CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

Materials in these slides have been borrowed from textbooks and existing operating systems courses

March 26, Communication with I/O Devices

What good is a computer without any I/O devices?

- keyboard, display, disks

We want: - H/W that will let us plug in different devices

- OS that can interact with different combinations

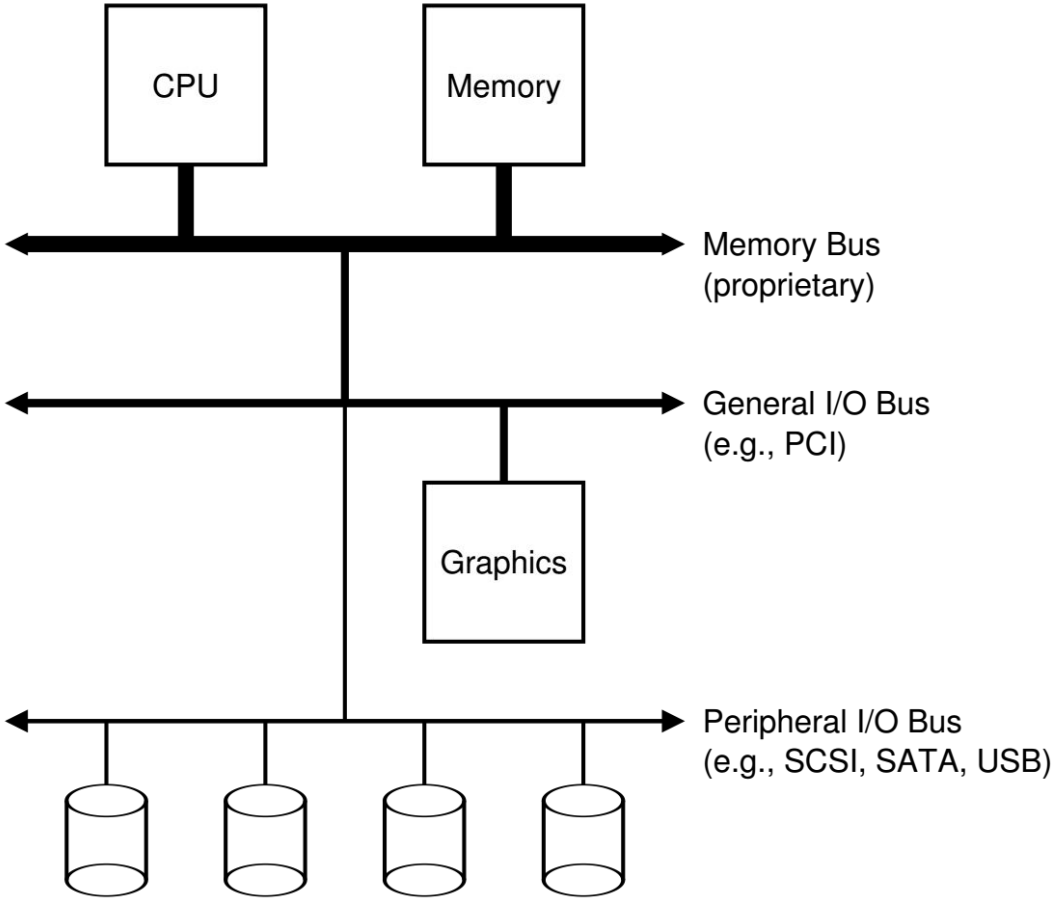
I/O is critical to computer system to interact with systems.

Issue : How should I/O be integrated into systems?

What are the general mechanisms?

How can we make the efficiently?

H/w Support for I/O



- CPU is attached to the main memory of the system via some kind of memory bus.
- Some devices are connected to the system via a general I/O bus.
- Point of connection to the system: port

Need for Hierarchical Bus ?

