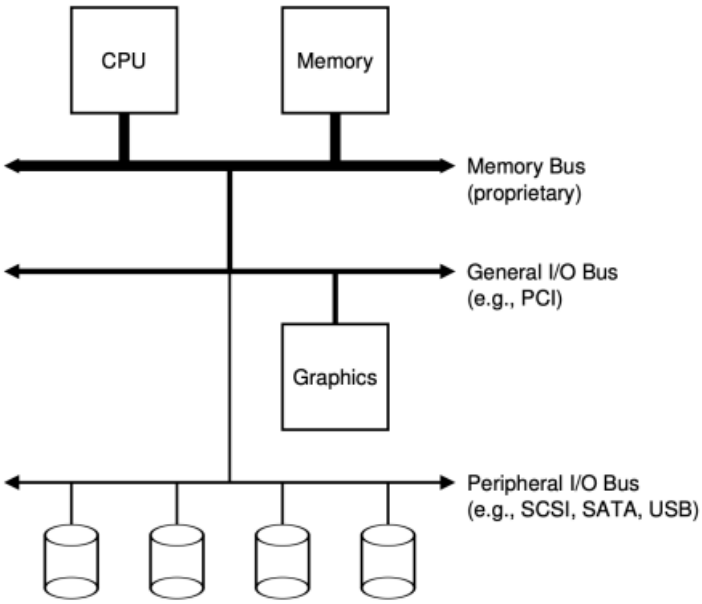
Lecture10 I/O

1. I/O Management

Challenges of I/O management

- Diverse devices: each device is slightly different
- Unreliable device: media failures and transmission errors
- Unpredictable and slow devices

A Classic View of Computer System

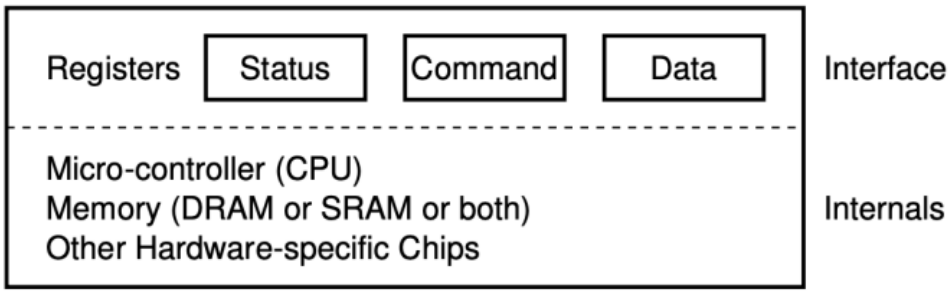


- A **single CPU** attached to the **main memory** of the system via memory bus or interconnect
- Some devices (**graphics** and some other **higher-performance I/O** devices) are connected to the system via a general **I/O bus** (e.g., PCI)
- Finally, a **peripheral bus**, such as SCSI, SATA, or USB, connects slow devices to the system, including **disks, mice, and keyboards**

A Canonical View of Devices

- **Interface**: The **hardware interface** a device present to the rest of the system
    - **Status registers**: check the current status of the device
    - **Command register**: tell the device to perform a certain task
    - **Data register**: pass data to the device or get data from the device

- **Internal structures**: Implementation of the abstract of the device



2. I/O Hardware

Memory-Mapped I/O

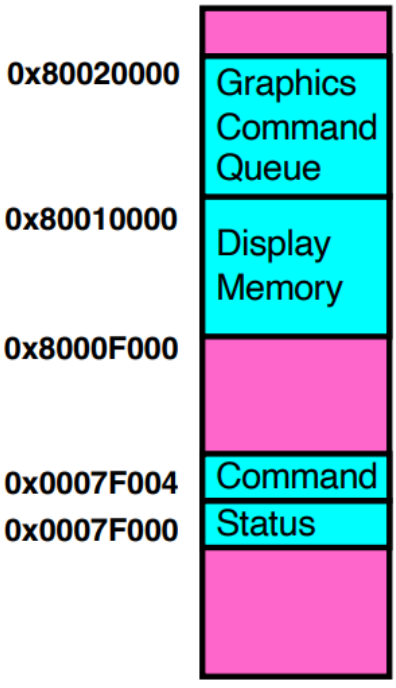
How does the processor give commands and data to a controller to accomplish an I/O transfer?

