CSCI 360
Introduction to Operating Systems

I/O System

Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

Outline

- I/O Concepts
  - I/O Devices
  - Device Controllers
  - I/O Ports
  - Memory Mapped I/O
  - Programmed I/O
  - Interrupt Driven I/O
  - Direct Memory Access (DMA)
  - I/O Using DMA
  - Interrupt Controller
- I/O Software Layers
  - User I/O Layer
  - Device Independent I/O Layer
  - Device Driver
  - Interrupt Handler

I/O Devices

- Mainly **2 types** of I/O devices
  - Block Devices: Hard Disk, Blue-ray Disk, and USB Stick
  - Character Devices: Printer, Network Interface Card, and Mouse.

I/O Devices

- Block Devices
  - Stores information in fixed-size blocks, each one with its own address.
  - **Transfers** are in units of entire blocks.
  - Allows to **read** or **write** each block **independently**.

I/O Devices

- Character Devices
  - **Transfers** stream of characters, without regard to block structure
  - **Not addressable**, does not have any *seek* operation

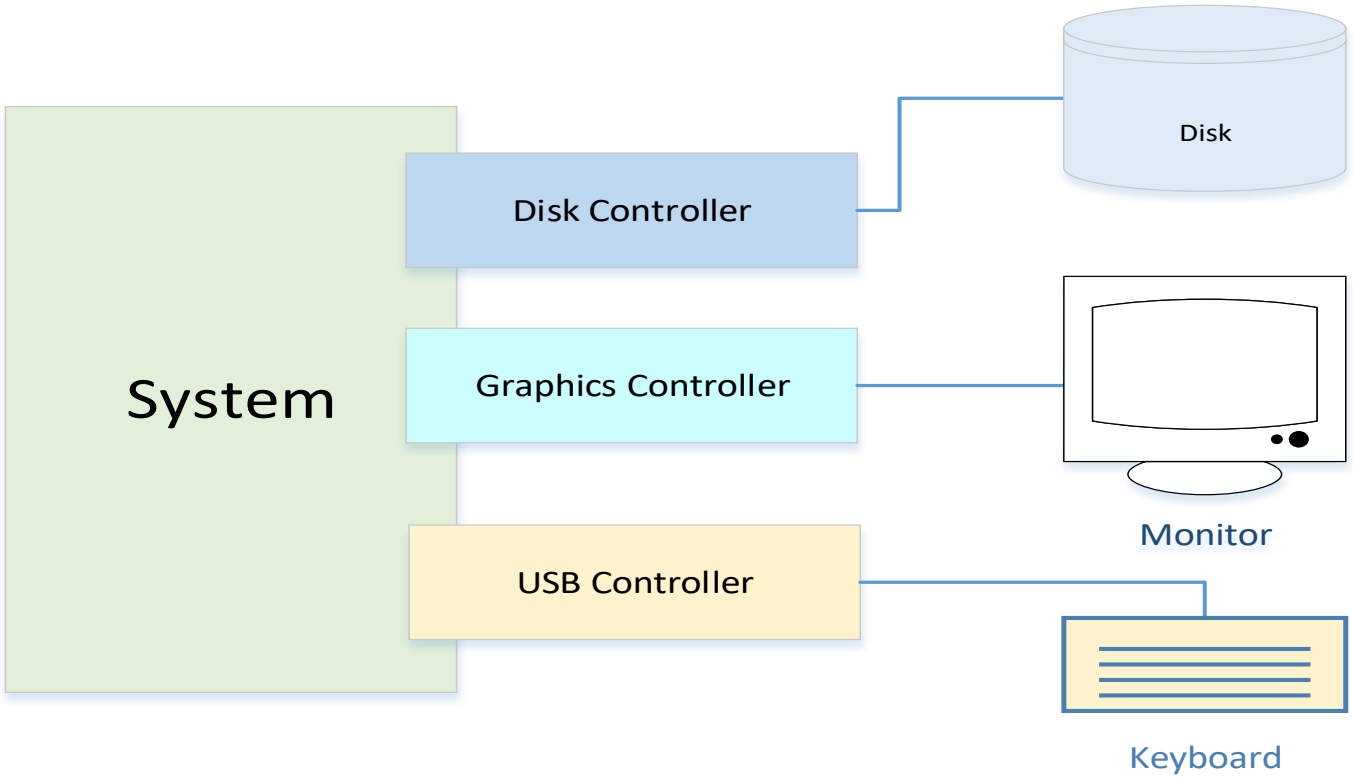
I/O Devices

Come with **fixed** data rate.

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner at 300 dpi	1 MB/sec
Digital camcorder	3.5 MB/sec
4x Blu-ray disc	18 MB/sec
802.11n Wireless	37.5 MB/sec
USB 2.0	60 MB/sec
FireWire 800	100 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk drive	600 MB/sec
USB 3.0	625 MB/sec
SCSI Ultra 5 bus	640 MB/sec
Single-lane PCIe 3.0 bus	985 MB/sec
Thunderbolt 2 bus	2.5 GB/sec
SONET OC-768 network	5 GB/sec

Device Controller

Device Controllers **connect** devices to the systems



Device Controller

- Each Device Controller has control registers that the system can use to **write** control commands to the device.
- Control registers or status registers can be **read** to know the status of the device.
- Some devices may have data buffer in addition to control registers.
- Control registers and data buffer can be **addressed** in two ways:
  - Using port numbers
  - **Mapping** to memory addresses.

Port Mapped I/O

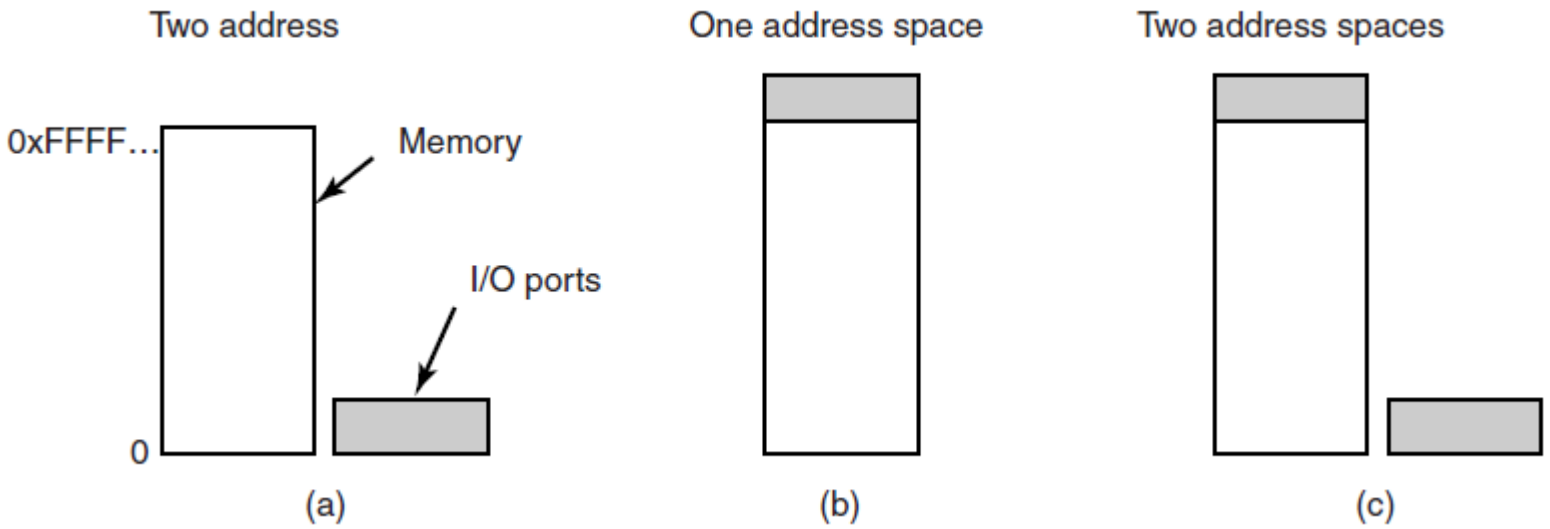
- Controller registers are **assigned** 8 or 16-bit port numbers to address.
- I/O port space is **separate** from memory address space.
- System **access** I/O ports by using special I/O instructions.