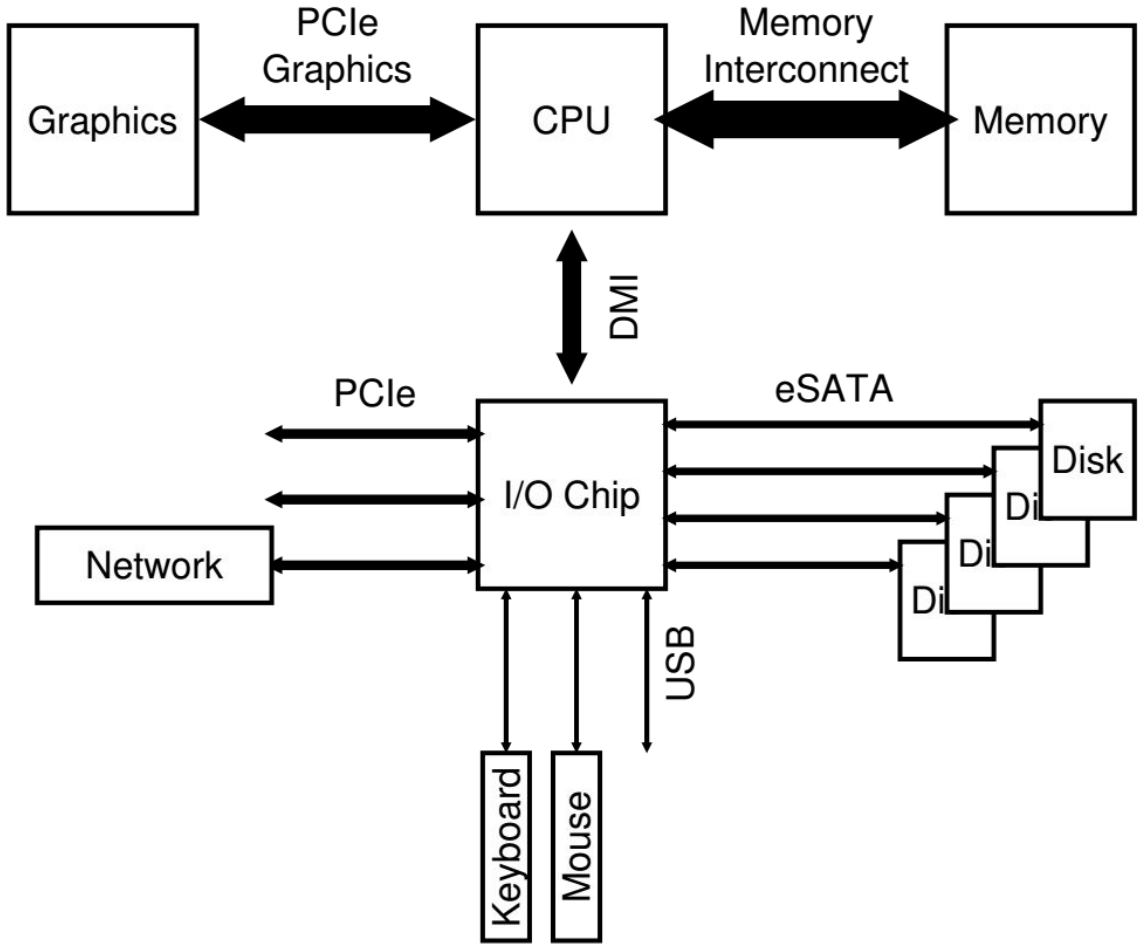
Physics, and cost. The faster a bus is, the shorter it must be;

High-performance memory bus does not have much room to plug devices and such into it.

In addition, engineering a bus for high performance is quite costly.

Hierarchical approach -- components that demand high performance (such as the graphics card) are nearer the CPU. Lower performance components are further away.

The benefits of placing disks and other slow devices on a peripheral bus are manifold; in particular, you can place a large number of devices on it.

