CS 310 Operating Systems

Lecture 39: Device Drivers – Concept and Functions

Ravi Mittal

IIT Goa

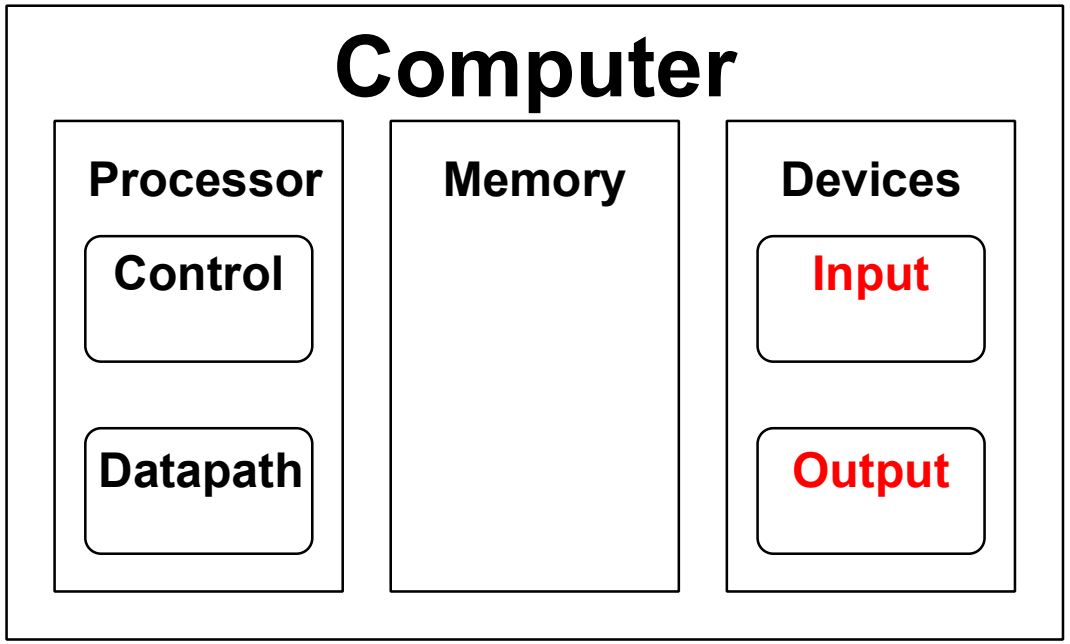
References

- CS162, Operating Systems and Systems Programming, University of California, Berkeley
  - Book: Modern Operating Systems by Andrew Tanenbaum and Herbert Bos
  - Book: Operating System Concepts: A Silberschatz, Peter Galvin, Greg Gagne

Input Output – We have learnt ..

Recall: Five Components of a Computer

- Components a computer needs to work
  - Control
  - Datapath
  - Memory
  - **Input**
  - **Output**



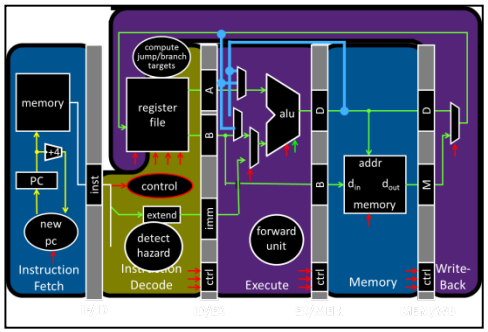
Ref: Morgan Rae Reschenberg, Course 61C, Univ of California, Berkeley

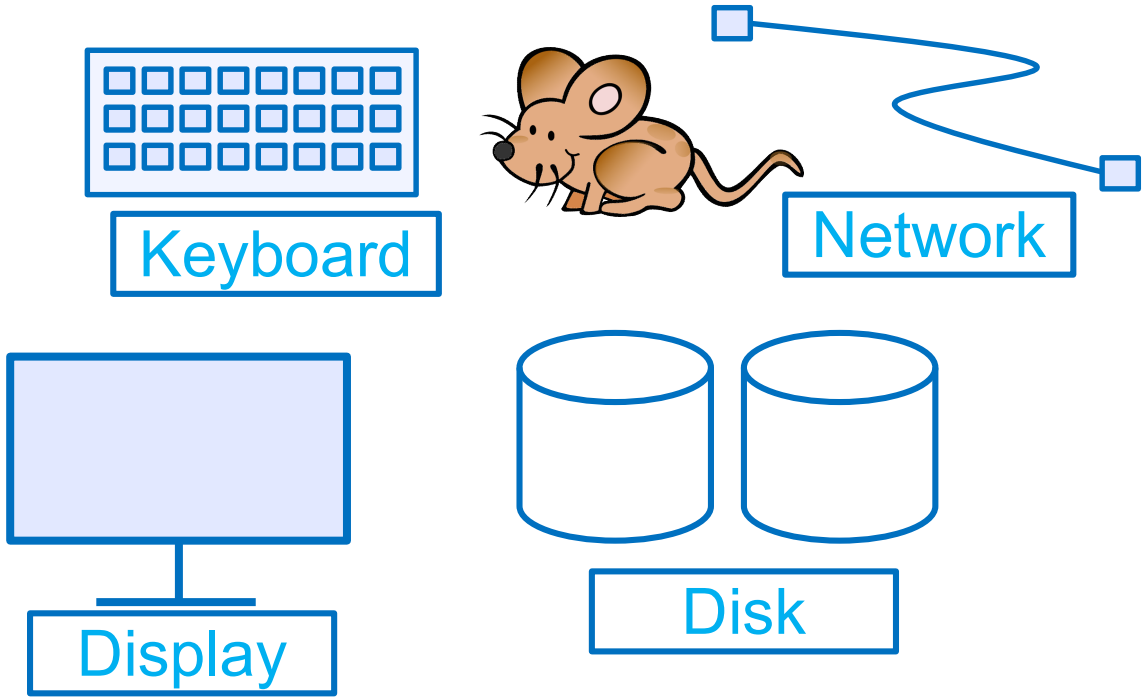
Recall: Big Picture: Input/Output (I/O)

How does a processor interact with its environment?

