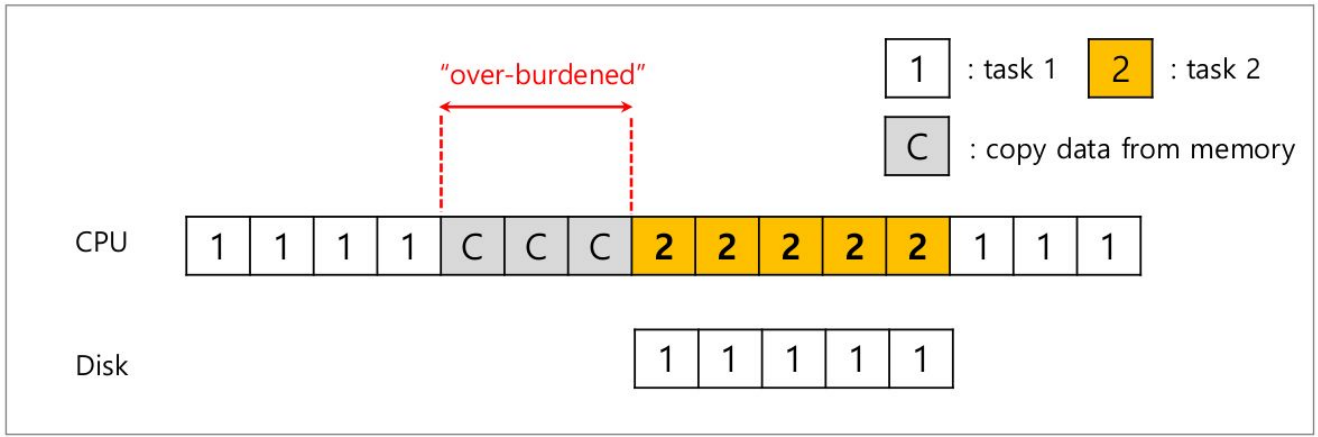
vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
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

Polling v.s. Interrupts

- However, “interrupt is not always the best solution”
  - If, device performs very quickly, interrupt will “slow down” the system
    - Interrupts have overhead (context switch)
    - Interrupt handlers have high priority
      - In worst case, can spend 100% of time in interrupt handler and never make any progress – livelock
- If a device is fast, polling is best
- If a device is slow, interrupts are better

Programmed I/O

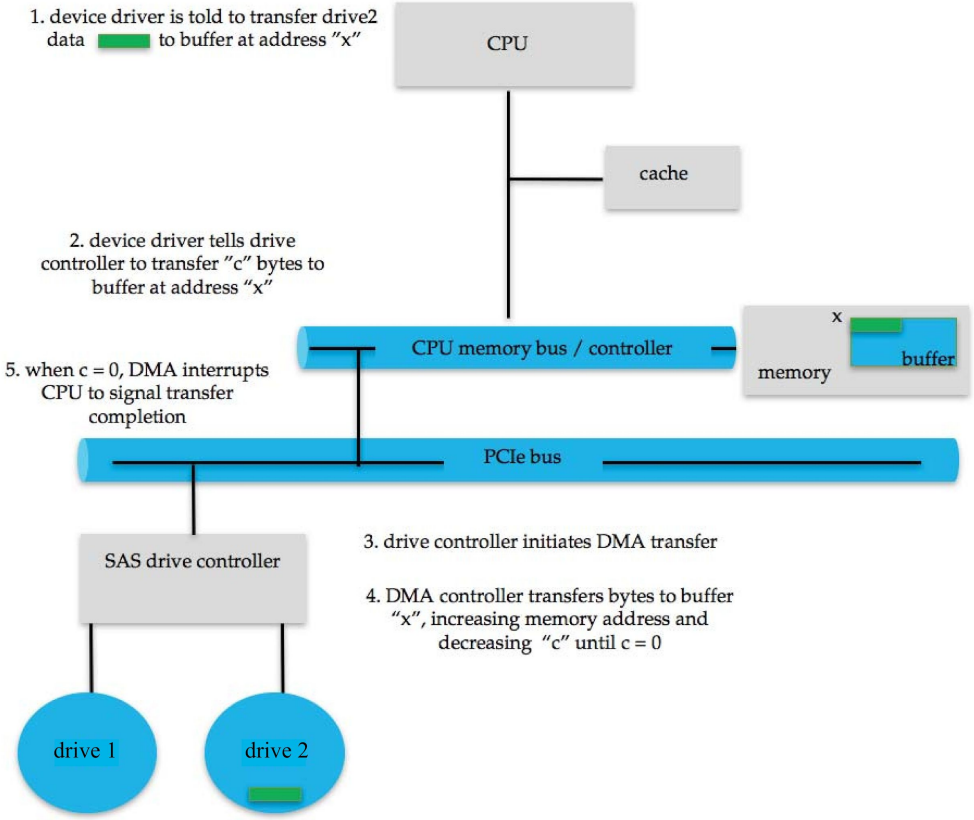
- Programmed I/O is a method of data transmission
  - Each data item transfer involves the CPU
  - CPU wastes a lot of time in copying a large chunk of data from memory to the device