- The controller has one or more registers for data and control signals
- the processor communicate with the controller by reading and writing bit patterns in theseregisters

Memory-mapped I/O

- the device-control **registers** are mapped into the **address space** of the processor
- The CPU executes I/O requests using the standard data transfer instructions to read and write the device-control registers
- I/O accomplished with **load and store instructions**
- I/O protection with **address translation**

Memory-Mapped Display Controller



- Hardware maps control **registers** and display memory into **physical address space**
  - Addresses set by Hardware jumpers or at boot time
- Simply writing to **display memory** (also called the “frame buffer”) changes image on screen
  - Addr: 0x8000F000 — 0x8000FFFF
- Writing graphics description to cmd queue
  - Say enter a set of triangles describing some scene
  - Addr: 0x80010000 — 0x8001FFFF
- Writing to the command register may cause on-board graphics hardware to do something
  - Say render the above scene
  - Addr: 0x0007F004

Polling (Basic I/O)

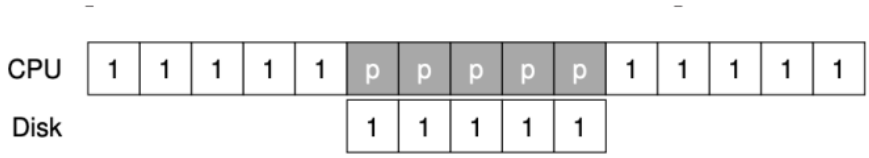
Write a byte of data to device

1. **OS waits** until the **device is ready** to receive a command by **repeatedly reading the status register**
2. **OS sends some data** down to the **data register**
3. **OS writes a command** to the **command register**
4. **OS waits for the device to finish** by again **polling** it in a loop, waiting to see if it is finished

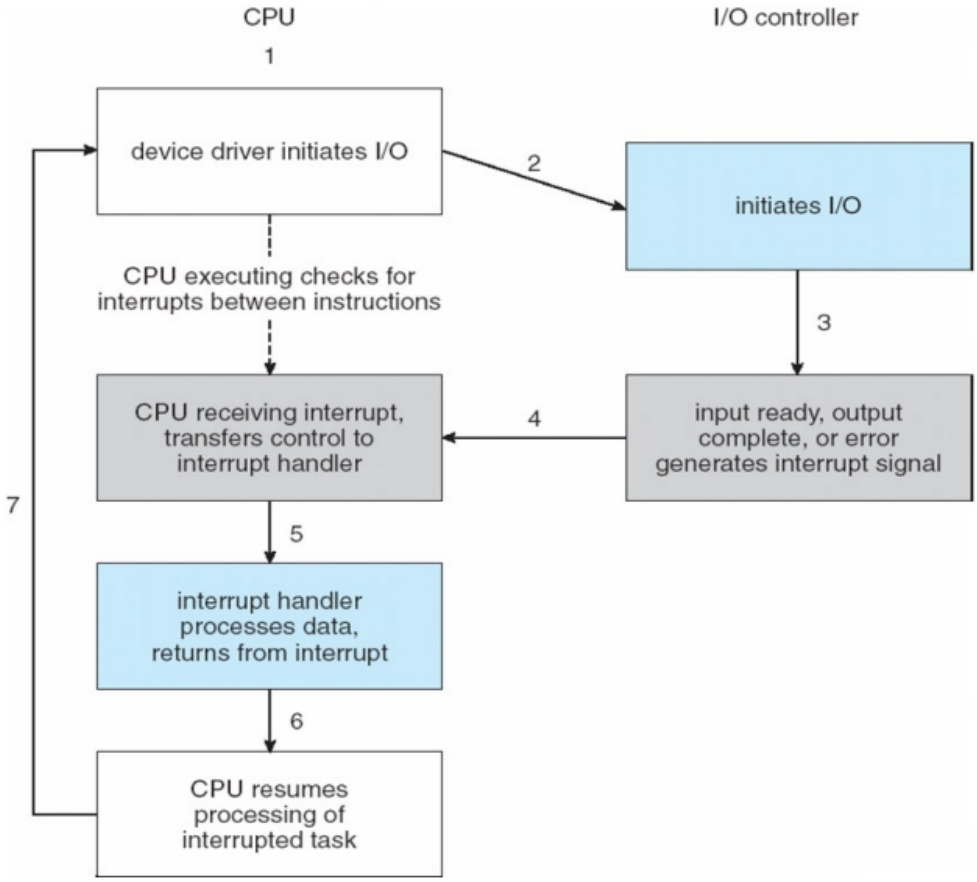
```
while(STATUS == BUSY){
    ; // wait until device is not busy
}
write data to DATA resister;
write command to COMMAND register; // starts the device and executes the command
while(STATUS == BUSY){
    ; // wait until device is done with your requirement
}
```