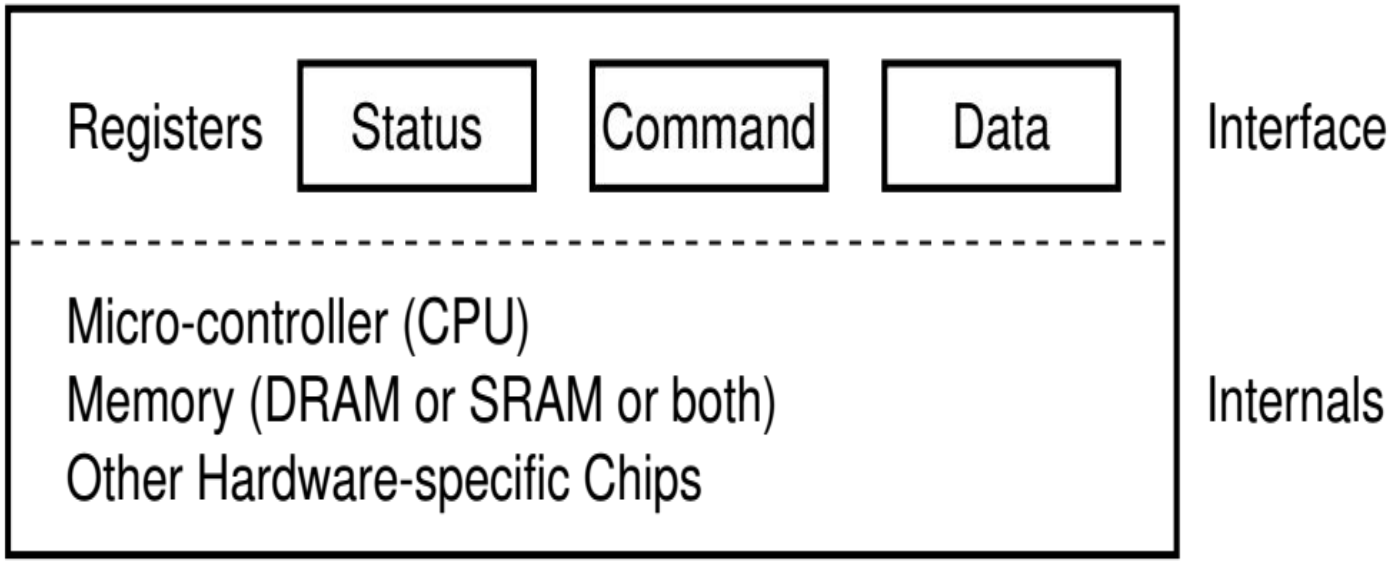


Modern systems increasingly use specialized chipsets and faster point-to-point interconnects to improve performance.

Simple device model

- **Block devices** store a set of numbered blocks (disks)
- **Character devices** produce /consume stream of bytes(keyboard)
- Devices expose an interface of memory registers
  - Current status of device
  - Command to execute
  - Data to transfer

- The internals of device are usually hidden



Canonical Device

- **status register**
  - See 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

By reading and writing above three registers,
the operating system can control device behavior.

How does OS read/write to registers

- How does OS read/write to registers like status and command?
- **Explicit I/O instructions**

–E.g., on x86,inandoutinstructions can be used to read and write to specific registers on a device