Computer System =

Memory + Datapath + Control **+ Input + Output**

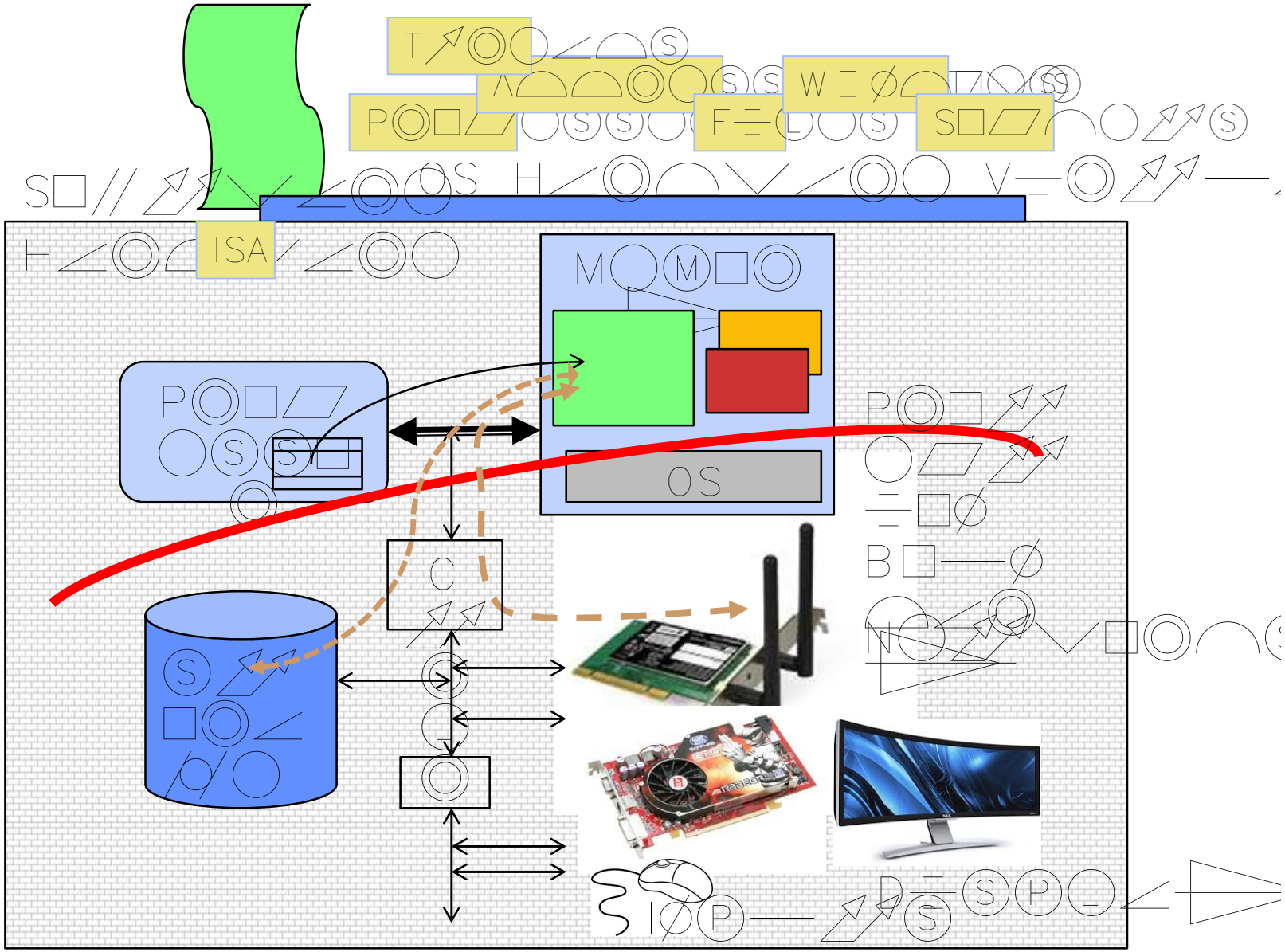




Ref: I/O lecture, Prof Hakim Weatherspoon, Cornell Univ

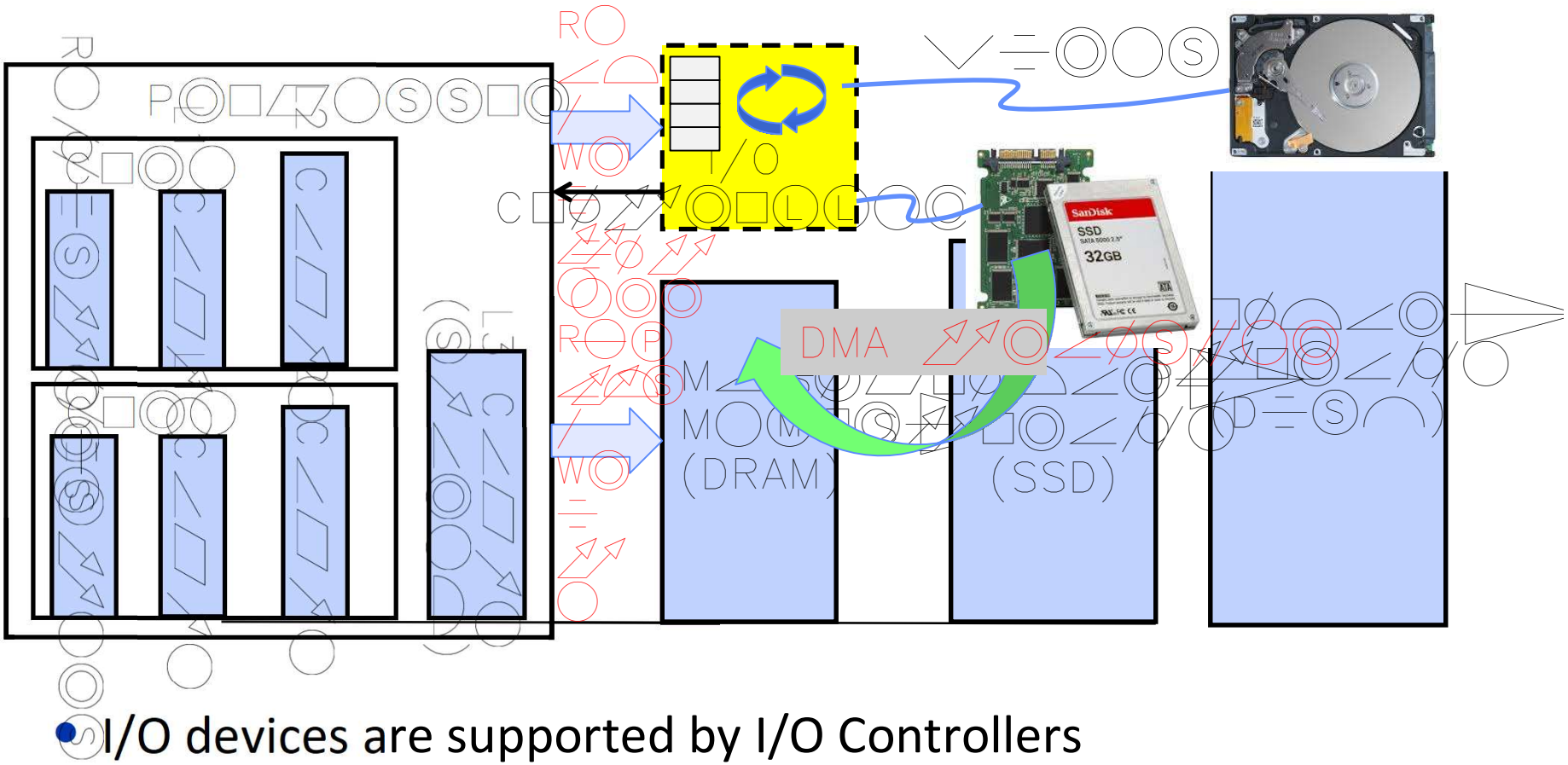
Recall: OS Basics: I/O

OS provides common services in the form of I/O



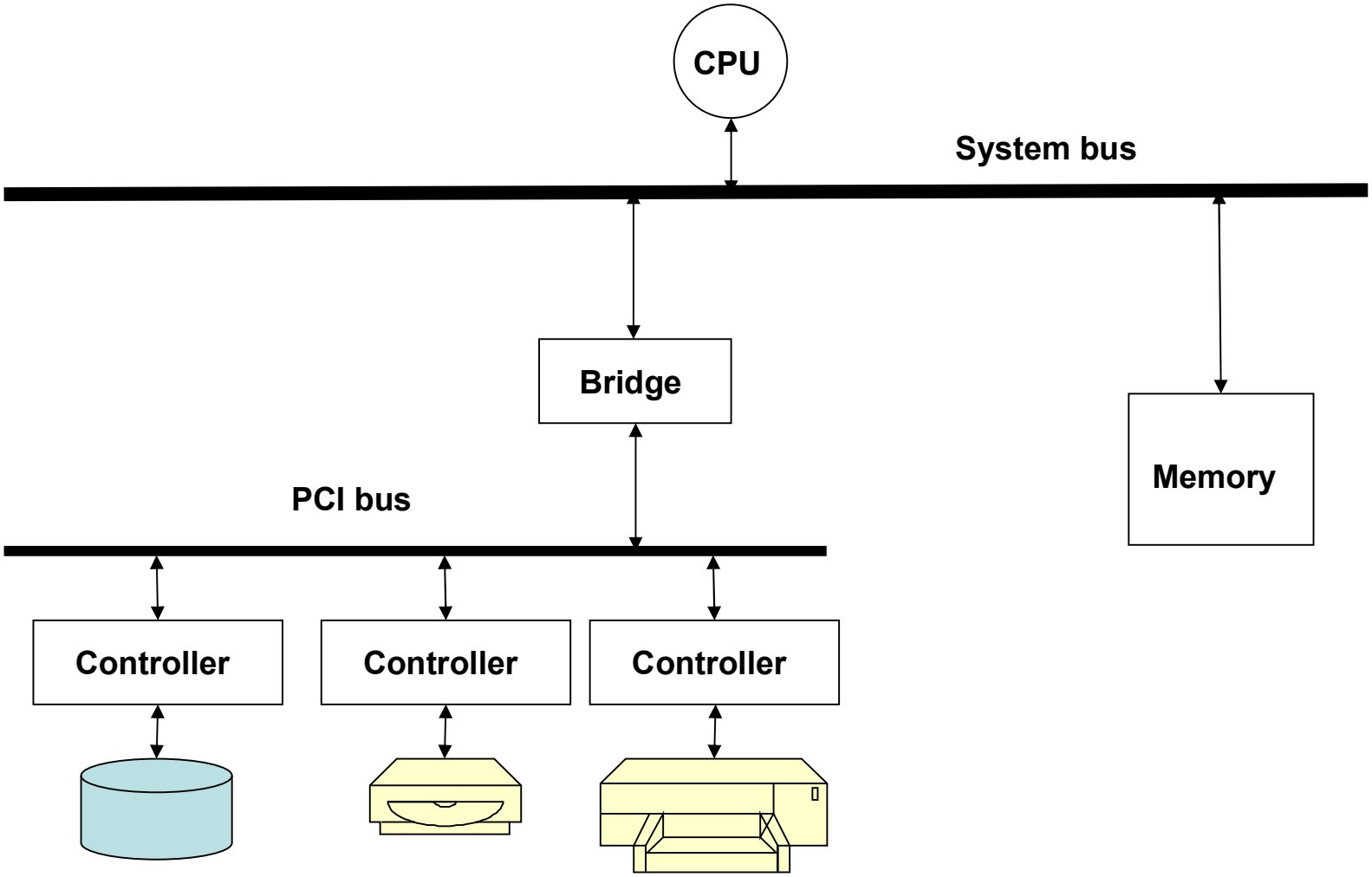
Adapted from lectures CS162 at UC Berkeley, 2020

In a Picture



- I/O devices are supported by I/O Controllers
- Processors accesses them by reading and writing IO registers as if they were memory
  - Write commands and arguments, read status and results

Buses


CPU
System bus
Bridge
Memory
PCI bus
Controller
Controller
Controller

PCI Express Bus

- Serial, point-to-point, switched bus, non-shared connections between devices
  - Similar to switched Ethernet
  - Slow devices don't have to share with fast ones
  - No longer a parallel bus
  - Star like architecture

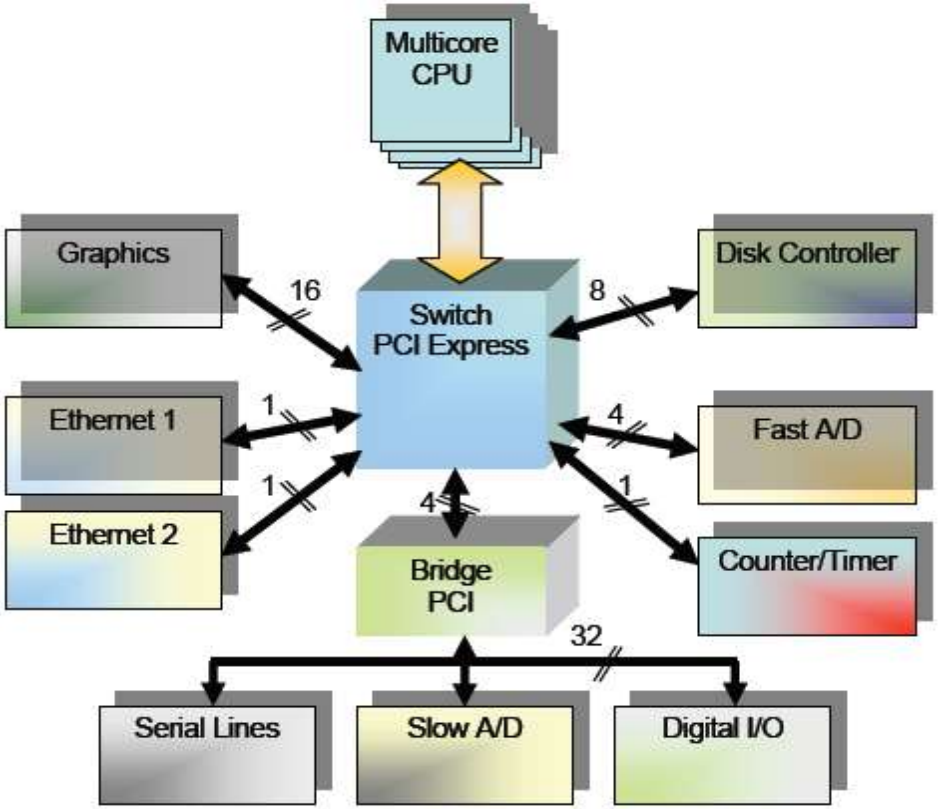
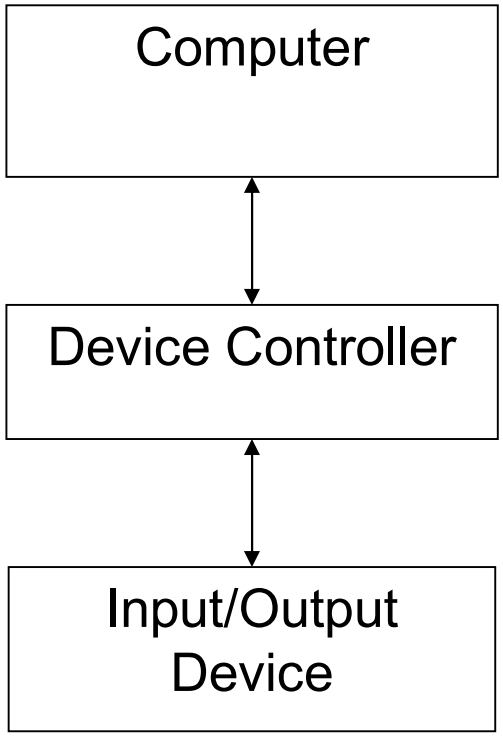
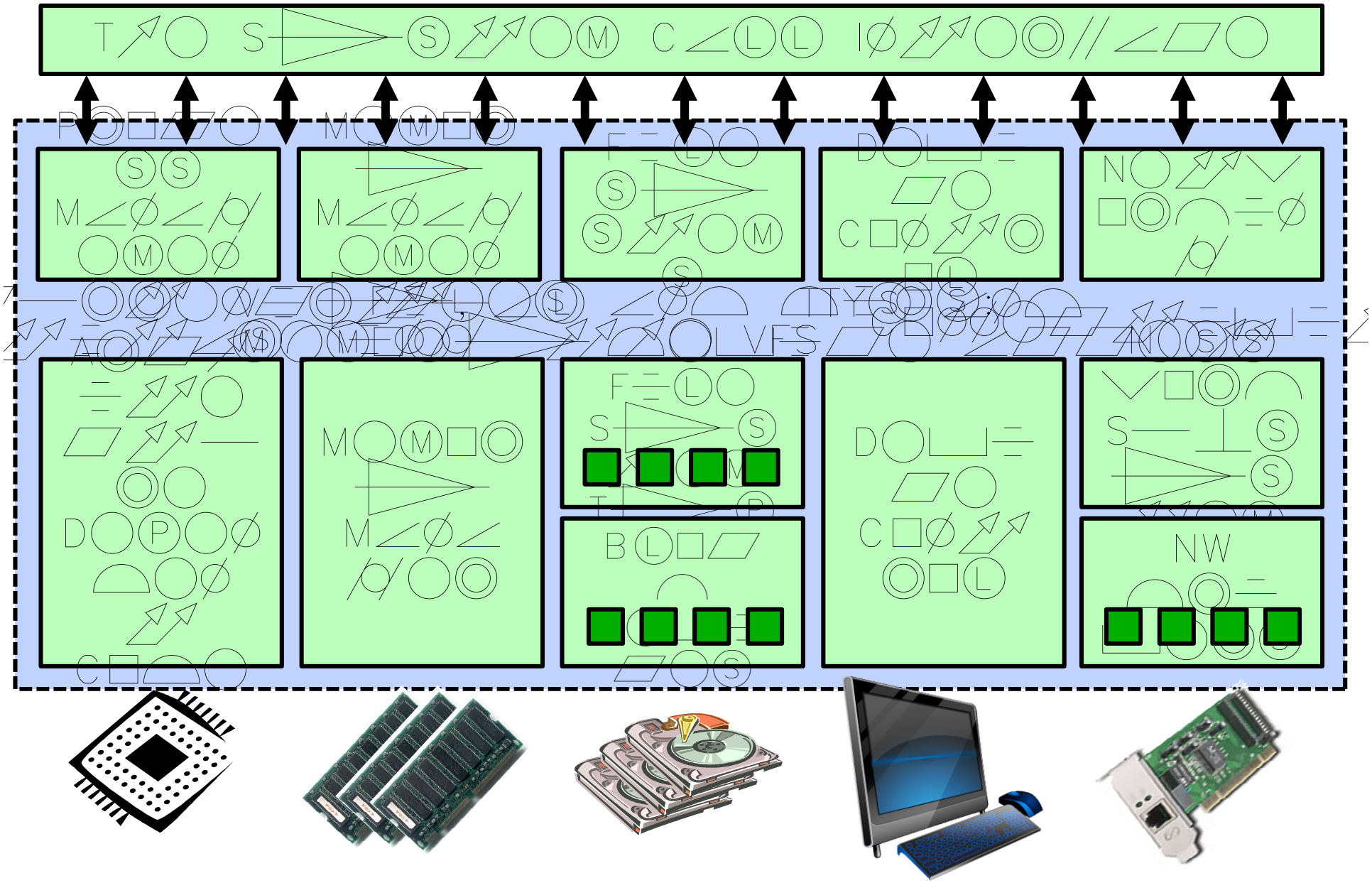