Direct Memory Access (DMA)

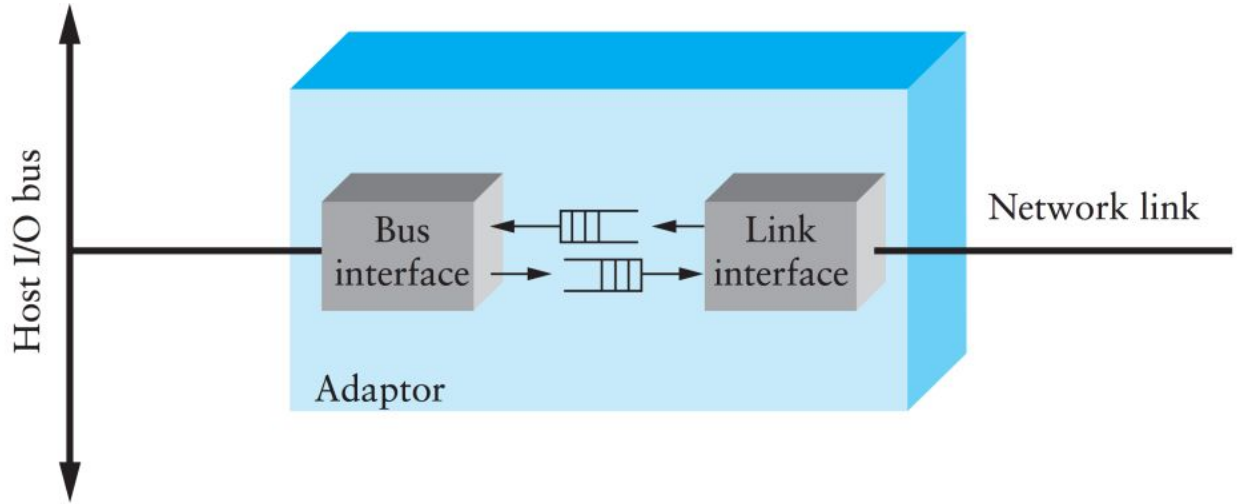
- Idea: only use CPU to transfer control requests, not data
  - Used to avoid programmed I/O (one byte at a time) for large data movement
  - Requires DMA controller
  - Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location of command block to DMA controller
  - Bus mastering of DMA controller – grabs bus from CPU
    - Cycle stealing from CPU but still much more efficient
  - When done, interrupts to signal completion

Steps to Perform DMA Transfer



Example: Network Interface Card

- Link interface talks to wire/fiber/antenna
  - FIFOs on card provide small amount of buffering
- Bus interface logic uses DMA to move packets to and from buffers in main memory



Addressing I/O Devices

- How the OS communicates with the I/O device?
  - I/O instructions
    - A way for the OS to send bits directly to specific device registers
    - E.g., in and out instructions on x86
    - A separate I/O address space
  - Memory-mapped I/O
    - Device data and command registers mapped to processor address space
    - The OS load (to read) or store (to write) to the device instead of main memory
    - 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)