–Privileged instructions accessed by OS

- **Memory mapped I/O**

–Device makes registers appear like memory locations

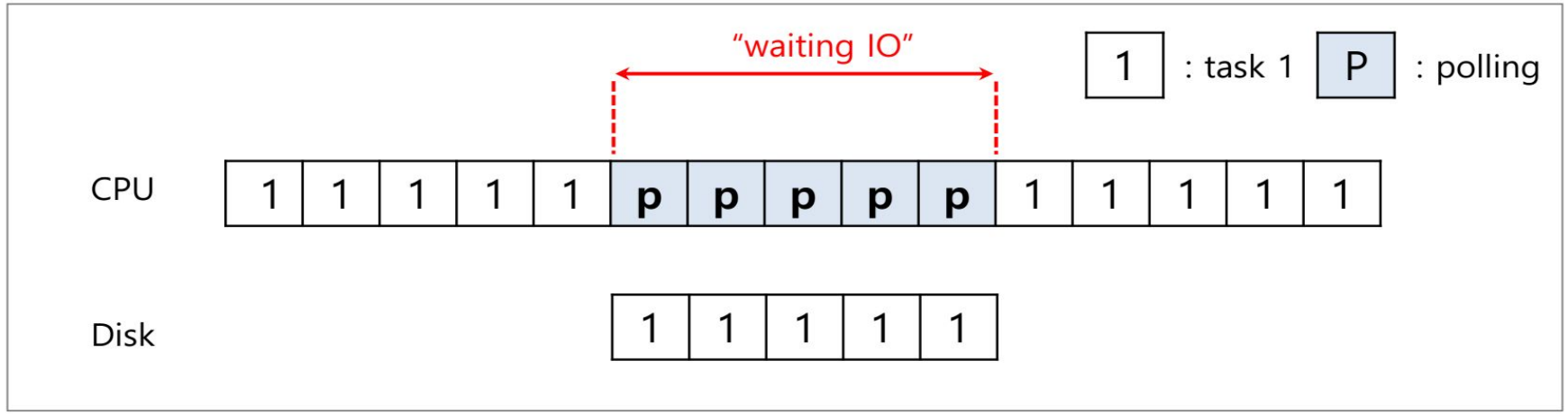
–OS simply reads and writes from memory

–Memory hardware routes accesses to these special memory addresses to devices

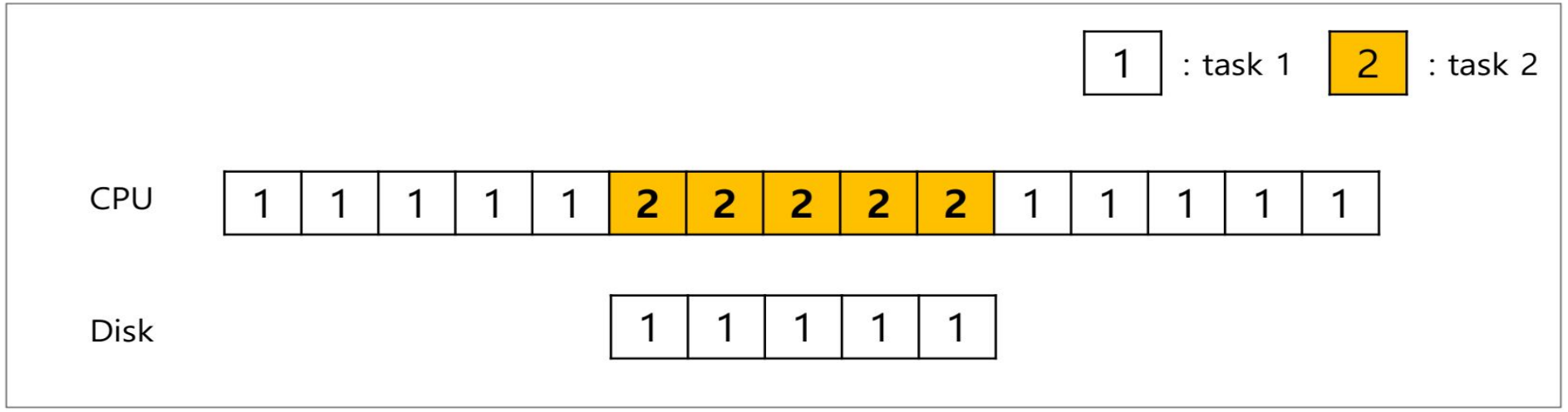
Example Write Protocol

```
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
```

- Operating system waits until the device is ready by **repeatedly** reading the status register.
  - Positive aspect is simple and working.
  - **However, it wastes CPU time just waiting for the device**.
    - Switching to another ready process is better utilizing the CPU.