Figure 3: PCI Express switched fabric

Communication between the CPU and the I/O devices



Kernel Device Structure



In this lecture, we will study

- Programmed I/O, Interrupt driven I/O, DMA (repeat)
- Characteristics of I/O Devices
- Device Driver – Requirement
- Device Driver – Concept
- Device Driver – Functions
- Use of Interrupt Handler (for I/O operations)
- Device Independent Software
- User Space I/O Software – Libraries
- Lecture Summary

Programmed I/O, Interrupt Driven I/O, DMA

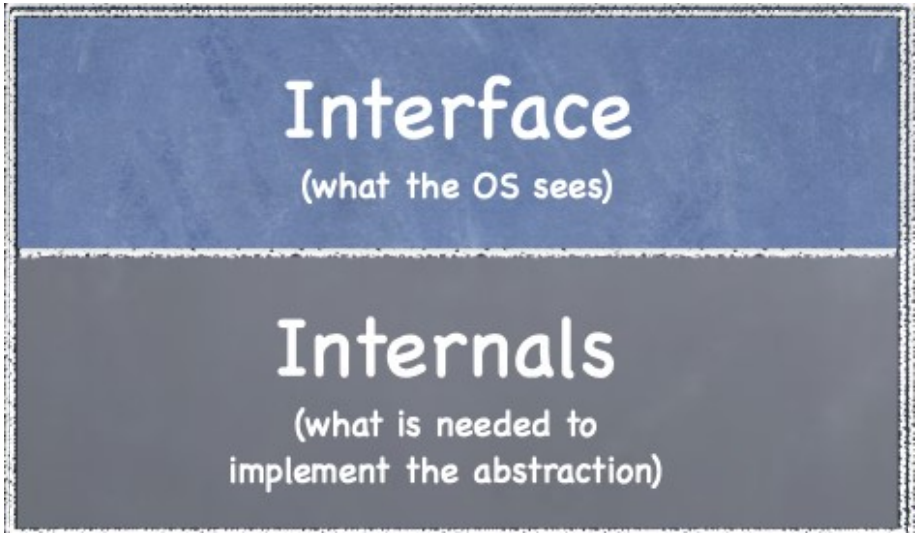
Interacting with a Device

What Users See

Abstraction

Interacting with a Device

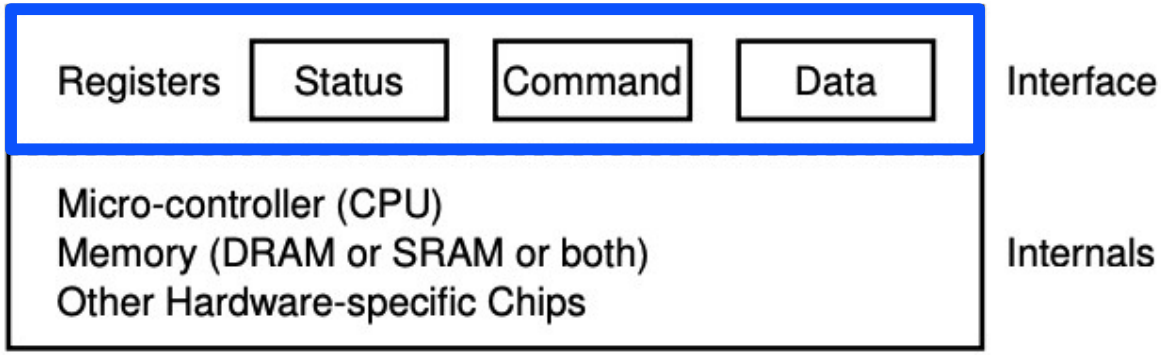
All devices have some specified interface and protocol for interaction



A Device

Interacting with a Device

All devices have some specified interface and protocol for interaction



- Device Interface: 3 registers
  - Status Register: to see the current status
  - Command Register: to tell device to perform a certain task
  - Data Register: to pass data to device or to receive data from device
- OS can control device behavior using these registers

Ref: OSTEP Book

Interacting with a Device – Programmed I/O

- CPU is repeatedly reading status register to check if the device is free - Polling
- OS sends some data down to the data register
  - Multiple such writes are needed to write a block (4KB) for disk
- Here, main CPU is involved with the data movement
  - Programmed I/O
- OS writes command to command register
  - It will lets the device know that both the data is present
  - it should begin working on the command
- OS waits for device to finish by polling it in a loop – waiting to see it it has finished

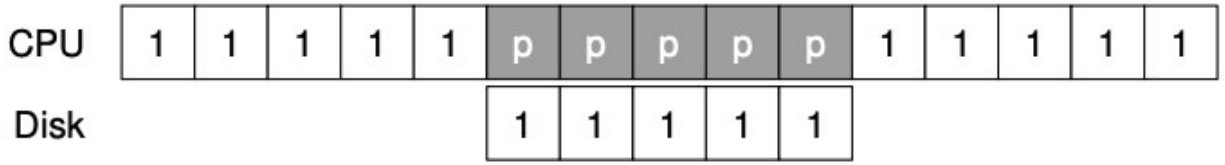
OS controls device by reading/writing registers

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

Simple but inefficient; Wastage of CPU time as OS keep polling

Ref: OSTEP Book

Interacting with a Device – Programmed I/O



- Process 1 runs on CPU for some time
- Issues I/O request to disk to read some data
- System simply spins –polling status repeatedly
  - Shown as p – polling
- The disk services the request and finally Process 1 can run again

Ref: OSTEP Book

Interrupt Driven I/O

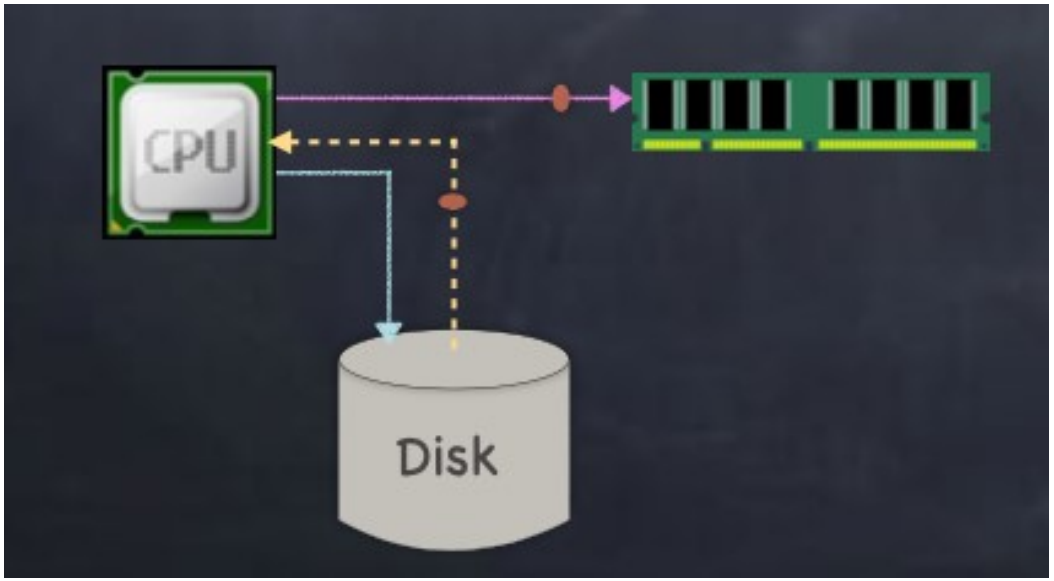
- Device ←→ CPU ←→ RAM

```
for (i = 1...n)
```

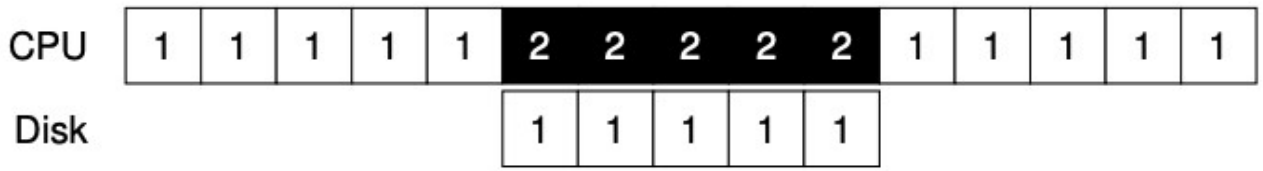
CPU issues read request

Device interrupts CPU with data

CPU writes data to memory



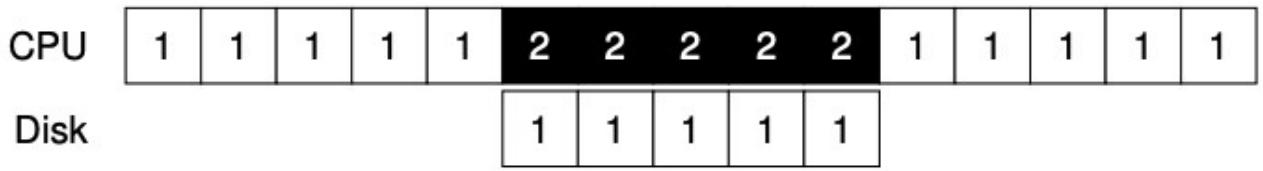
Interrupt Driven I/O



- OS runs Process 2 on CPU when disk is servicing Process 1's request
- When request is complete the interrupt occurs, OS wakes up process 1 and runs it again

- If device is fast, polling performs better than interrupt driven I/O
- If device is slow, interrupt driven I/O is faster than polling

Interrupt Driven I/O



- Livelock
  - A huge stream of incoming packets – each generating an interrupt
  - OS finds itself only processing interrupts
    - Not allowing a user process to run and actually service the request
  - It's better to use polling

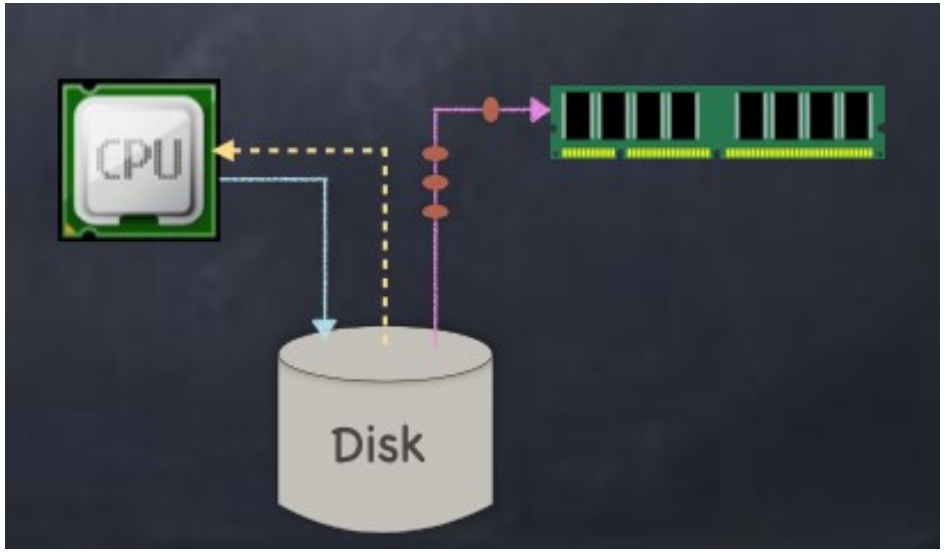
- Both Programmed I/O and Interrupt driven I/O are not efficient as CPU gets busy doing mundane task