IN *reg port_number*

OUT *port_number reg*

Memory-Mapped I/O

Each control register is **mapped** to a unique memory address to which no memory is assigned.



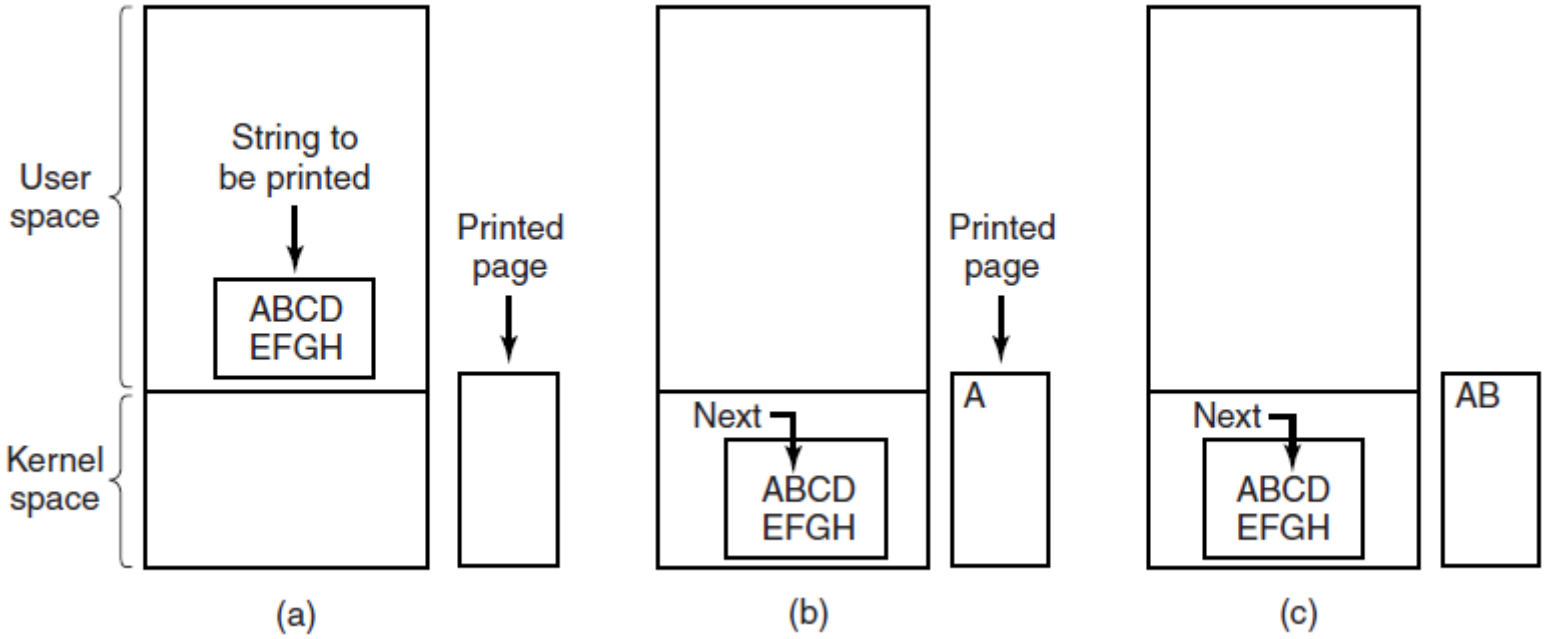
Separate I/O and memory spaces

Memory-mapped I/O

Hybrid I/O

Programmed I/O

Steps in **printing** a string



String in user Buffer

String **copied** to Kernel buffer and letter A has been **printed**

Letter B has been **printed**

Programmed I/O

Writing a string to the printer using programmed I/O.

```
print_driver(buffer, p, count) {
        copy_from_user(buffer, p, count);              // p is the kernel buffer
        for( i=0; i<count; i++) {                      // loop on every character
                while(*printer_status_reg != READY);   // loop until ready
                *printer_data_register = p[i];         // output one character
         }
         return_to_user();
}
```

Print Device Driver code, **invoked** through system call

Programmed I/O

- Programmed I/O is **Synchronous** or **Blocking**.
- CPU is **busy** with I/O operation **until** the I/O transfer is **complete**.

Interrupt Driven I/O

Writing a string to the printer using Interrupt Driven I/O

```
print_driver(buffer, p, count) {
        copy_from_user(buffer, p, count);          // p is the kernel buffer
        i = 0;                                     // initialize print count
        enable_interrupts();
        while(*printer_status_reg != READY);       // loop until ready

       *printer_data_register = p[i++];            // output first character and
                                                   // increment print count
         scheduler();
}
```

Print Device Driver code, **invoked** through system call

Interrupt Driven I/O

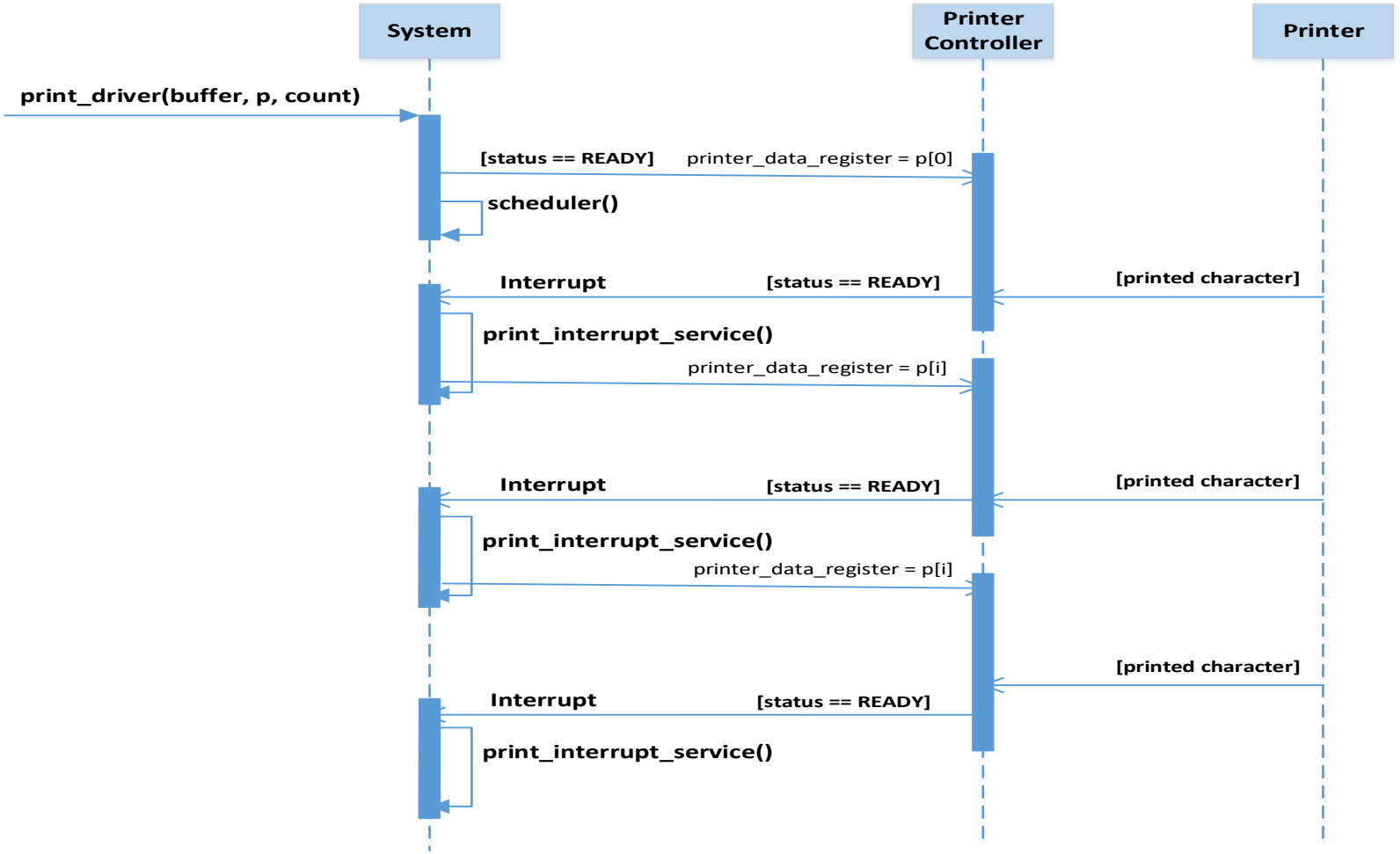
Writing a string to the printer using Interrupt Driven I/O

```
print_interrupt_service() {
        if(count == 0) {
                unblock_user();
        }
        else {
                *printer_data_register = p[i++];    // output one character and increment print count
                 count--;                           // decrement character count
         }
         acknowledge_interrupt();
         return_from_interrup();
}
```