Issues of Polling

- Polling: frequent checking the status of I/O devices
- Polling is inefficient and inconvenient
  - Polling **wastes CPU time waiting for slow devices** to complete its activity
  - If CPU switches to other tasks, data may be overwritten
    - e.g., keyboard data overflow the buffer



Interrupts (Efficient I/O)



- Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task
- When the device is finally finished with the operation, it will raise a **hardware interrupt**, causing the **CPU to jump into the OS** at a predetermined **interrupt handler**

Basic Interrupt Mechanism

- The basic interrupt mechanism works as follows
  - The CPU hardware has a wire called the **interrupt-request line**, that CPU scenes after executing every instruction
  - When CPU detects that a **controller has asserted a signal** on the interrupt-request line, the CPU performs a **state save** and jumps to the **interrupt-handler routine** at a fixed address in the memory
- We say that
  - the device controller **raises** an interrupt by asserting a signal on the interrupt request line

    - the CPU **catches** the interrupt and **dispatches** it to the interrupt handler
    - and the handler **clears** the interrupt by servicing the device.

Sophisticated Interrupt-handling Features

Provided by **interrupt-controller hardware**

- **defer interrupt handling** during critical processing
- **dispatch to the proper interrupt handler for a device** without first polling all the devices to see which one raised the interrupt
- **multilevel interrupts**, so that the operating system can **distinguish between high- and low-priority interrupts** and can respond with the appropriate degree of urgency when there are multiple concurrent interrupts
- **get the operating system's attention directly** (separately from I/O requests), for activities such as page faults and errors such as division by zero. (get)

Nonmaskable and maskable interrupt

- Most CPUs have two interrupt request lines
    - **nonmaskable**: reserved for events such as unrecoverable memory errors
    - **maskable**: can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted

Interrupt Vectors

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

- The interrupt mechanism accepts an **address**: a **number** that selects a specific **interrupt-handling routine** from a small set
    - this address is an offset in a table called the **interrupt vector**, contains the **memory addresses** of specialized interrupt handlers
- computers have more devices than they have address elements in the interrupt vector
    - use **interrupt chaining**, in which each element in the interrupt vector points to the **head of a list of interrupt handlers**
    - Interrupt handlers on the corresponding chain are called one by one
    - The **size** of the **interrupt table** (i.e., number of interrupt vectors) and **length** of **interrupt chains** are results of system design trade-off

Interrupt Priority Levels

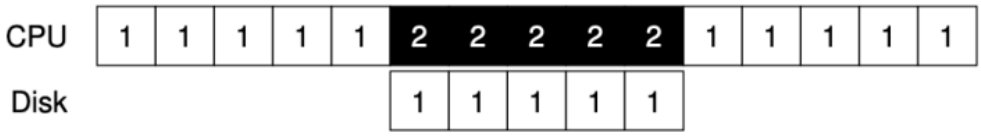
enable the CPU to **defer the handling of low-priority interrupts** without masking all interrupts and **make it possible for a high-priority interrupt to preempt** the execution of a low-priority interrupt