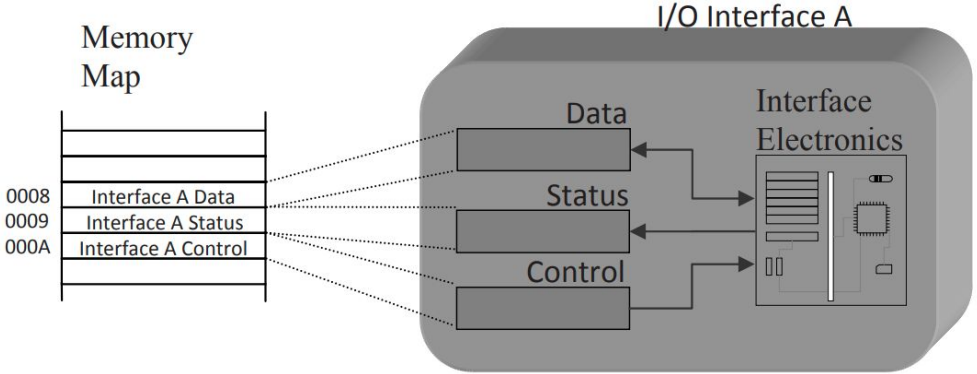
x86 I/O Instructions

Opcode	Mnemonic	Description
E4 ib	IN AL,imm8	Input byte from imm8 I/O port address into AL.
E5 ib	IN AX,imm8	Input byte from imm8 I/O port address into AX.
E5 ib	IN EAX,imm8	Input byte from imm8 I/O port address into EAX.
EC	IN AL,DX	Input byte from I/O port in DX into AL.
ED	IN AX,DX	Input word from I/O port in DX into AX.
ED	IN EAX,DX	Input doubleword from I/O port in DX into EAX.

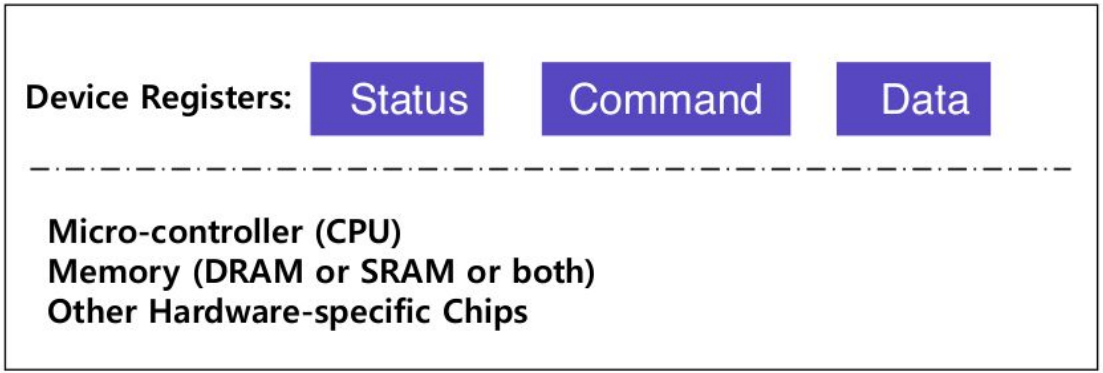
Opcode	Mnemonic	Description
E6 ib	OUT imm8, AL	Output byte in AL to I/O port address imm8.
E7 ib	OUT imm8, AX	Output word in AX to I/O port address imm8.
E7 ib	OUT imm8, EAX	Output doubleword in EAX to I/O port address imm8.
EE	OUT DX, AL	Output byte in AL to I/O port address in DX.
EF	OUT DX, AX	Output word in AX to I/O port address in DX.
EF	OUT DX, EAX	Output doubleword in EAX to I/O port address in DX.

Memory-Mapped I/O (MMIO)

- I/O instructions have limits
  - Instruction format restricts what registers you can use
  - Only allows 2^16 different port numbers on x86
  - Per-port access control turns out not to be useful
- When memory and I/O share the same address space, the CPU is said to use memory-mapped I/O
  - We treated the I/O as though it is a memory location