- **Put the I/O request process to sleep** and context switch to another.
- When the device is finished, wake the process waiting for the I/O by **interrupt**.
  - Positive aspect is allow to **CPU and the disk are properly utilized.**

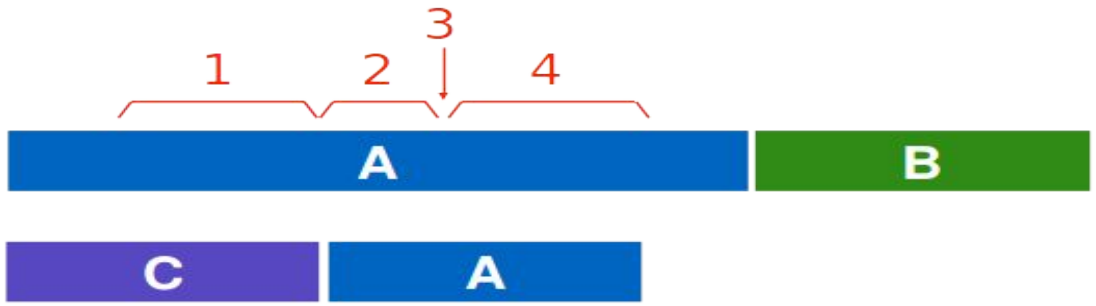


CPU A

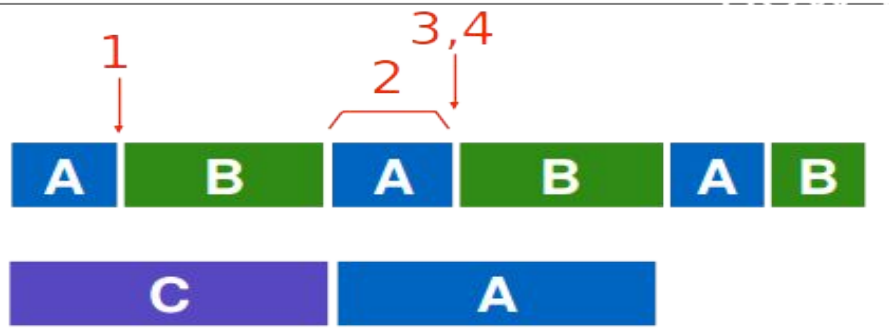
Disk C

```
while (STATUS == BUSY)              // 1
    ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
    ;
```

Spinning Interrupts ?




```
while (STATUS == BUSY)              // 1
      wait for interrupt;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
      wait for interrupt;
```




```
while (STATUS == BUSY)              // 1
      wait for interrupt;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
      wait for interrupt;
```

Interrupt Handler

- Interrupt switches process to kernel mode
- Interrupt Descriptor Table (IDT) stores pointers to interrupt handlers (interrupt service routines)
  - Interrupt (IRQ) number identifies the interrupt handler to run for a device
- Interrupt handler acts upon device notification, unblocks the process waiting for I/O (if any), and starts next I/O request (if any pending)
- Handling interrupts imposes kernel mode transition overheads
  - Note: polling may be faster than interrupts if device is fast

Interrupts Vs Polling

Are interrupts ever worse than polling?

Fast device: Better to spin than take interrupt overhead

Device time unknown? Hybrid approach (spin then use interrupts)

Flood of interrupts arrive

Can lead to livelock (always handling interrupts)