In both cases, CPU time is wasted

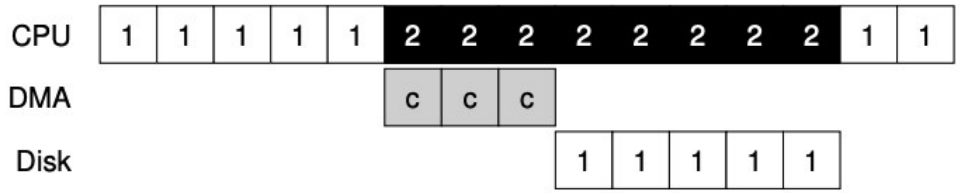
Direct Memory Access (DMA) I/O

- Device ←→ RAM

- CPU sets up DMA request
- Device puts data on bus & RAM accepts it
- Device interrupts CPU when done

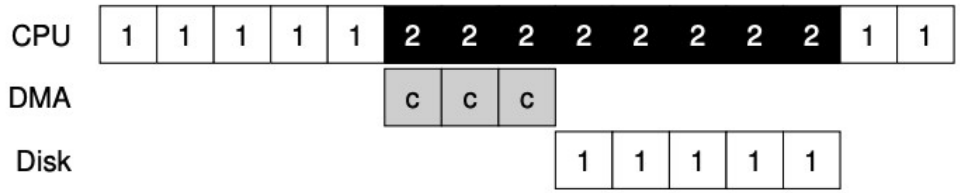


Direct Memory Access (DMA) I/O



- Process 1 is running and it wants to transfer some data to the disk
- It initiates I/O
  - Copy data from memory to the device explicitly
    - One word at a time (marked as c)
  - When copy is complete, I/O can begin on the disk
- CPU can be used for something else say process 2
- When transfer is complete, interrupt is raised and process 1 runs again

Direct Memory Access (DMA) I/O



- DMA engine
  - A very specific device within a system that can orchestrate transfer between devices and MM without CPU intervention
- To transfer data to device, OS programs DMA engine
  - Data location in the MM
  - How much data to copy
  - Which device to send it to
- When DMA is complete, DMA controller raises an interrupt and the OS knows that the transfer is complete

I/O Device Notifying the OS

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- I/O Interrupt: Device generates interrupt when it needs service
  - Handles unpredictable events well, but high overhead
- Polling: OS periodically checks device-specific status register
  - Low overhead, but may waste cycles for infrequent or unpredictable I/O
- Actual devices combine both polling and interrupts
  - E.g., high-bandwidth network adapter

Characteristics of I/O Devices

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

Characteristics of I/O Devices (Cont.)

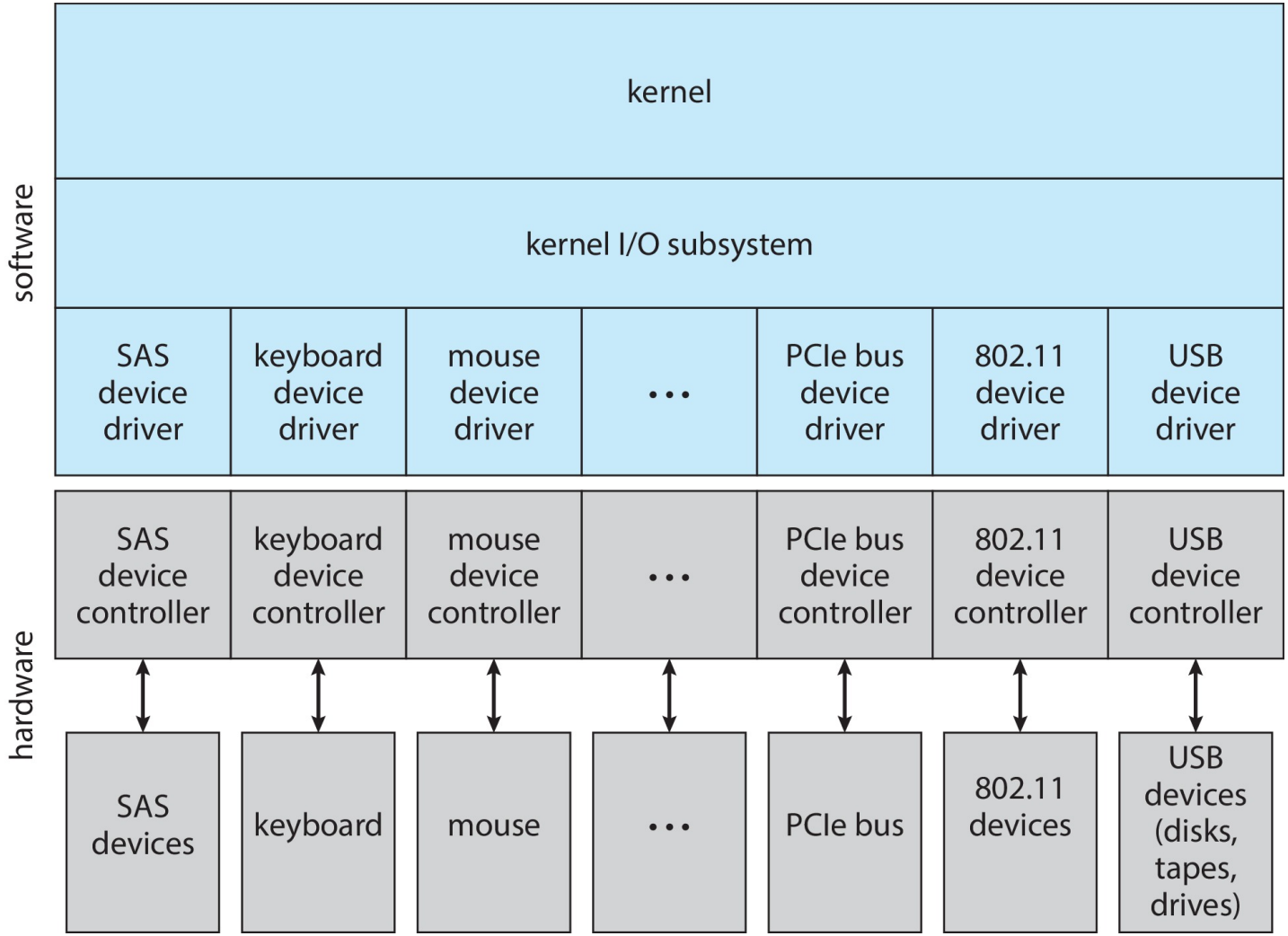
- Subtleties of devices handled by device drivers
- Broadly I/O devices can be grouped by the OS into
  - Block I/O
  - Character I/O (Stream)
  - Network sockets
  - Clocks and Timers

Classes of Devices and Drivers

- OSs classify devices and drivers in to number of categories
  - Block Devices: Disks
    - Abstraction: array of storage blocks
    - A block device can host a file system
    - Commands include **`read, write, seek`**
  - Character Devices: Keyboards, Printers, Text console, Serial ports
    - Abstraction: stream of characters
    - Usually supports **`get, put, open`**, **`close`**, **`read`**, **`write`**
    - Accessed sequentially (in most cases)
  - Network Devices
    - Abstraction: Data packets
  - Others
    - USB, SCSI, FireWire etc