Protocol Variants

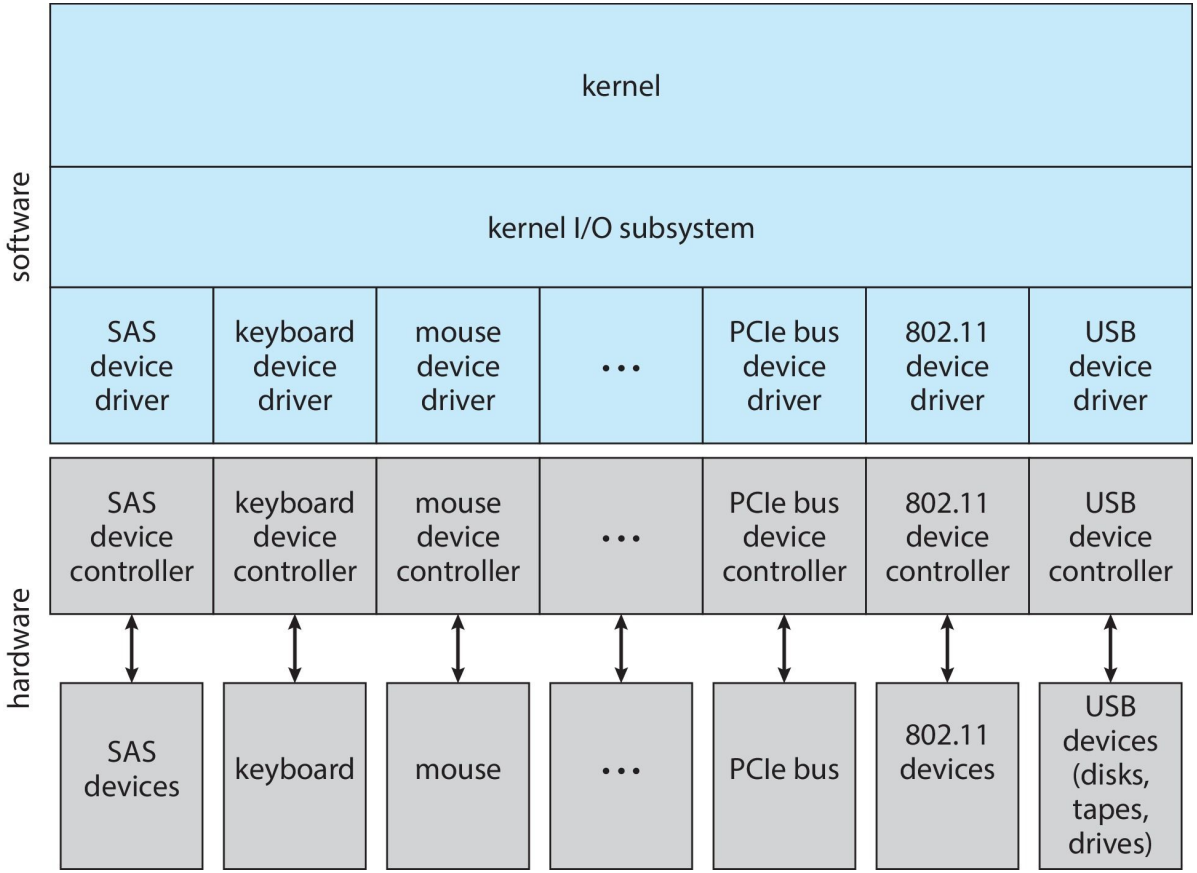


- Polling vs. interrupts
- Programmed I/O vs. DMA
- Special instructions vs. memory-mapped I/O