Print Interrupt Service code, invoked through device Interrupt

Interrupt Driven I/O

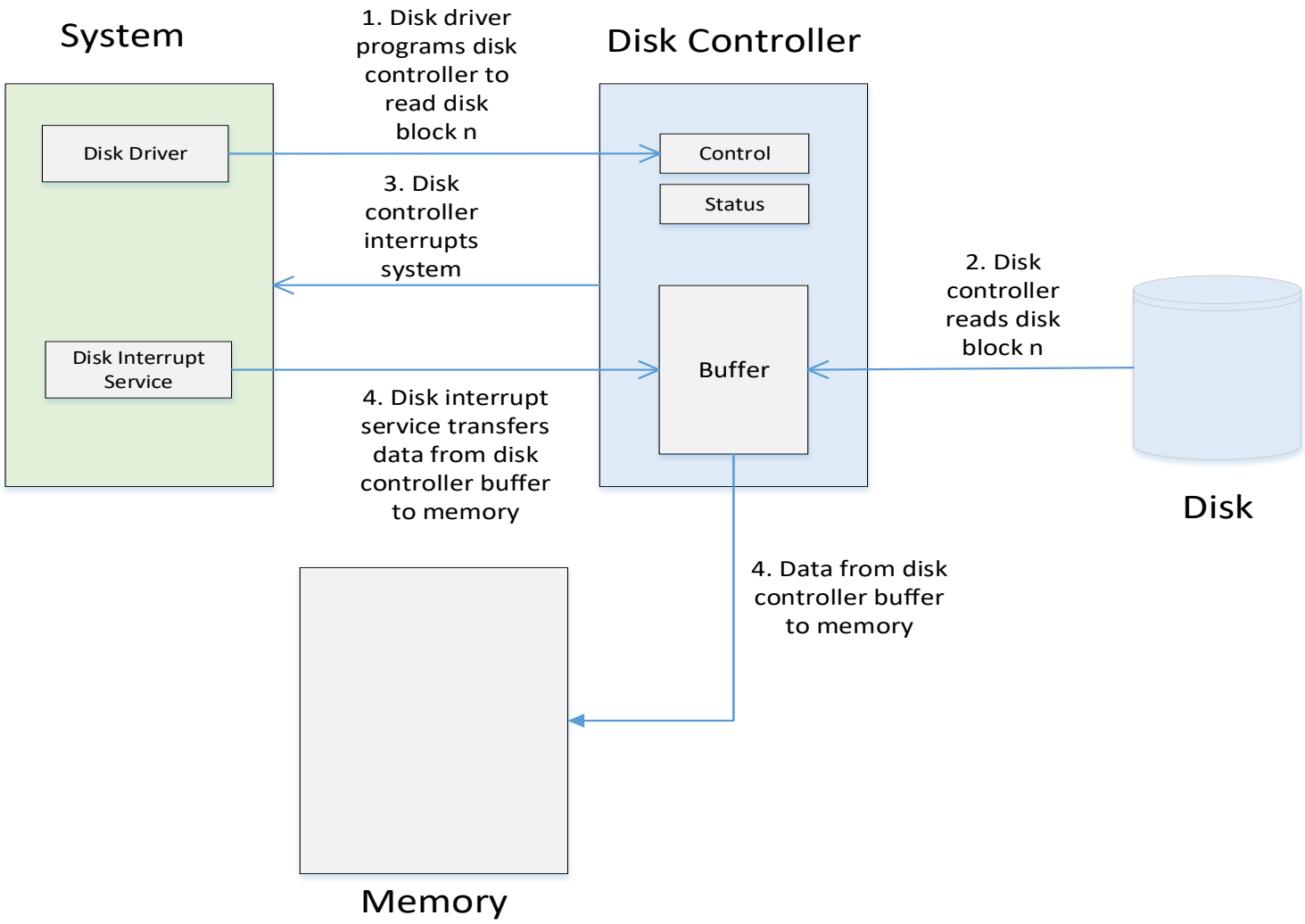
Writing a string to the printer using Interrupt Driven I/O



Interrupt Driven I/O

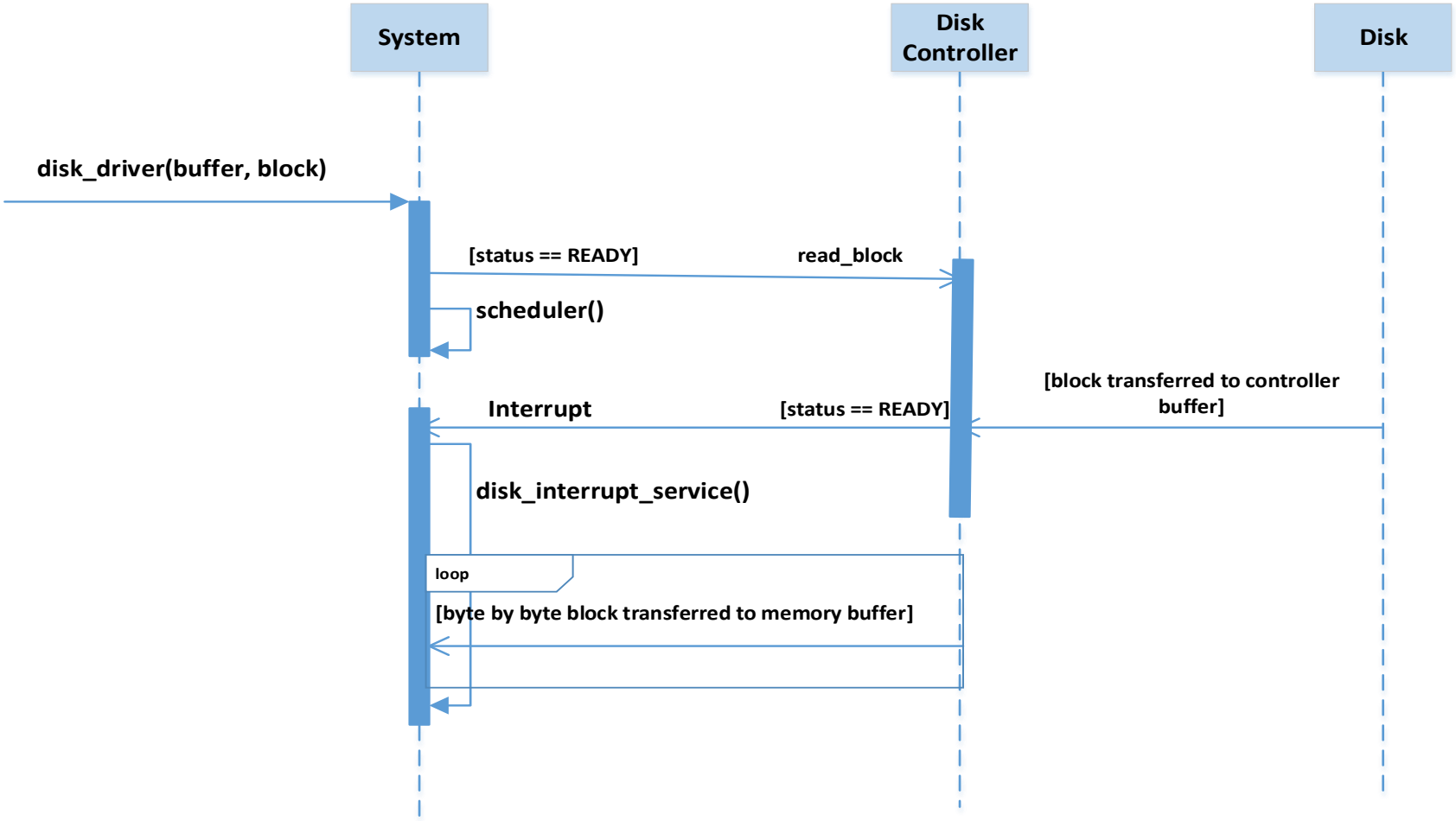
- Interrupt driven I/O is **asynchronous** or **non-blocking**.
- CPU **proceeds** with other jobs until **interrupted** by the device controller or interrupt controller.