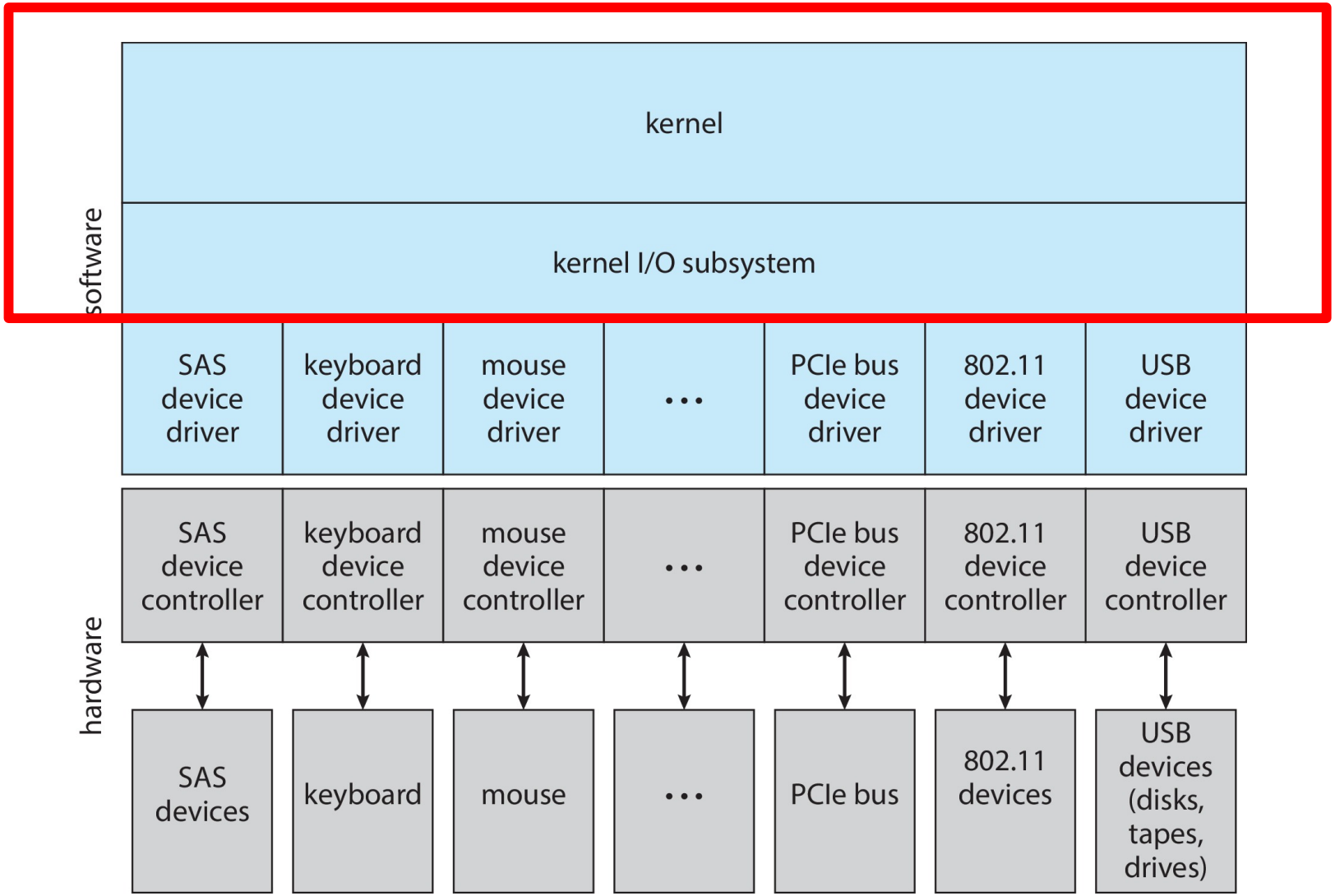
Device Driver - Requirement

A Kernel I/O Structure



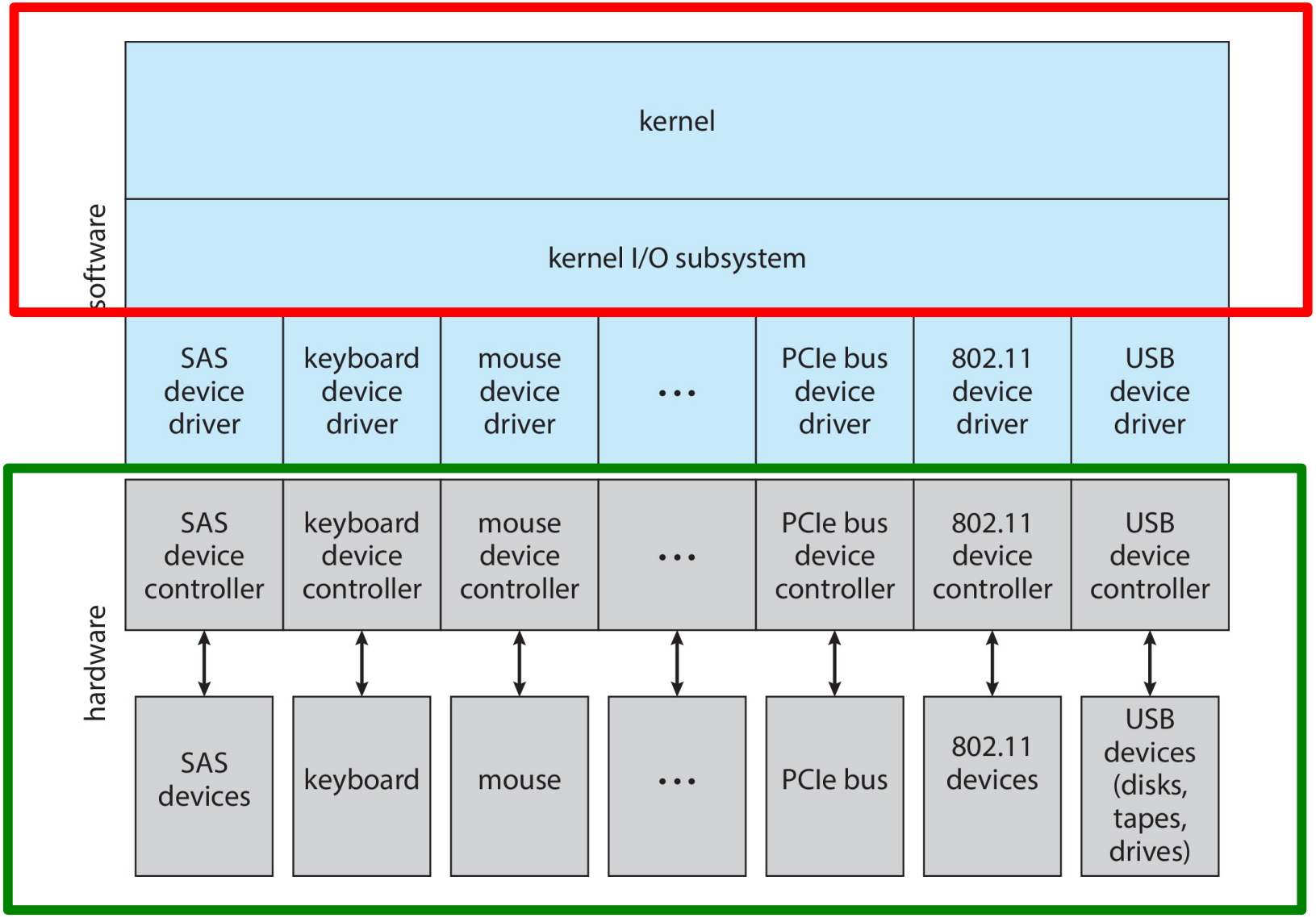
A Kernel I/O Structure

Generic- standard set of functions



A Kernel I/O Structure

Generic standard set of functions

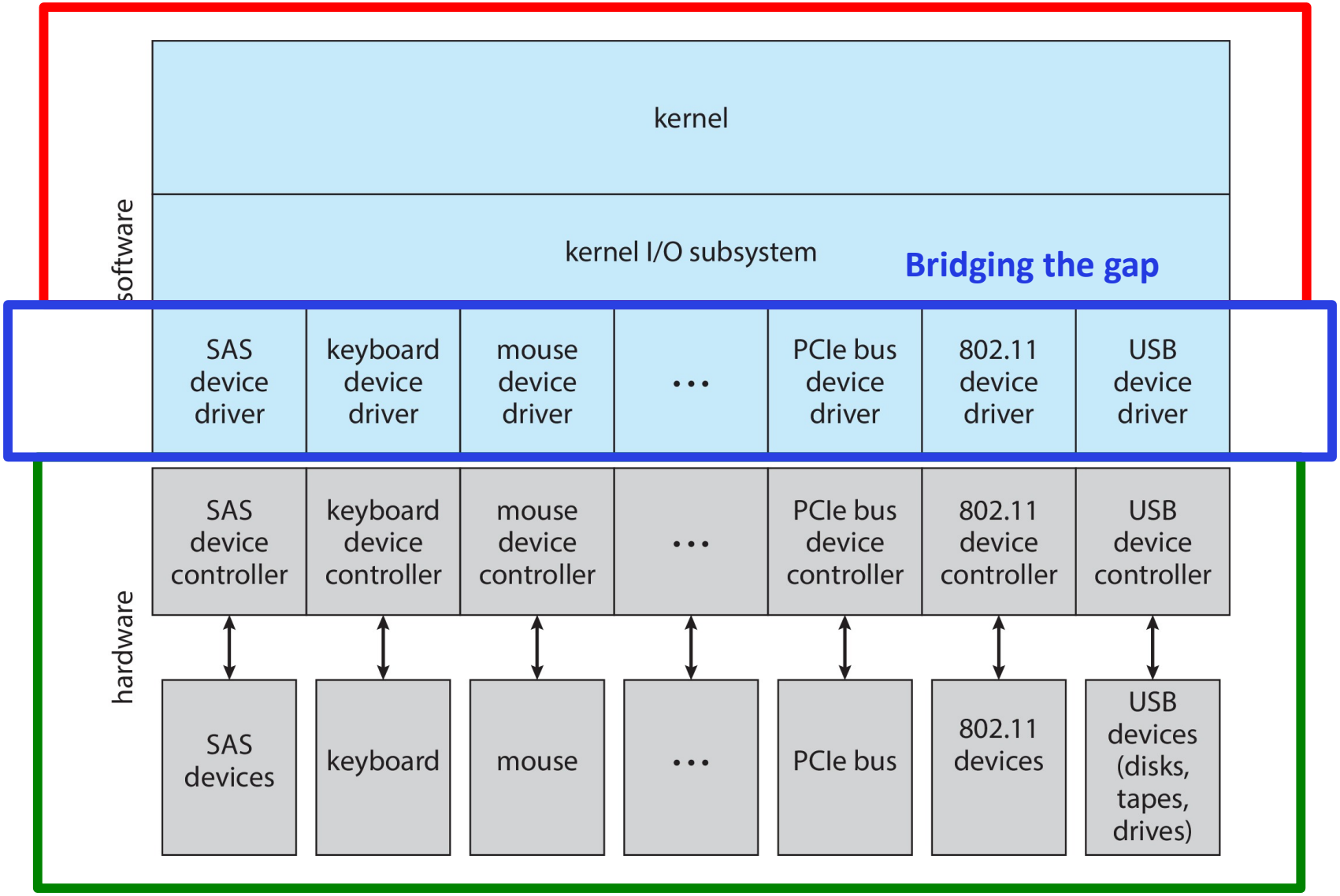


Device specific functions

Book: OS Concepts: Silberschatz, Galvin and Gagne ©2018

A Kernel I/O Structure

Generic standard set of functions



Device specific functions

Book: OS Concepts: Silberschatz, Galvin and Gagne ©2018

Device Driver - Concept

Device Driver - Concept

- Black boxes to hide details of hardware devices
- Use standardized calls
  - Independent of the specific driver
- Map standard calls to device-specific operations
- Can be developed separately from the rest of the kernel
    - Plugged in at runtime when needed
- Implements the *mechanisms* to access the hardware
  - E.g., show a disk as an array of data blocks

Device Driver - Concept

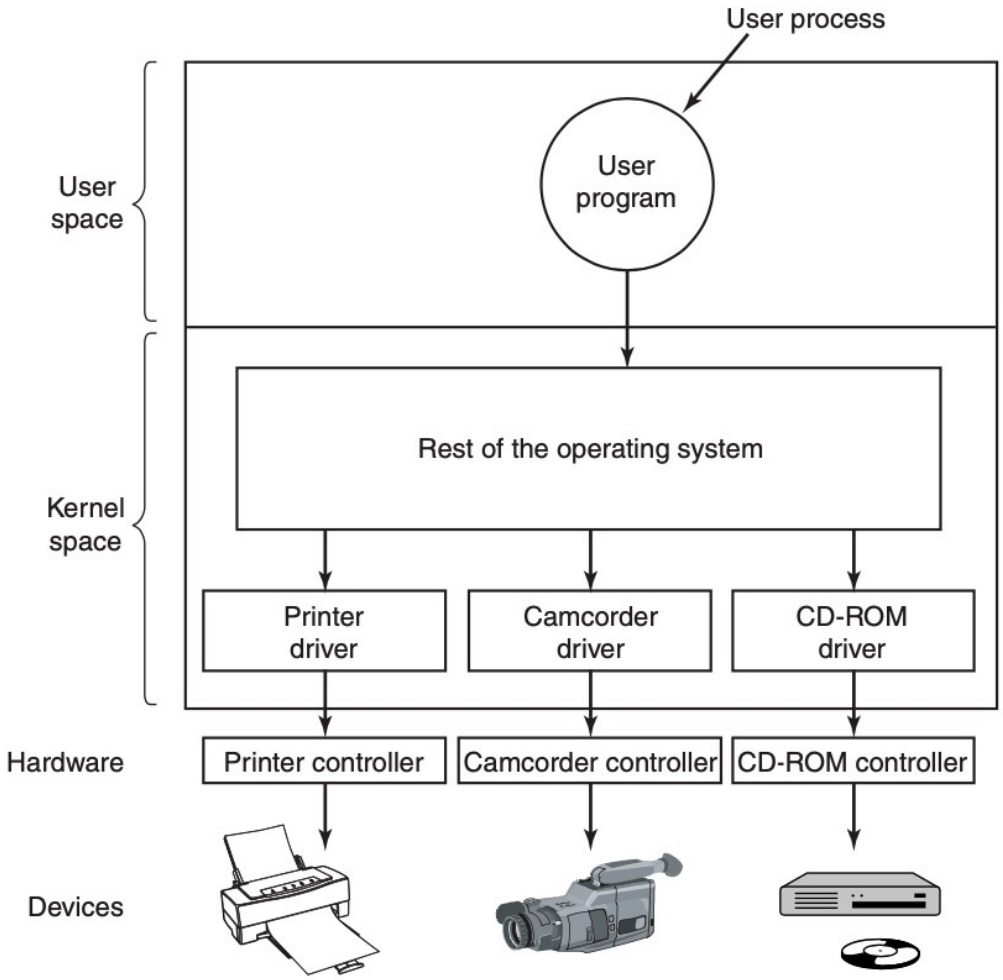
- Device-drivers hide the differences between device controllers and I/O subsystem of the kernel
- I/O system calls encapsulate the behavior of devices in a few generic classes that hide hardware differences from applications
- It benefits device manufactures
  - They can advance or change the design the device
    - Make them compatible with existing controller interface (eg SATA)
    - Or, develop device driver for the device for each OS
      - As different OSs have different I/O subsystem
- Device Drivers accept abstract read and write (or other) requests from the device-independent software above it
- Device Drivers execute the requests

Device Driver Concept

- The number of device-registers and nature of commands vary radically from device to device. For example:
  - Mouse driver has to accept information from the mouse how far it has moved and which button is pressed
  - Disk driver have to know about sectors, track, cylinders, arm motion
- Each I/O device attached to a computer needs some device-specific code for controlling it
  - This code is the device driver
- Device driver is generally written by the device's manufacturer and delivered along with the device
  - Device drivers are supplied for OS
- Each device driver usually handles one device type or class of closely related types