Interacts with the interrupt mechanism

- **boot time**: OS probes the hardware buses to determine what device are present and installs the corresponding interrupt handlers into the interrupt vector
- **I/O**: the various device controllers raise interrupts when they are ready for service
- **handle exceptions**
    - dividing by zero
    - accessing a protected or nonexistent memory
    - attempting to execute a privileged interaction from user mode
- **virtual memory paging (page fault)**
    - suspends the current process
    - jump to the page-fault handler in the kernel
    - save the state of the process
    - move the process to the WAIT queue
    - performs page-cache management
    - schedules an I/O operation to fetch the page
    - schedules another process to resume execution
    - returns from the interrupt
- **system calls: software interrupt / trap**
    - a process executes the trap instruction
    - the interrupt hardware saves the state of the user code, switches to kernel mode, and dispatches to the kernel routine or thread that

implements the requested service
    - **trap** is given a relatively **low interrupt** priority compared with those assigned to device interrupts

Polling or Interrupt



- Polling works better for **fast devices**
    - Data fetched with first poll
- Interrupt works better for **slow devices**
    - Context switch is expensive
- Hybrid approach if speed of the device is not known or unstable
    - Polls for a while
    - Then use interrupts

Direct-Memory Access (DMA)

Programmed I/O Problem

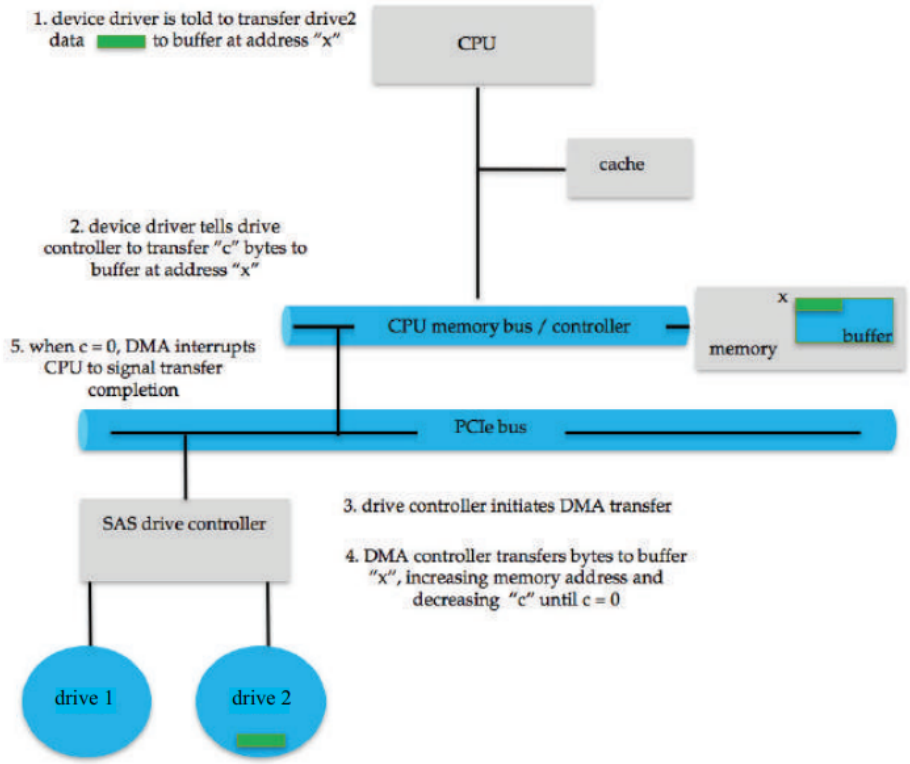
A general-purpose processor to watch status bits and to feed data into a controller register one byte at a time