Interrupt-Driven Disk I/O



Interrupt-Driven Disk I/O

Reading a disk block from the disk using Interrupt Driven I/O



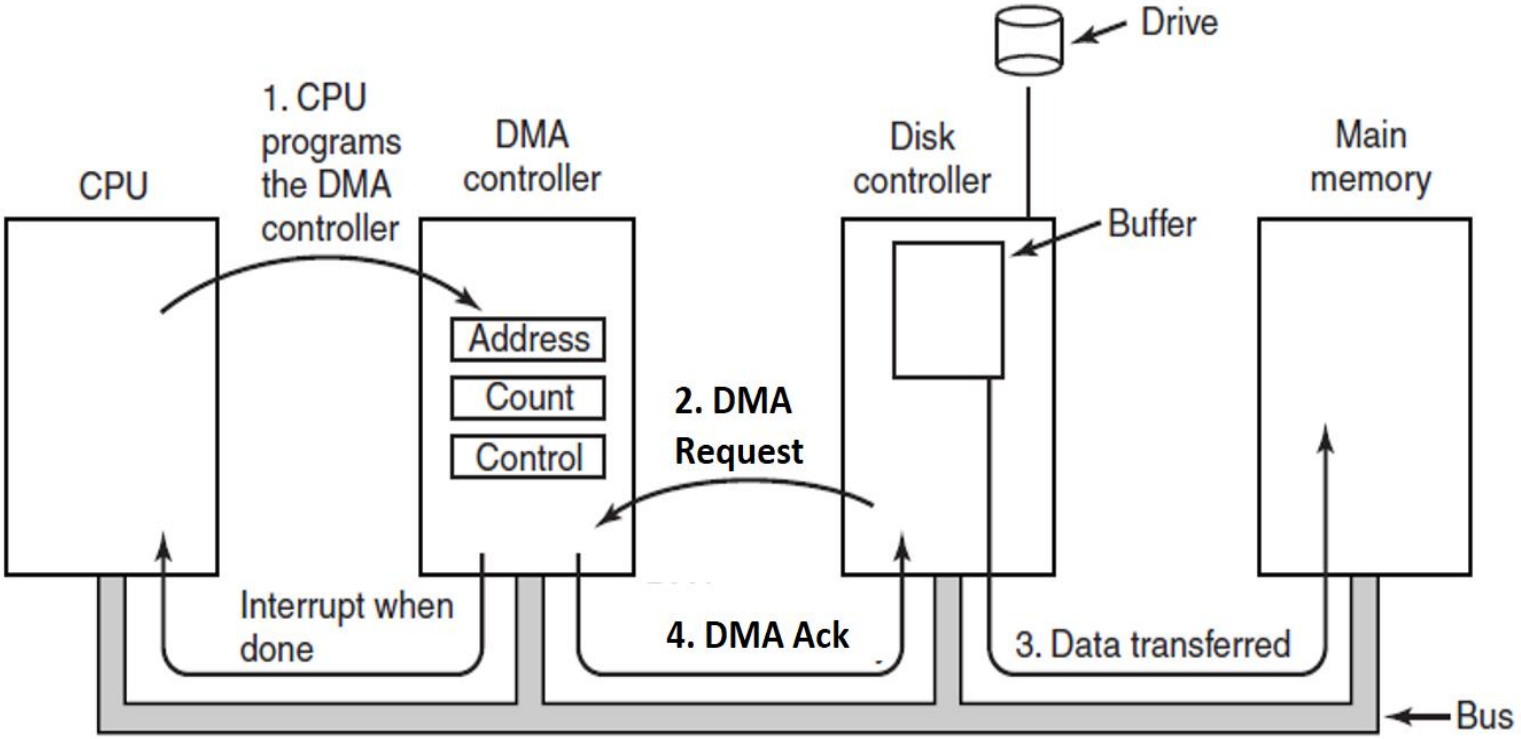
Interrupt-Driven Disk I/O

- System **writes** a read command on disk controller.
- Disk controller
  - **Reads** the data block from the drive serially, bit by bit, until the entire block is in the controller's internal buffer.
  - **Computes** the checksum to verify that no read errors have occurred.
  - **Asserts** an interrupt to the CPU to transfer the data from the buffer.
- Disk Interrupt Service **transfers** the data **byte** by **byte** from the controller buffer to the memory.

Direct Memory Access

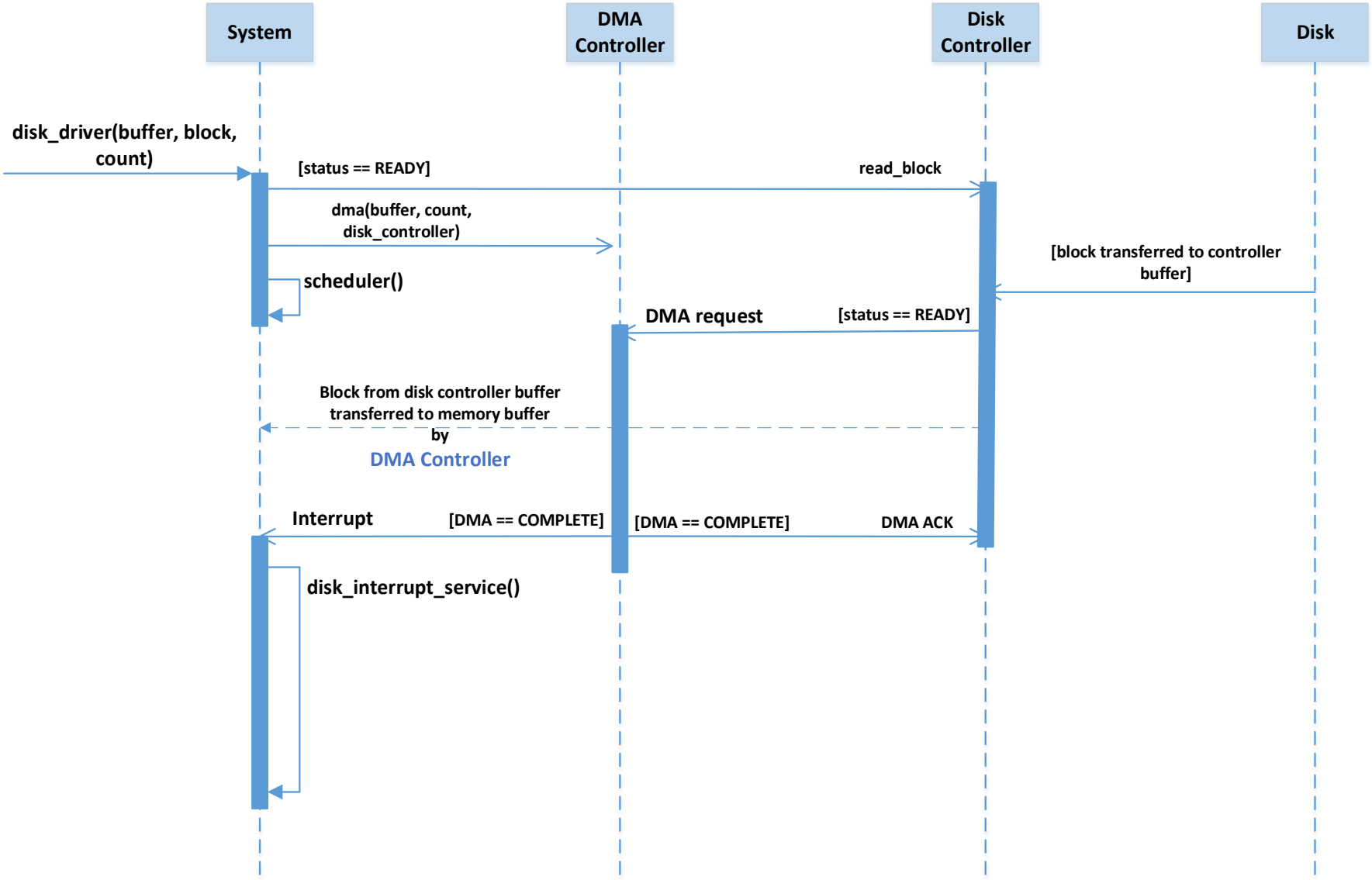
- **Getting** I/O data one byte at a time **wastes** CPU time.
- Using Direct Memory Access (DMA) CPU time waste is **avoided**.
- System needs a DMA Controller, which has direct **access** to the system bus to **transfer** data from I/O buffer to memory without involving CPU.
- DMA Controllers come with control registers, memory address registers, and byte count register.