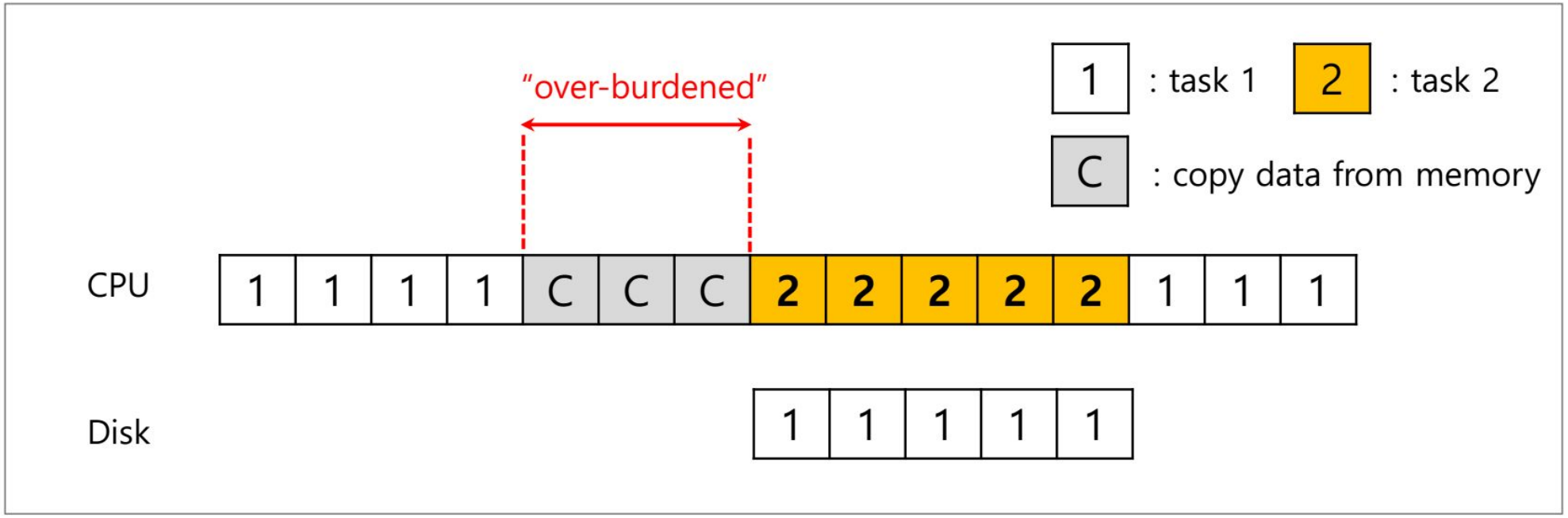
Better to ignore interrupts while make some progress handling them

Other improvement

Interrupt coalescing (batch together several interrupts)

Programmed I/O

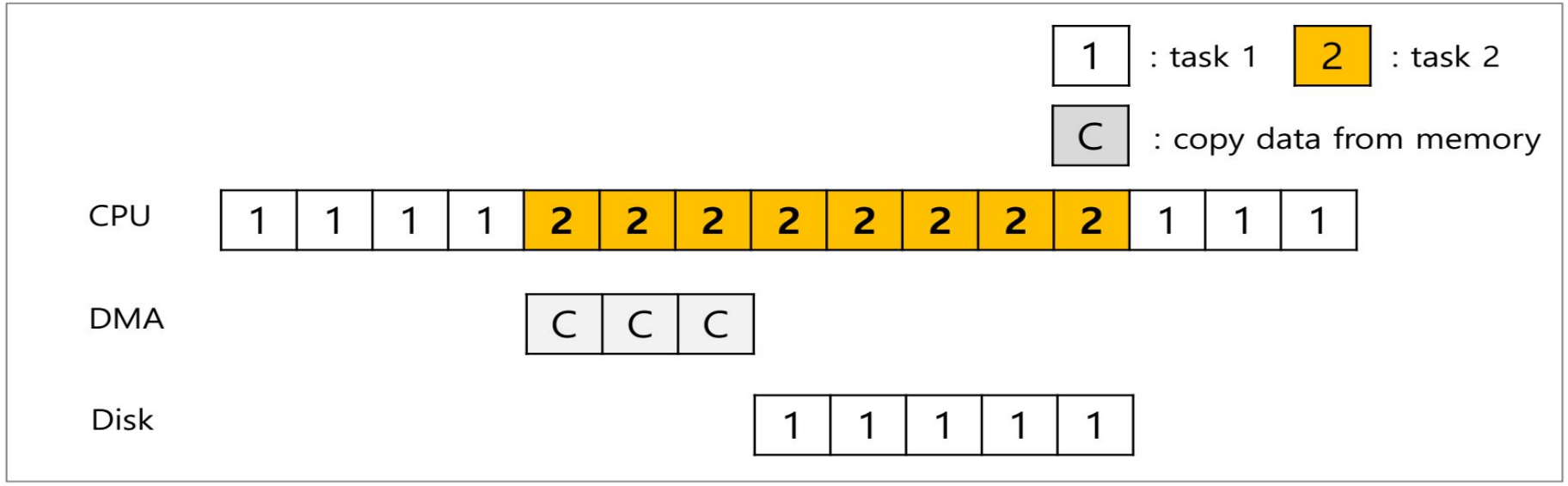
- CPU **wastes a lot of time** to copy the *a large chunk of data* from memory to the device. **CPU explicitly copies data to/from device**