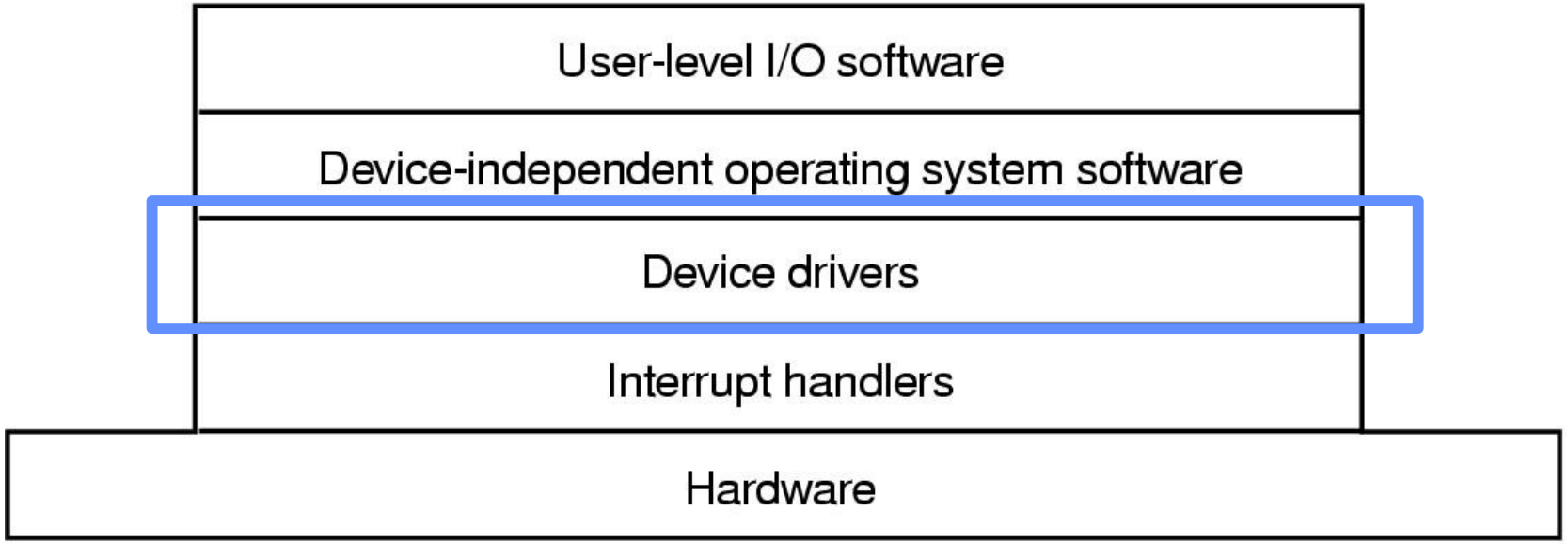
Device Driver – Kernel Space

- Device driver has to be part of OS
  - to access device's hardware



Device Driver Functions

I/O Software Layers



Layers of the I/O Software System

Device Drivers – Functions (1)

- 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: Runs as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete

Device Driver – Functions (2)

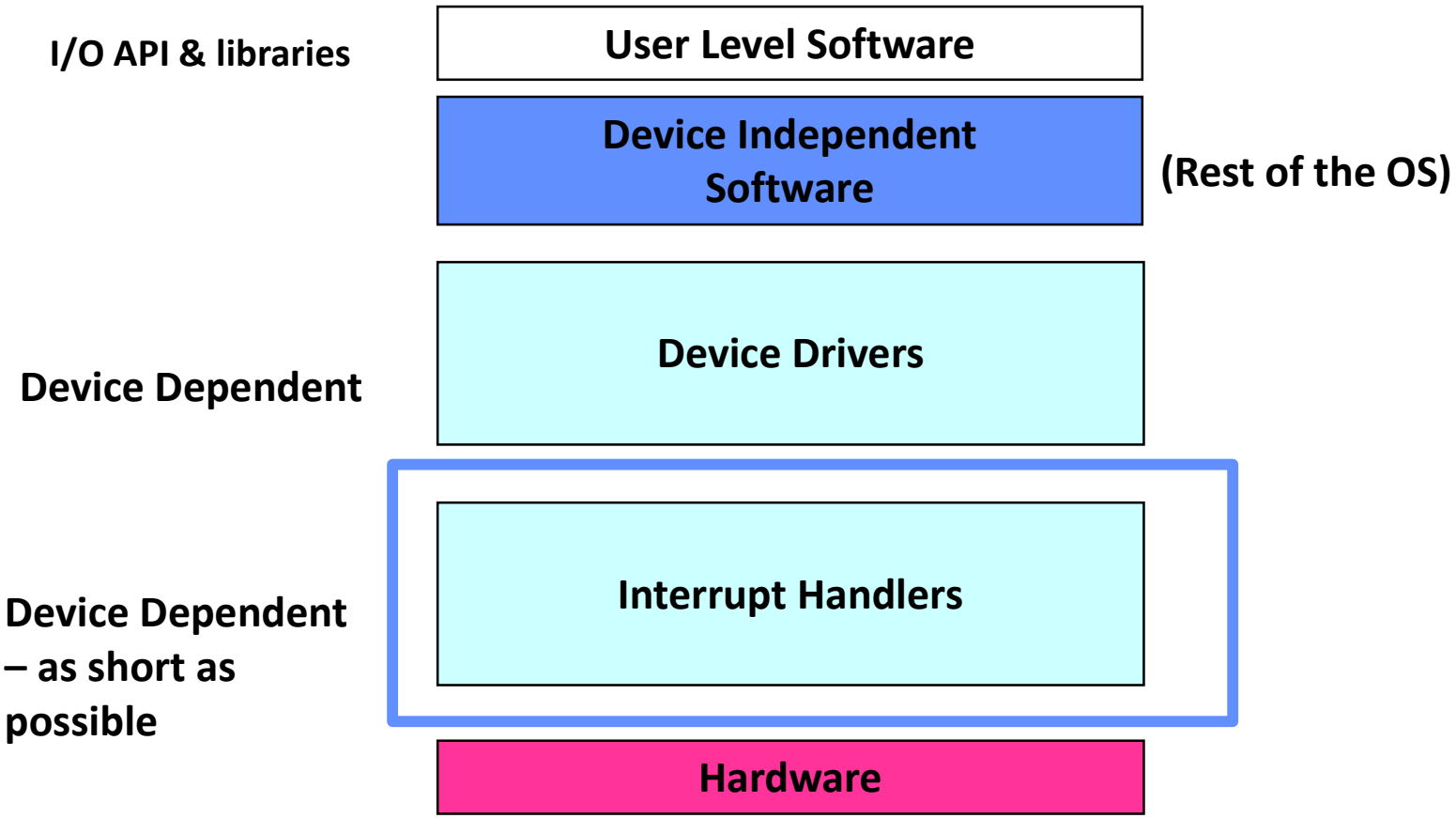
- Initialize the device
- Check if input parameters are valid
  - If not, error is generated
- If inputs are valid, translation is done from abstract to concrete term
  - Example: For a disk driver
    - Convert a linear block number to head, track, sector, and cylinder numbers
- Check if the device is in use
  - If busy, the request is put in the queue
  - If Idle, hardware status is checked – if it can accept the request – there may be need to switch on the device etc
- When device is free, actual control can begin

Device Driver – Functions (2)

- Controlling a device -> issuing a sequence of commands
- Driver issues commands
- To drive a commend
  - Driver writes a command into device's control registers
    - Once the command is accepted by the device, the controller issue more commands
- Device driver waits until controller does some work on it
  - In some operations (eg screen scrolling), waiting is not required
- Blocked driver is awakened by the interrupt by the controller
- Driver checks for errors
  - If there is error, it passes it upwards
- If everything is all right, the driver may have data to pass on to the device-independent software (eg block just read)
- Selects next request for execution

Interrupt Handler

I/O Software "Stack"

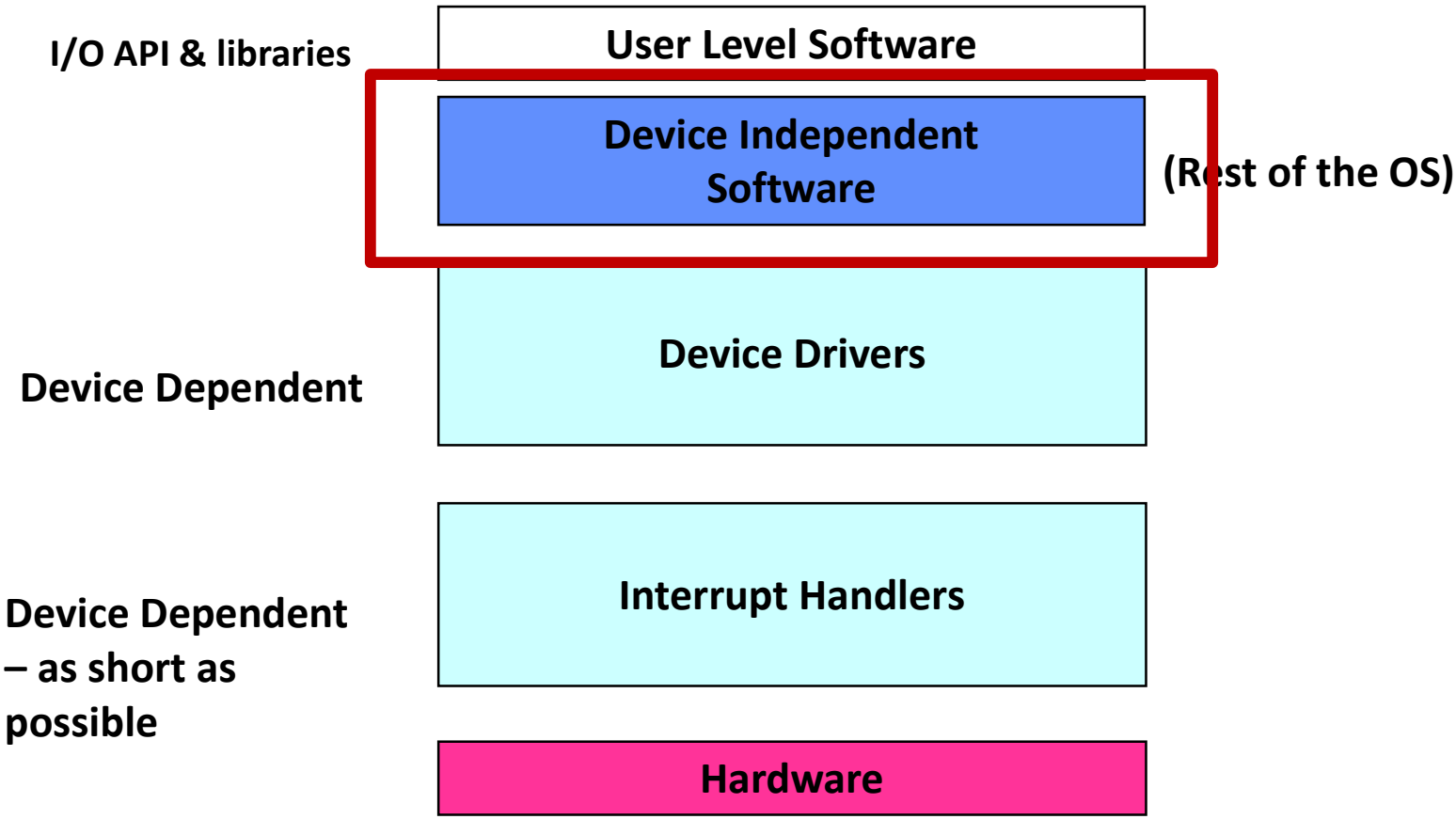


Interrupt Handlers (Repeat)

- Device drivers after starting an operation- blocks
- On receiving an interrupt, the driver unblocks
- Interrupt handling
  - Unlocks the driver (using semaphore)
- A lot of other activities are done as a part of interrupt handling
  - Register Saving
  - Set up context for Interrupt service routine
    - Setting up TLb, MMU, Page Table
  - Set up stack for interrupt service-procedure
  - Ack the interrupt controller
  - Copy registers
  - Run Interrupt service procedure
  - Choose the process to run next
  - Set up MMU context for the process to run next
  - Load and run new process

Device Independent Software

I/O Software "Stack"



Device Independent I/O Software (1)