Disk I/O with DMA



Operations of a DMA transfer.

Disk I/O with DMA



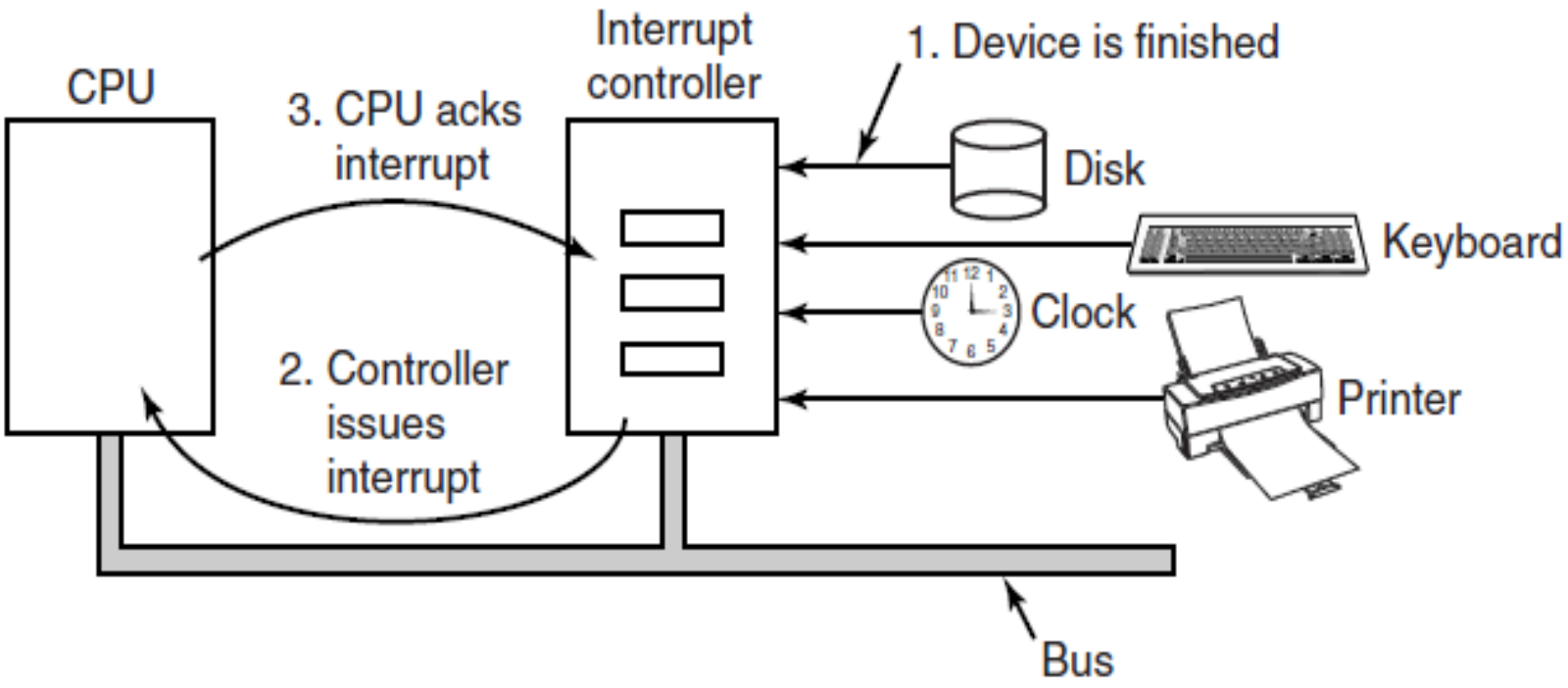
Disk I/O with DMA

- System **instructs** DMA controller by **setting** the source (disk controller buffer) and destination (memory buffer) and the byte count.
- System also **instructs** the disk controller to **read** a block of data from the disk.
- The disk controller **reads** the whole block into its internal buffer and **asserts** a DMA request to DMA controller.
- DMA Controller **requests** for the system bus access.

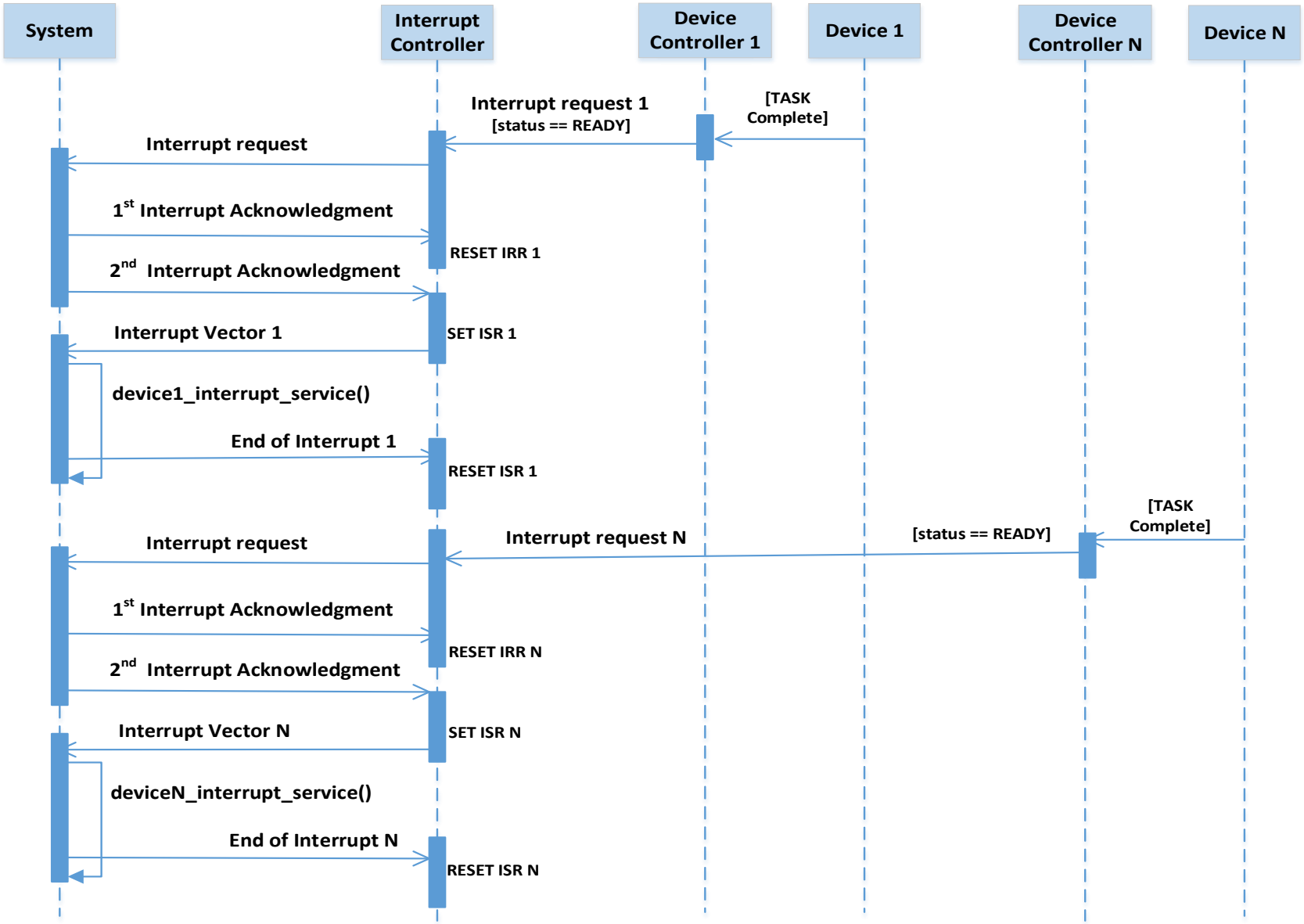
Disk I/O with DMA

- DMA controller **completes** direct data transfer from disk controller buffer to memory after acquiring the system bus access.
- Once the transfer is complete, DMA controller **asserts** *DMA acknowledgement* to the disk controller and *interrupt* to the system.
- System (Interrupt Service Routine) **asserts** interrupt acknowledgement to DMA controller and **unblocks** the user process that was waiting for the I/O to complete.