- for a device does large transfer, it is a waste to use an expensive general-purpose processor

DMA is used to **avoid programmed I/O** for large data movement

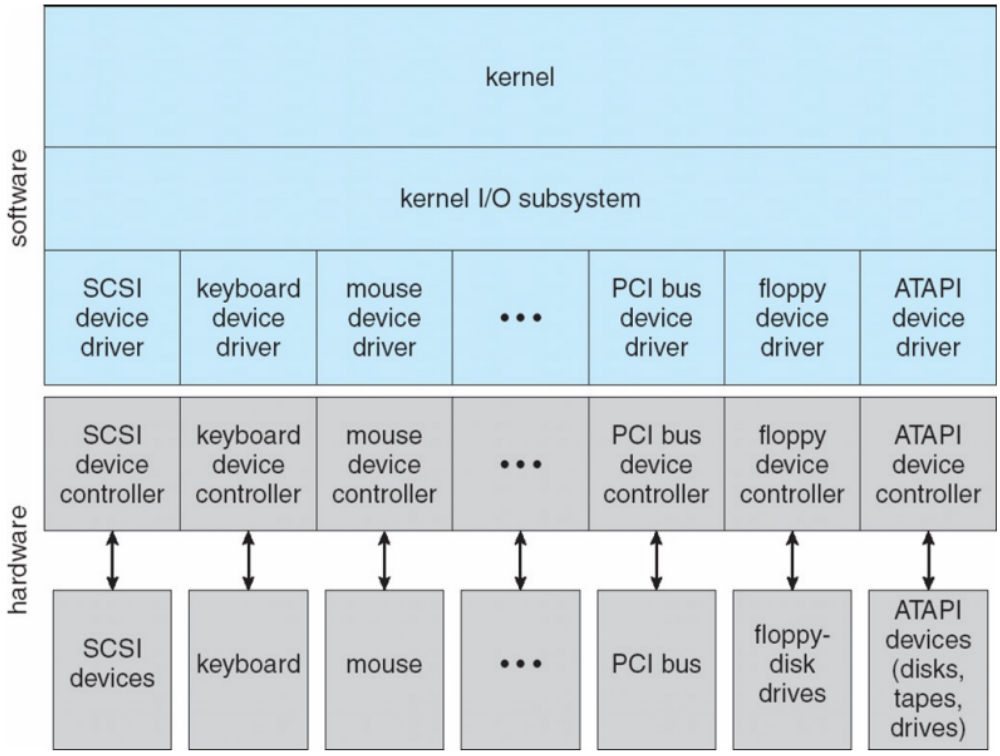
- Programmed I/O (PIO): when CPU is involved in data movement
- PIO consumes CPU time
- bypasses CPU to transfer data directly between **I/O device and memory**