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

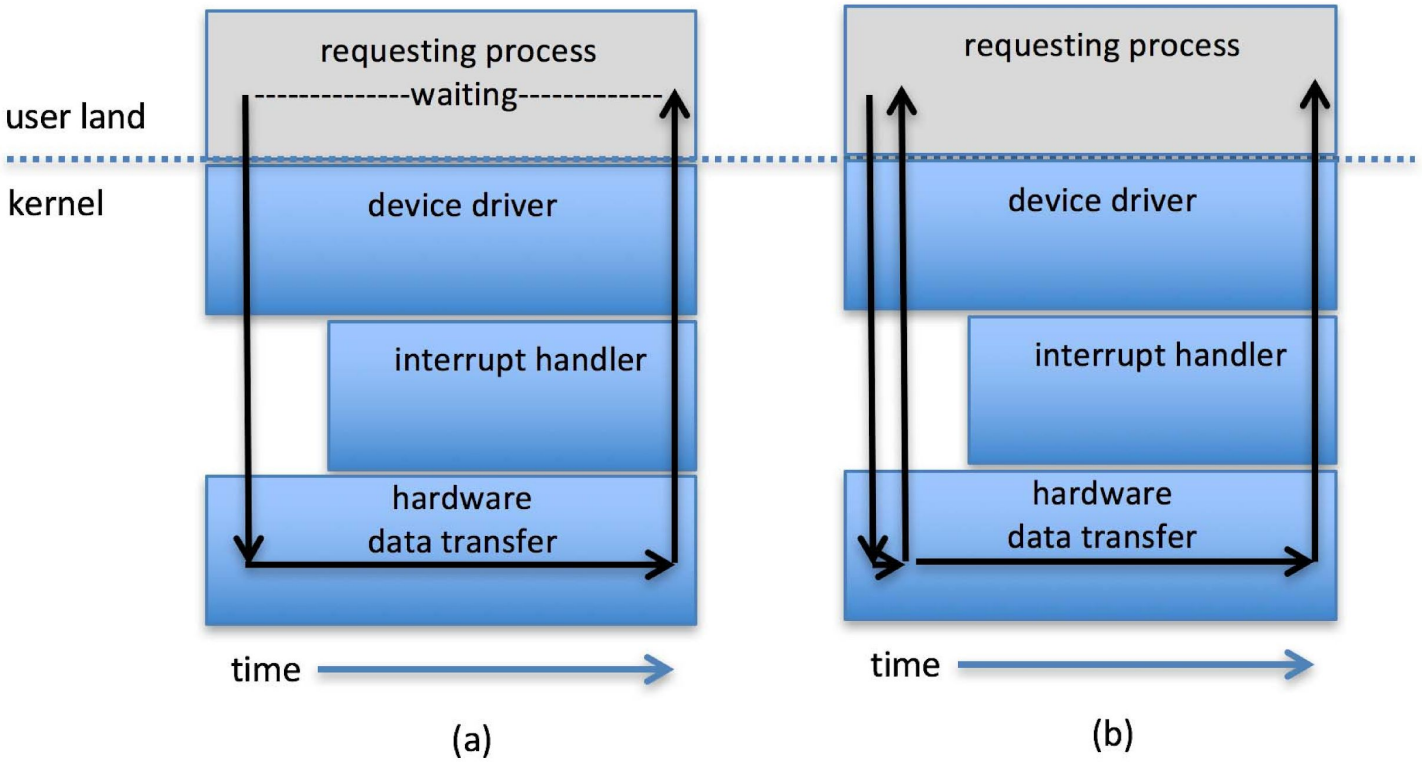
A Kernel I/O Structure



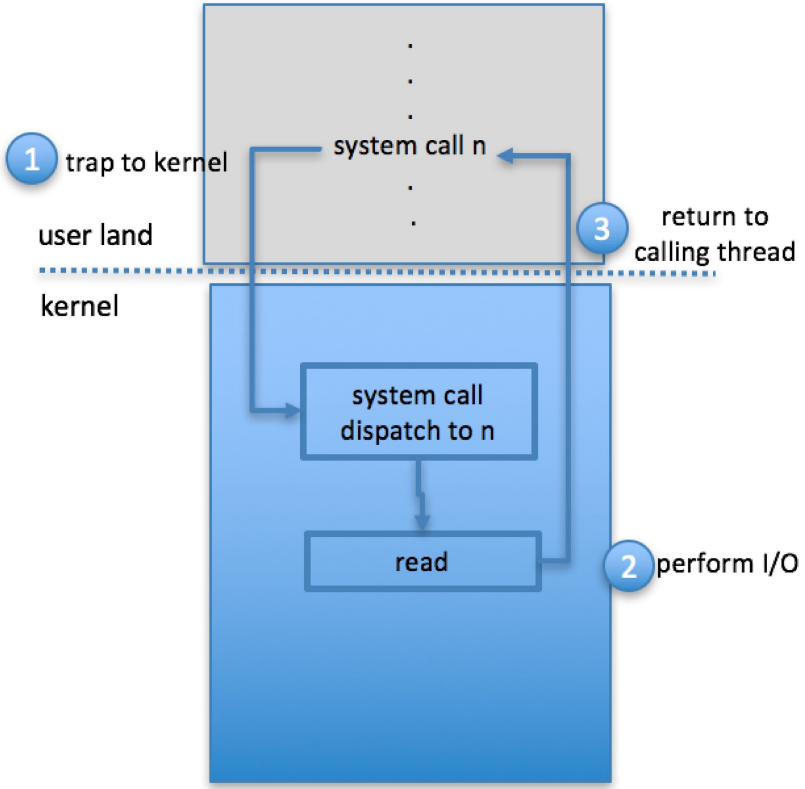
Characteristics of I/O Devices (Cont.)