Interrupt Controller



Interrupt Controller



Interrupt Controller

- Most of the systems come with a **single** Interrupt Request line and a **single** Interrupt Acknowledgement line.
- A **centralized** Interrupt Controller is often used to get interrupts from **multiple** I/O devices.
- An Interrupt Controller comes with **multiple** Interrupt Request lines to connect with multiple I/O devices.
- An Interrupt Controller comes with an Interrupt Priority Resolver to resolve priority of the interrupts from **multiple** I/O devices.
- The **interrupt** from the highest priority device is **asserted** through the **system** Interrupt Request line.

Interrupt Controller

- System **asserts** the **first** Interrupt Acknowledgement to Interrupt Controller to inform the acceptance of the interrupt.
- System **asserts** the **second** Interrupt Acknowledgement to the Interrupt Controller and **waits** for the Interrupt Vector corresponding to the requesting device.
- Interrupt Controller **asserts** the Interrupt Vector to the system through the data lines.
- System **invokes** the Interrupt Service Routine corresponding to the interrupt vector.
- System **asserts** End of Interrupt to the Interrupt Controller at the completion of interrupt service routine.

Goals of I/O Software

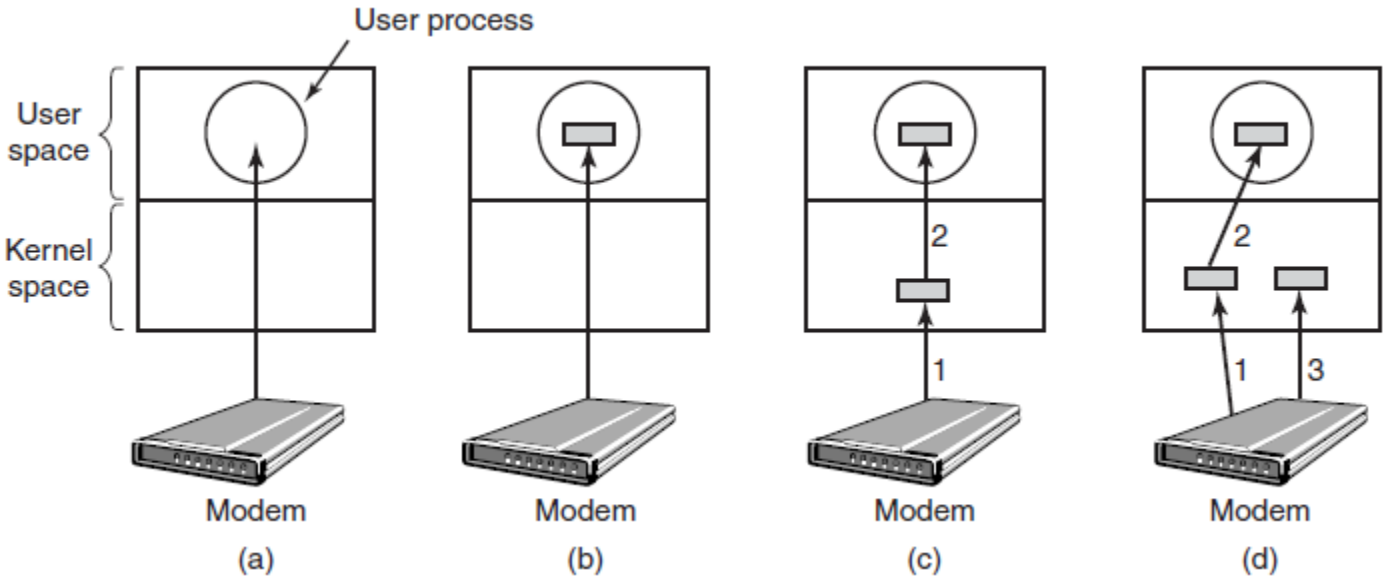
- **Device independence**
  - Similar methods to access different types of devices.
- **Uniform naming**
  - Similar naming scheme for different types of devices.
- **Error handling**
  - Lower layers must **handle** and **conceal** as many errors as possible from the upper layer

Goals of I/O Software

- **I/O operations**
  - **Supports** synchronous (blocking) or asynchronous (interrupt driven) I/O operations.
- **Buffering**
  - Should **employ** buffers to **decouple** one layer from another layer.

Goals of I/O Software

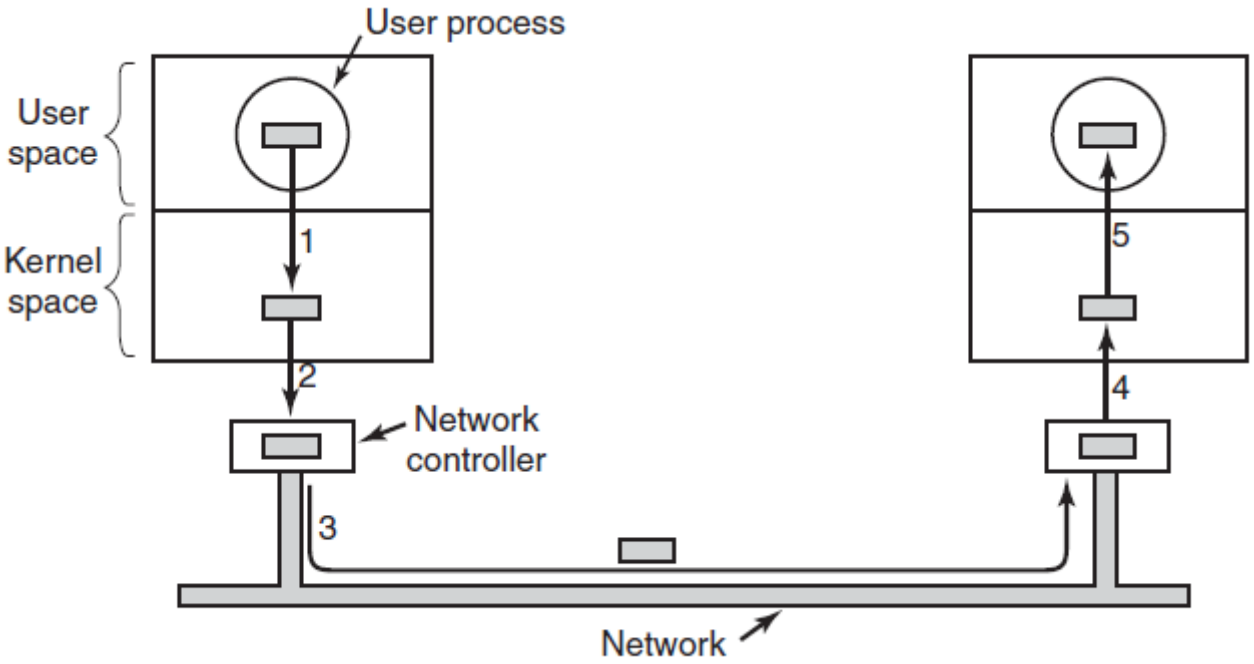
Buffering



(a) **Unbuffered** input. (b) **Buffering** in user space. (c) **Buffering** in the kernel followed by **copying** to user space. (d) **Double buffering** in the kernel.