Steps

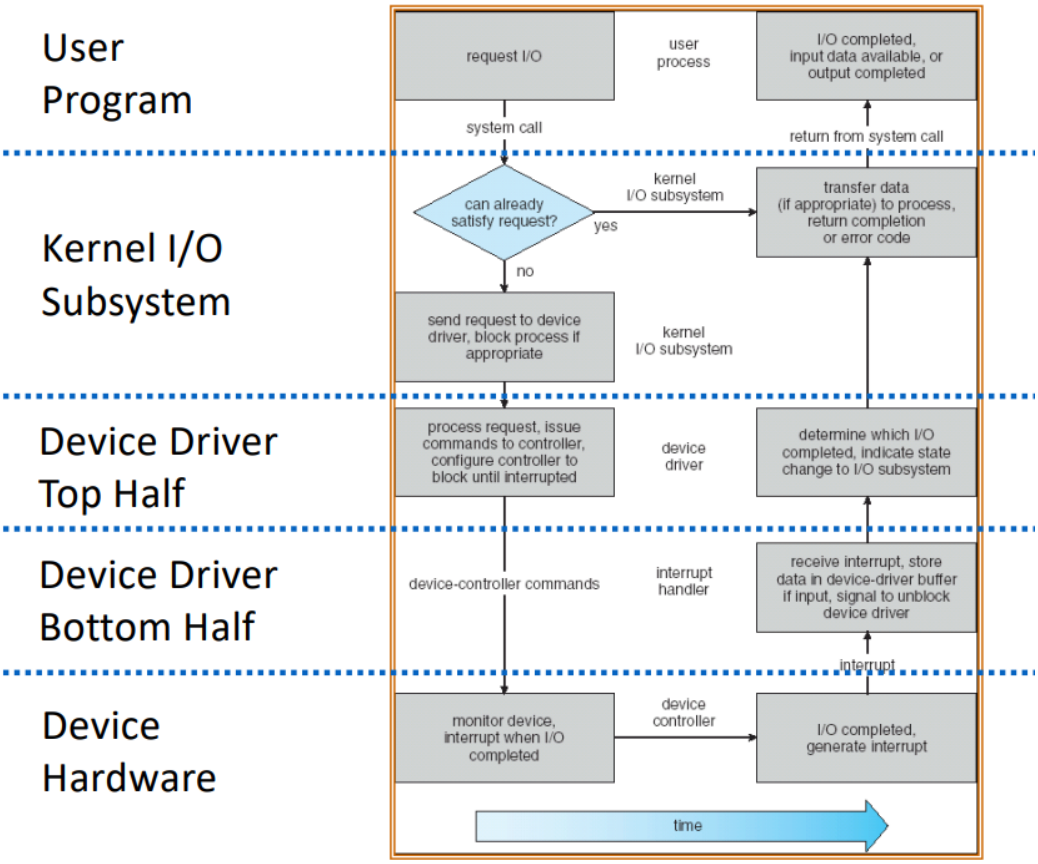


- the host writes a **DMA command block** into **memory**, it contains
  - a pointer to the source of a transfer
  - a pointer to the destination of the transfer
  - a count of the number of bytes to be transferred
  - **allows multiple transfers** to be executed via a single DMA command
- the **CPU** writes the address of this command block to the **DMA controller**, and goes on with other work
- the DMA controller proceeds to **operate the memory bus directly**, placing the address on the bus to perform transfers without the help of the main CPU
- when the entire transfer is finished, the **DMA controller interrupts the CPU**

3. Kernel I/O Structure



Life Cycle of An I/O Request



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**
- Devices vary in many dimension

Character-stream	Block	
Sequential	Random-access	
Synchronous	Asynchronous	
Sharable	Dedicated	
Speed of operation		
read-write	read only	write only

Device Drivers

- **Device Driver**: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
  - **Top half**: accessed in call path from system calls
    - implements a set of standard, cross-device calls like open(), close(), read(), write(), ioctl(), strategy()
    - This is the **kernel's interface** to the device driver
    - Top half will start I/O to device, may put thread to sleep until finished
  - **Bottom half**: run as **interrupt** routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete

I/O Devices

- Subtleties 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 from user-space applications
    - Unix `ioctl()` call to send arbitrary bits to a **device control register** and data to **device data register**

Block and Character Devices

- **Block** devices include **disk drives**
    - Commands include read, write, seek
    - Raw I/O, allows direct file-system access
        - Applications (e.g., database) do not need buffering or locking by filesystems
    - Memory-mapped file access possible
        - File mapped to virtual memory via demand paging
    - DMA
- **Character** devices include **keyboards, mice, serial ports**
    - Commands include `get()`, `put()` of one character
    - Libraries layered on top allow line editing

Nonblocking and Asynchronous I/O

- **Blocking**: process **suspended** until I/O completed
    - Processes moved from run queue to wait queue
- **Nonblocking**: I/O call returns as much as available
    - Returns quickly with count of bytes read or written
- **Asynchronous**: process runs while I/O executes
    - An alternative to nonblocking I/O
    - I/O request will be completed at some future time
    - I/O subsystem signals process when I/O completed
        - Software interrupt
        - Signal
        - Callback routine