- Subtleties of devices handled by device drivers
- Broadly I/O devices can be grouped by the OS into
  - Character devices
  - Block devices
  - Network interface cards
- For direct manipulation of I/O device specific characteristics, there is usually an escape/back door
  - Unix/Linux ioctl() system 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

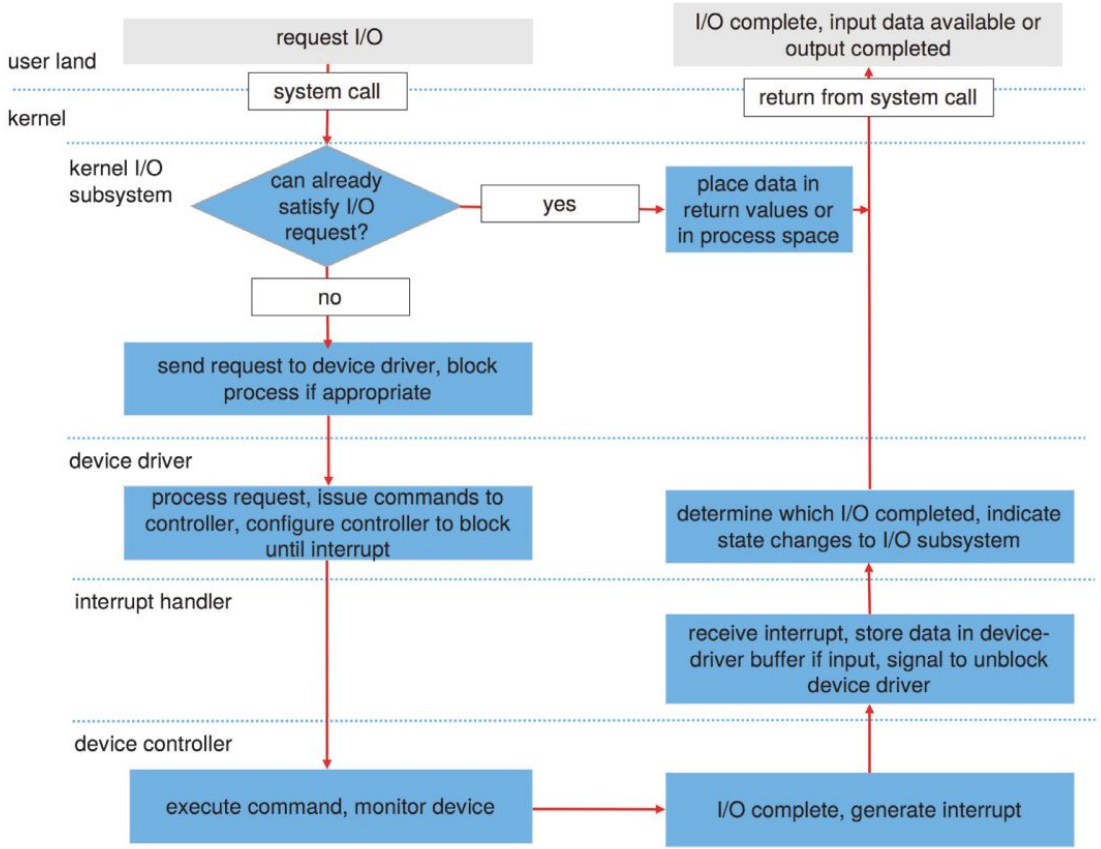
Two I/O Methods



Use of a System Call to Perform I/O



Life Cycle of An I/O Request



Next Lecture

We will look at massive-storage devices specifically