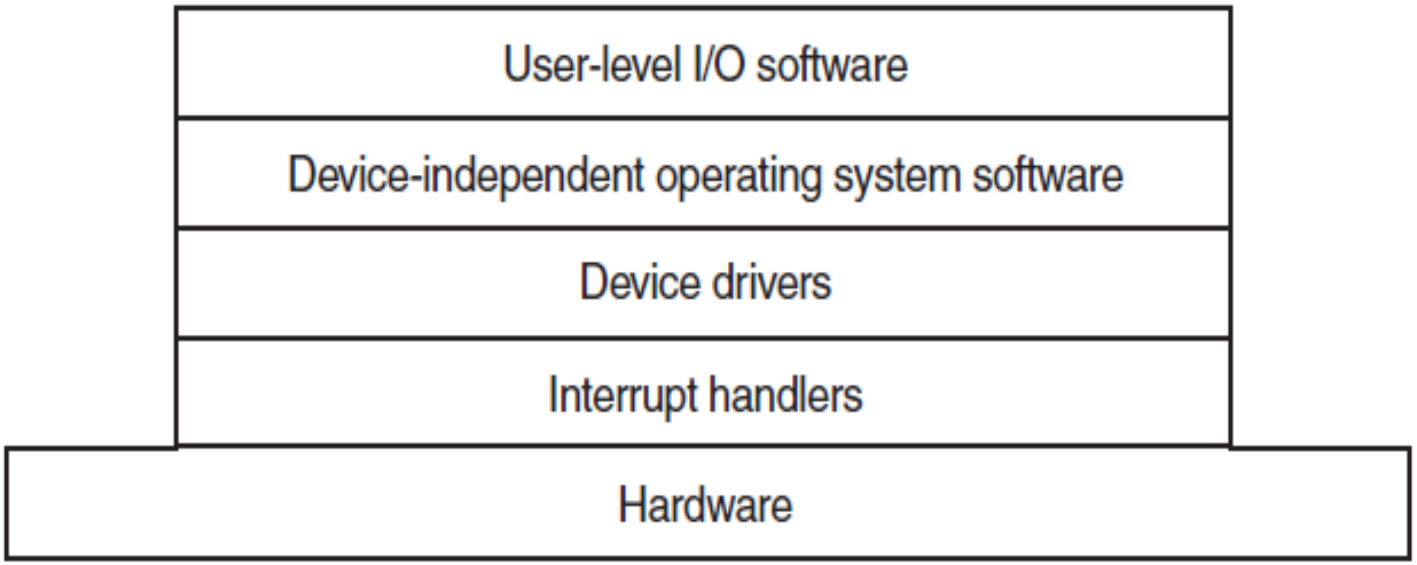
Goals of I/O Software

Buffering

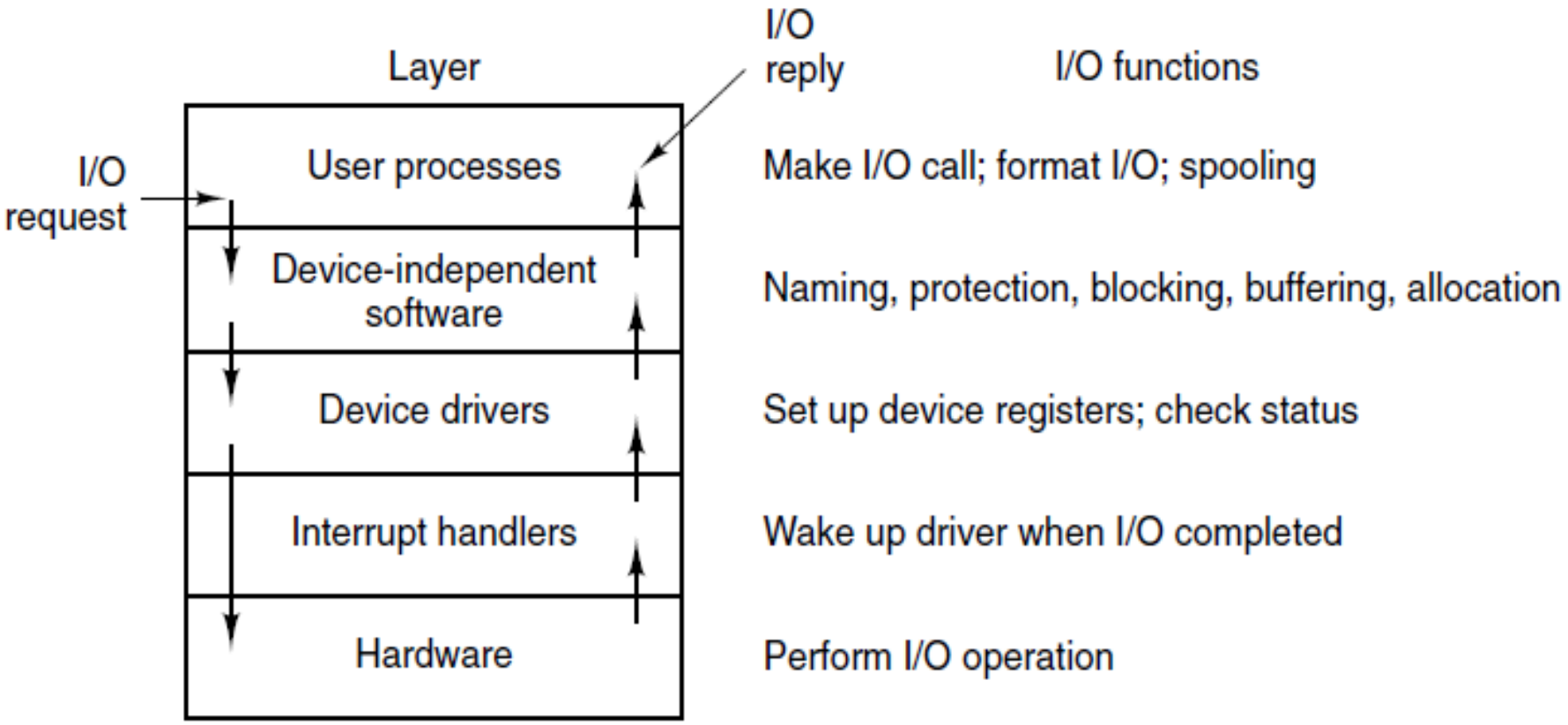


Buffering at **three levels** (user space, kernel space, and device controller)

I/O Software Layers



I/O Software Layers



User Process I/O Software

User process I/O software is actually the user space library functions that perform I/O operations, e.g., scanf(), printf(), gets(), puts() etc.

Device-Independent I/O Software

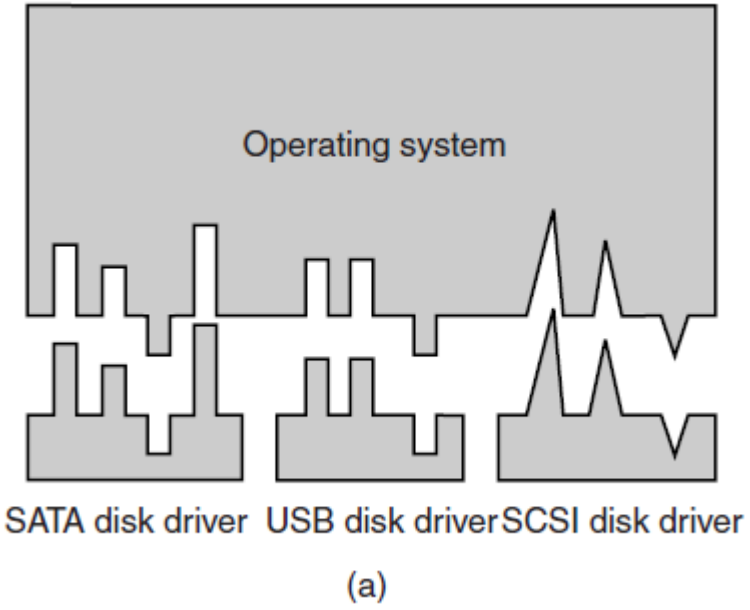
Functions of the device-independent I/O software