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

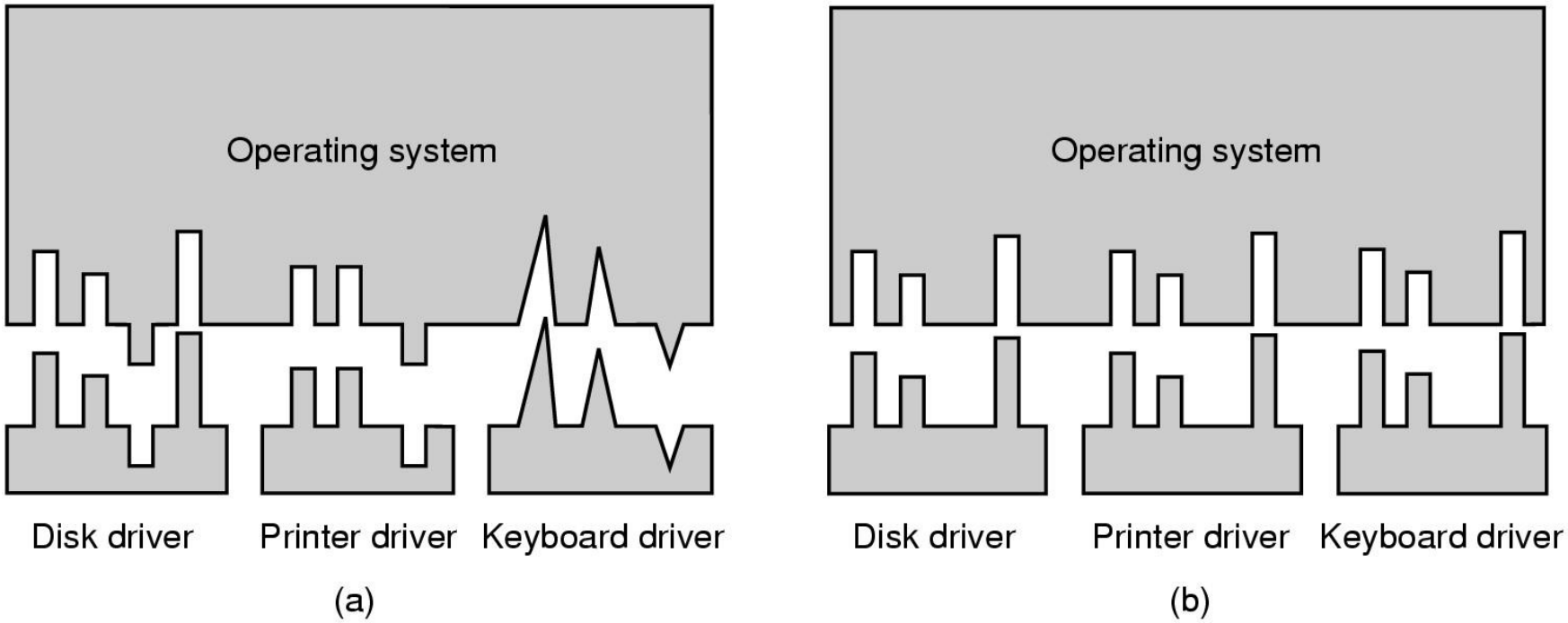
- Function of Device Independent I/O SW
  - to perform the I/O functions that are common to all devices
  - to provide a uniform interface to the user-level software

Device-Independent I/O Software (2) – Uniform Interfacing

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

(a) Without a standard driver interface

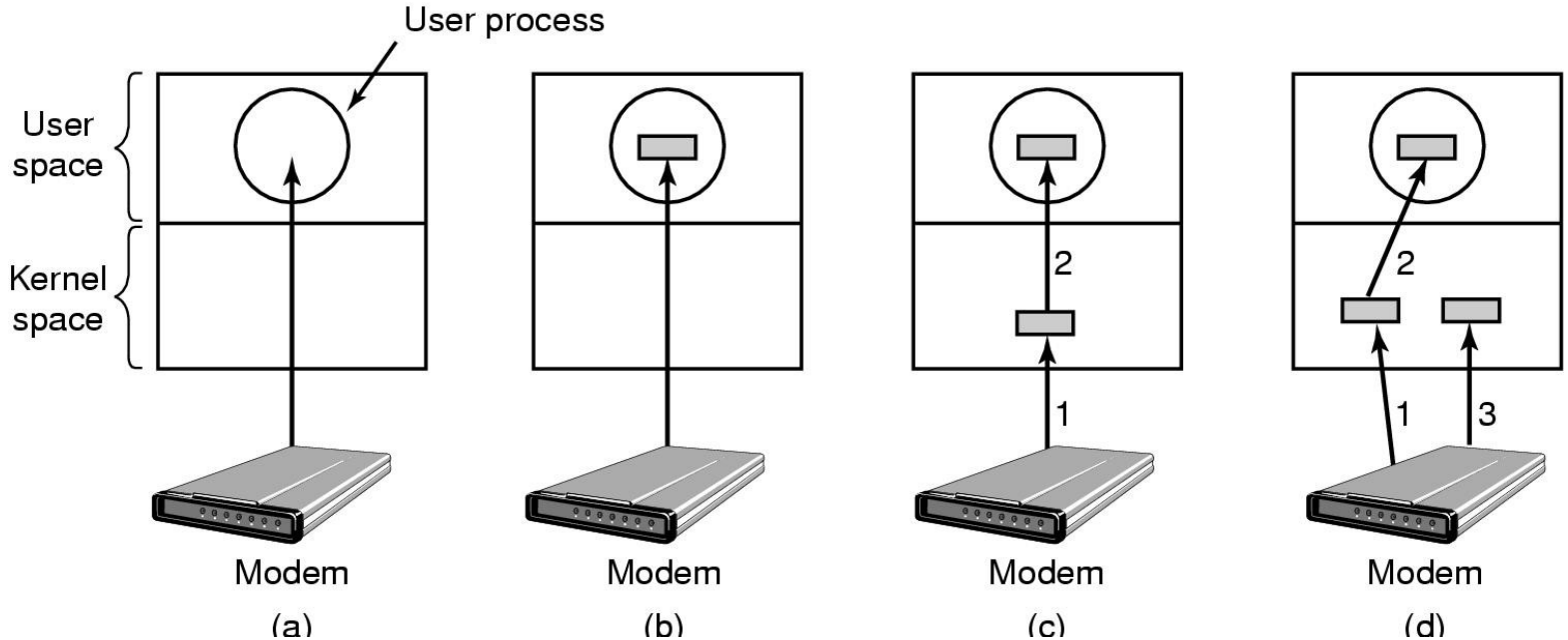
(b) With a standard driver interface



Device-Independent I/O Software (3) – Buffering

- Buffering improves efficiency

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

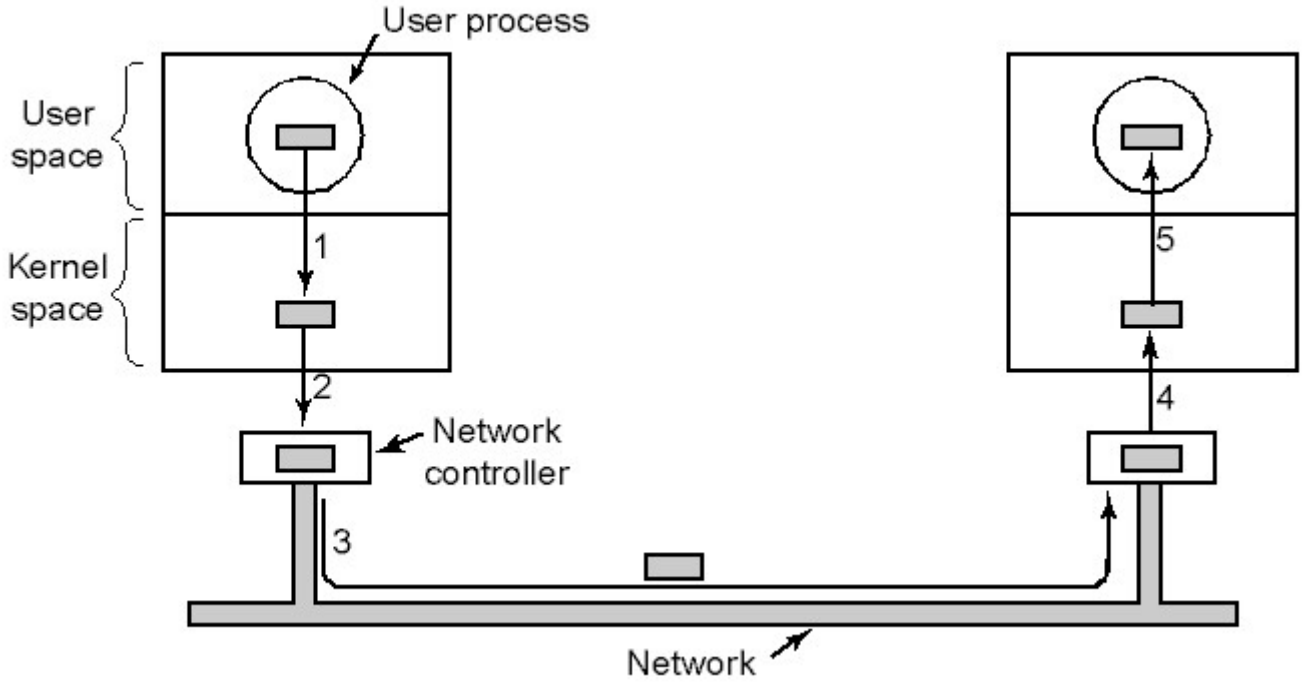


(a) Unbuffered input
(b) Buffering in user space
(c) Buffering in the kernel followed by copying to user space
(d) Double buffering in the kernel

Device-Independent I/O Software – Sending packet on network – buffering

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

Too much buffering can slow down transmission rate



Networking may involve many copies

Error Reporting

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

- Framework for error reporting
  - Device independent
- Errors can happen in many ways
  - Programming Errors – eg writing to a keyboard etc
  - Trying to write to a disk block that has been damaged
  - Trying to read from a device that is switched off
- Device specific errors may be handled by Device Driver
  - If it is not able to handle the error, it passes problem to device independent software. Solutions:
    - This software may inform user about the problem – eg using dialog box
    - Ignoring the error
    - Retry
    - Killing the process
    - Call fail with error code

Allocating and Releasing Dedicated Devices

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

- Some devices can be used by a single process at a time
- OS must examine requests – accept or reject them
- There are many ways to do this
  - Perform open on a special file for the device directly
    - If device is unavailable – open fails
  - Special mechanisms for requesting and releasing dedicated devices
    - Block the caller process instead of failing – if device is busy
      - Put the process in the queue
    - When device is available, the first process in the queue is allowed to use it

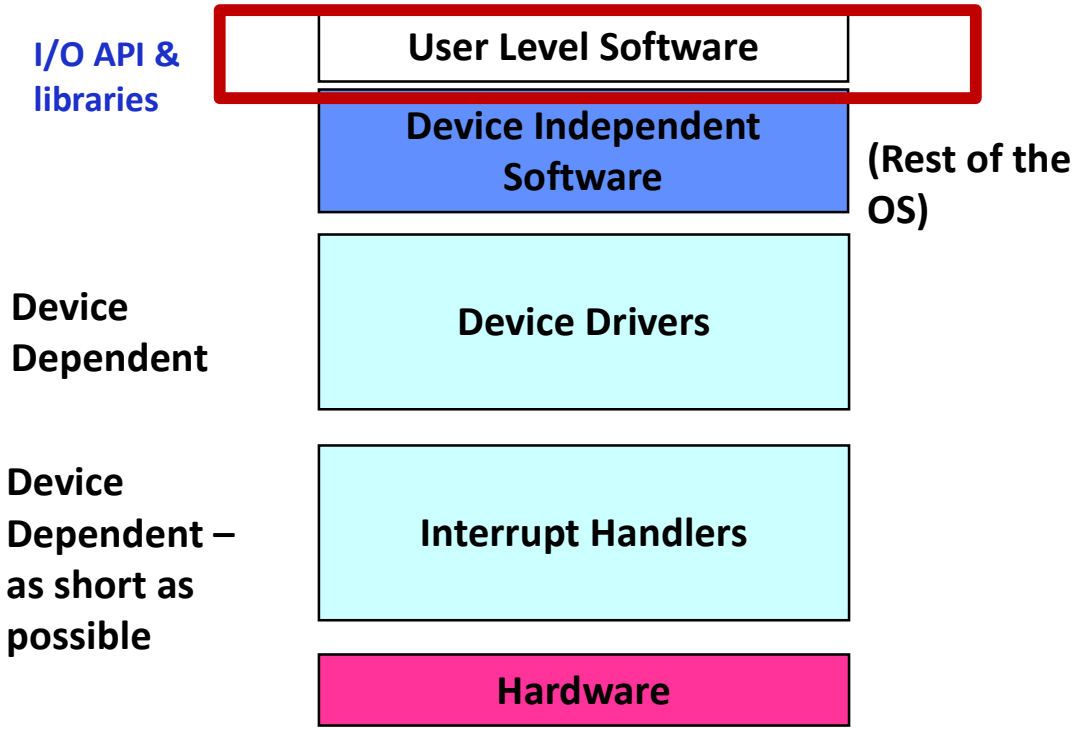
Device Independent block size

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

- Different disks may have different sector sizes
- Device independent software hides this
- It provides uniform block size to higher layers
- Example: Treating several sectors as a single logical block
  - Allows higher layers to deal with abstract devices