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

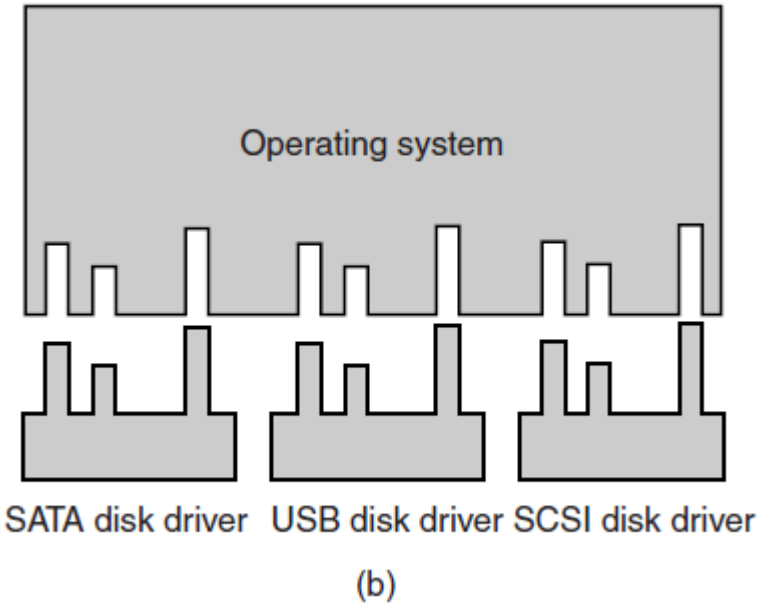
Device-independent I/O software is actually the system call functions both user space and kernel space versions, e.g., open(), close(), read(), write() etc.

Device-Independent I/O Software

Uniform Interfacing for Device Drivers



Without a standard driver interface



With a standard driver interface

Device-Independent I/O Software

Kernel level buffering is handled by device-independent I/O software

Device-Independent I/O Software

Error Handling

- Framework for **error handling** is device independent although many errors are device specific.
- **Program Errors**
  - Device independent error handing framework **reports** back an error code to the caller.
- **Actual I/O Errors**
  - Device drivers or Disk controllers mostly **handle** them.
  - Device independent error handing framework also **handles** them when the lower layers don't and the response is specific to error type.

Device-Independent I/O Software

Allocating and Releasing Dedicated Devices

- **Open** and **Close**
  - **Forces** the process to **open** and **close** a special file specific to a dedicated device **before** and **after** using it.
- **Device Queue**
  - Requesting process **enters** at the back of a device queue to get access to the device.
  - Process at the front of the queue get **access** of the device and is **removed** from the queue.

Device-Independent I/O Software

Providing Device-independent Block Size

- **Hides** the fact that devices (disks) often come with different block sizes by **providing** a common logical block size.
- **Treats** several device blocks as a single logical block.
- Upper layers **deal** only with abstract devices that all use the **same** logical block size, **independent** of the physical block size.

Device Drivers

- Code **specific** to a particular device or a class of devices. Directly **accesses** device controller's registers for **giving** commands, **reading** status, and **transferring** data.
- Device manufacturers **supply** the code along with the devices.
- Device manufactures **follow** the standard interfaces **defined** by the operating systems to **write** device driver codes.
- **Loaded** dynamically and **executed** as part of operating system code, i.e., kernel mode.

Device Drivers

Device Driver **Functions**

1. **Accepts** abstract **read** and **write** requests from device independent layer.
2. **Returns** error if the request parameters are not valid.
3. **Translates** abstract terms into concrete terms, e.g., device block into device head, cylinder, sector etc.
4. **Initializes** the device, if needed.
5. **Reads** device controller status register to check whether the device is in use or idle.

Device Drivers

Device Driver **Functions** (continued..)

6. If the device in use **enters** the request into device queue and **blocks** itself. Device interrupt **awakens** the blocked device driver.

7. **Writes** commands to the device controller's control register and **blocks** itself to let the device **take** actions and **awake** the driver by issuing a device interrupt.

8. If successful and if it is necessary, **passes** the data to the device-independent software. If unsuccessful, **passes** the error code to the caller.

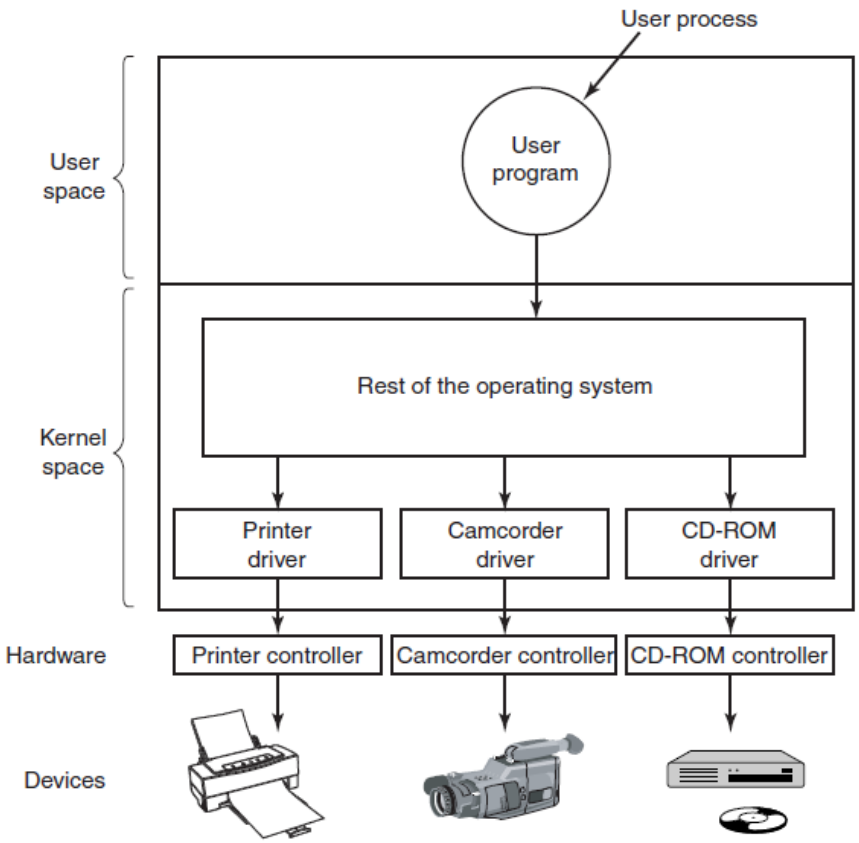
Device Drivers

Device Driver **Functions** (continued..)

9. If more requests are pending in the device queue, **proceeds** with the one at the front and **repeats** steps 8 to 10.

10. If no more pending requests, **blocks** itself until a new request arrives.

Device Drivers



- Logical positioning of device drivers.
- In reality all **communication** between drivers and device controllers goes over the bus.

Interrupt Handlers

- Interrupt hardware **flips** the mode bit in PSW to kernel mode.
- **Pushes** PC of the current process onto stack.
- **Jumps** to the interrupt handler routine.

Interrupt Handlers

Interrupt handler routine (I/O software) steps

1. **Pushes** registers (including the PSW) of the current process that are not saved by interrupt hardware onto the stack.
2. **Determines** which interrupt service routine to **invoke** based on interrupt vector**.**
3. **Sets up** context for interrupt service routine.
4. **Sets up** a stack for the interrupt service routine.
5. **Copies** saved registers (saved by the device driver) from the stack into process table.

Interrupt Handlers

Interrupt handler (I/O software) steps (continued…)

6. **Runs** interrupt service routine, which **extracts** information from interrupting device controller's registers and conditionally **unblocks** the corresponding user process.
7. **Acknowledges** interrupt controller. If no interrupt controller, re-enable interrupts.
8. **Gets** the interrupted process to run.
9. **Sets up** the MMU context for the interrupted process to run.

Interrupt Handlers

Interrupt handler (I/O software) steps (continued…)

10. **Loads** the interrupted process's PC, PSW, and other necessary registers.
11. **Return** from interrupt calling IRET, as a consequence hardware flips the mode bit to user mode.
12. **Start running** the interrupted process.

Summary

- I/O Concepts
  - I/O Devices
  - Device Controllers
  - I/O Ports
  - Memory Mapped I/O
  - Programmed I/O
  - Interrupt Driven I/O
  - Direct Memory Access (DMA)
  - I/O Using DMA
- I/O Software Layers
  - User I/O Layer
  - Device Independent I/O Layer
  - Device Driver
  - Interrupt Handler

Next

Protection

- Protection Domain
- Access Control List
- Capabilities