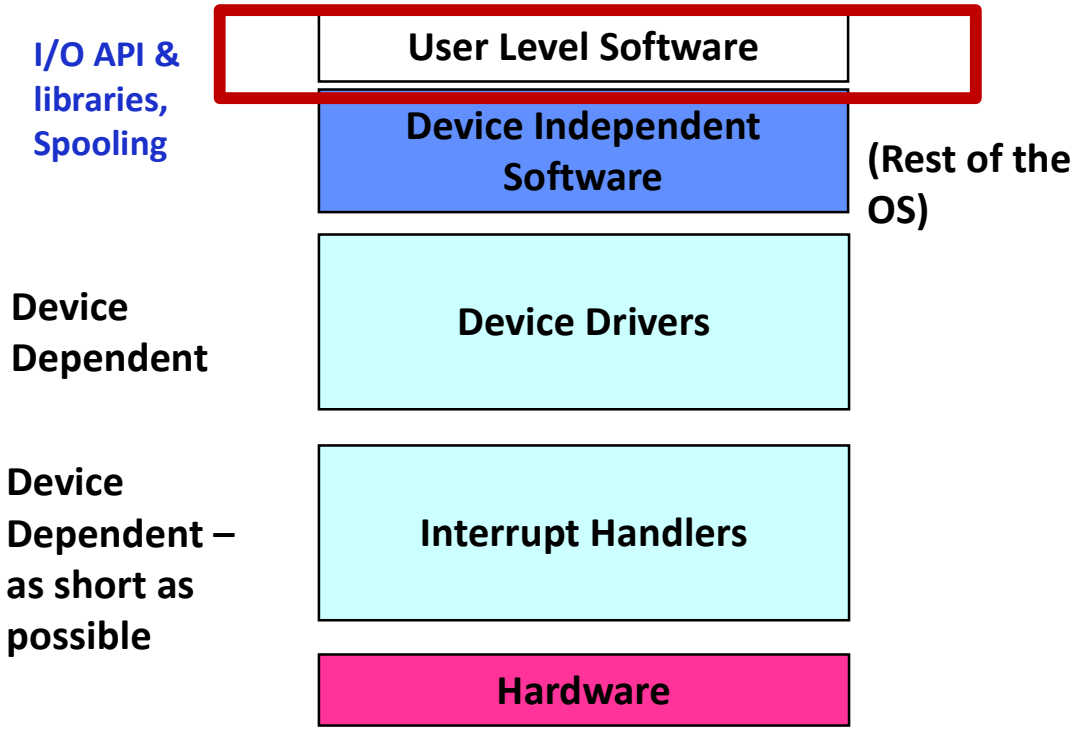
User Space I/O Software - Libraries

User Space I/O software - Libraries



- Libraries linked with user programs
- System calls (including I/O system calls) are made by library procedures
- Ex: printf("The square of %3d is %6d\n", i, i*i);

User Space I/O software - Spooling

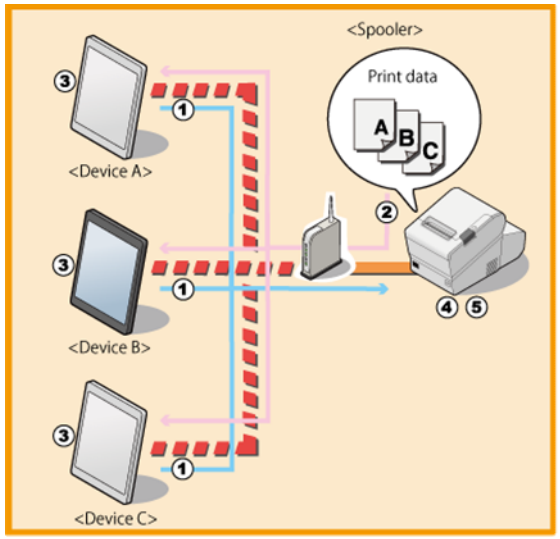


- A way of dealing with dedicated I/O devices in a multiprogramming system
  - Spooled device – Printer
- Consider a process opens a character special file associated with the printer and does nothing
  - Blocking other processes

Spooling

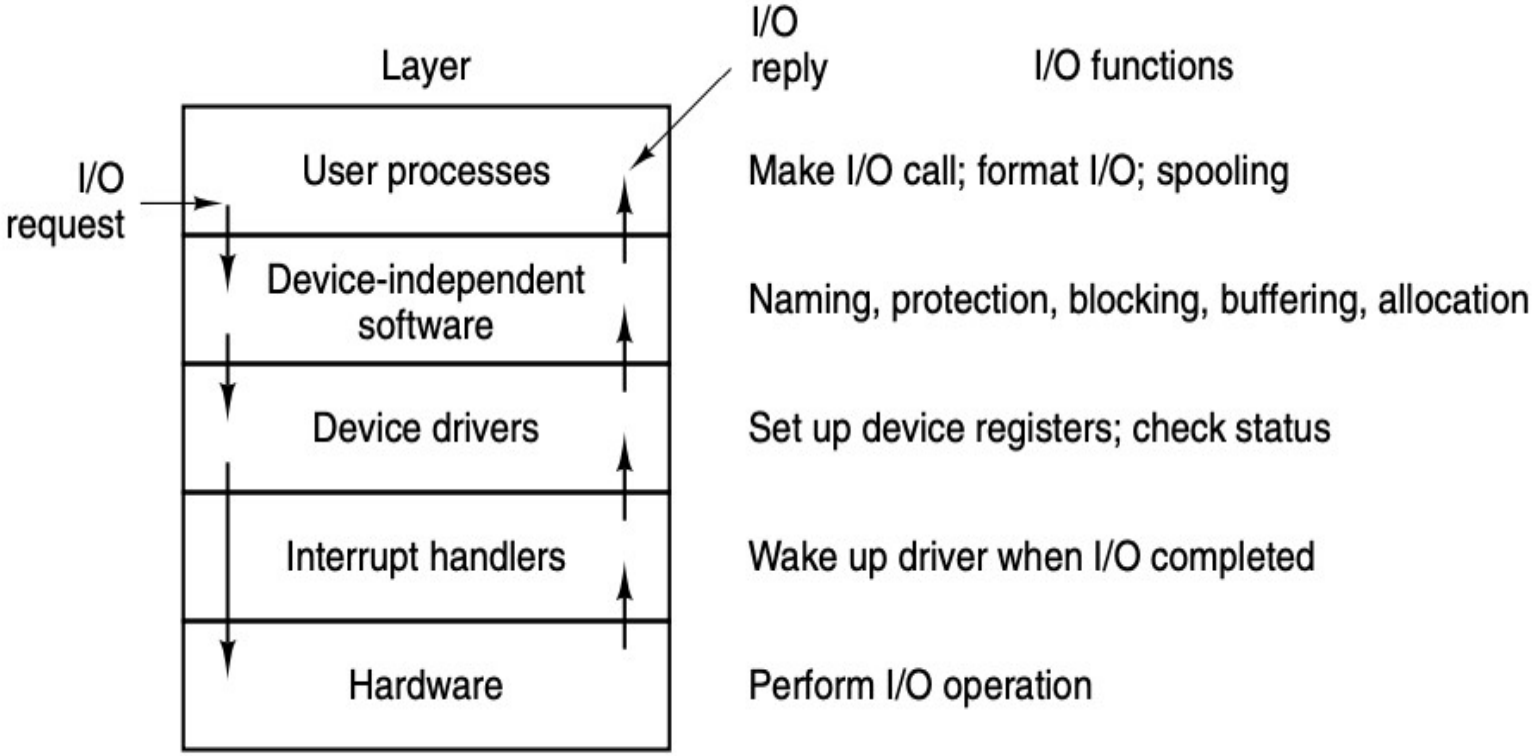
- Spooling is a method for queuing and holding data for input or output
- Uses special process called a daemon and special directory called spooling directory
- To print a file, a process
  - first generates the entire file to be printed
  - Puts the file in the spooling directory
- Daemon is the only process that has permission to use printer's special file
  - to print the files in the directory
- It protects use of the special file directly by users

Spooling – Example (Aside)



- Devices A to C request printing of data A to C almost simultaneously.
- The spooler function saves the print data in the spooler in the order it receives it, and returns the printing results to each device.
- Each device application completes the printing process and executes the next operation.
- This product successively prints print data A to C.
- The spooler function saves the printing results in the log.

Lecture Summary: Main Functions of layers of I/O System



Life Cycle of An I/O Request

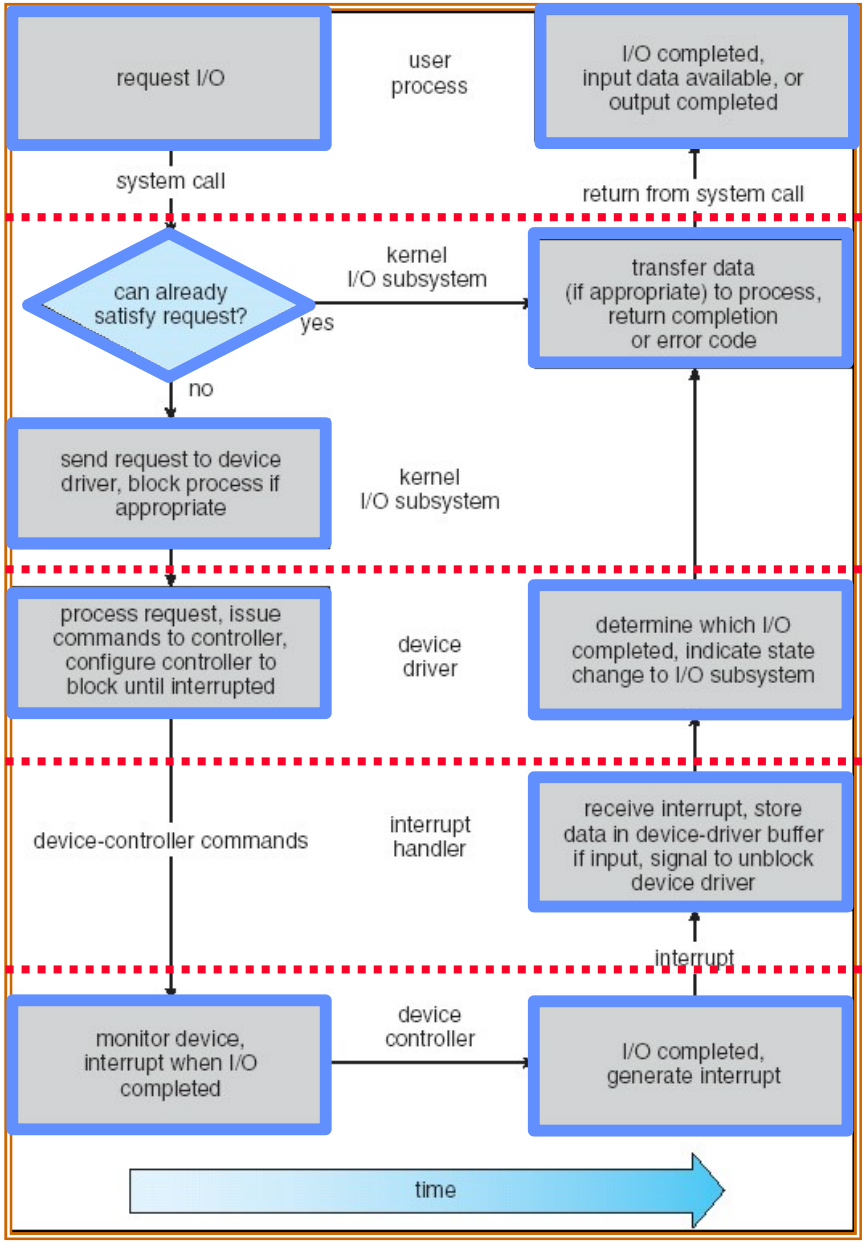
User Program

Kernel I/O Subsystem
Device Independent software

Device Driver Top Half

Device Driver Bottom Half

Device Hardware



Extra Slide

Device Drivers - USB

- Sometimes widely different devices are based on the same underlying technology
- Consider USB (Universal Serial Bus) Devices
  - Mouse, Keyboards, mini-fans, wireless network cards, keyboards, memory sticks, external disks, robots, credit card readers, bar code scanner etc
- They all use USB ..yet do different things
- USB drivers are stacked
  - Bottom layer – like Layer 2 of TCP/IP
  - Middle layer – handle packet transmission and reception
  - Higher layer = APIs such as mass storage, cameras etc
    - Separate device drivers