- CPU cycles wasted in copying data to/from device
- Instead, a special piece of hardware (DMA engine) within the system orchestrates transfers between devices and main memory without much CPU intervention.
  - CPU gives DMA engine the memory location of data
  - In case of read, interrupt raised after DMA completes
  - In case of write, disk starts writing after DMA completes

Direct Memory Access

- **Copy data** in memory by knowing "where the data lives in memory, how much data to copy"
- When completed, DMA raises an interrupt, I/O begins on Disk.

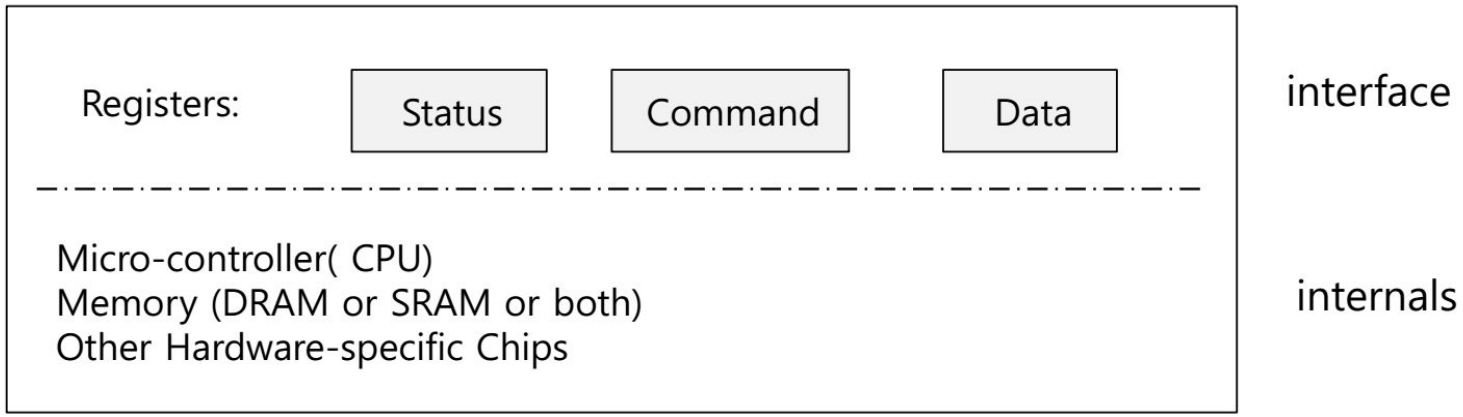


Protocol Variants

Control: special instructions vs. memory-mapped I/O

Status checks: polling vs. interrupts

Data: PIO vs. DMA



Variety is a Challenge

Problem:

Many, many devices -- each has its own protocol

How can we avoid writing a slightly different OS for each H/W combination?

Write device driver for each device

Drivers are 70% of Linux source code

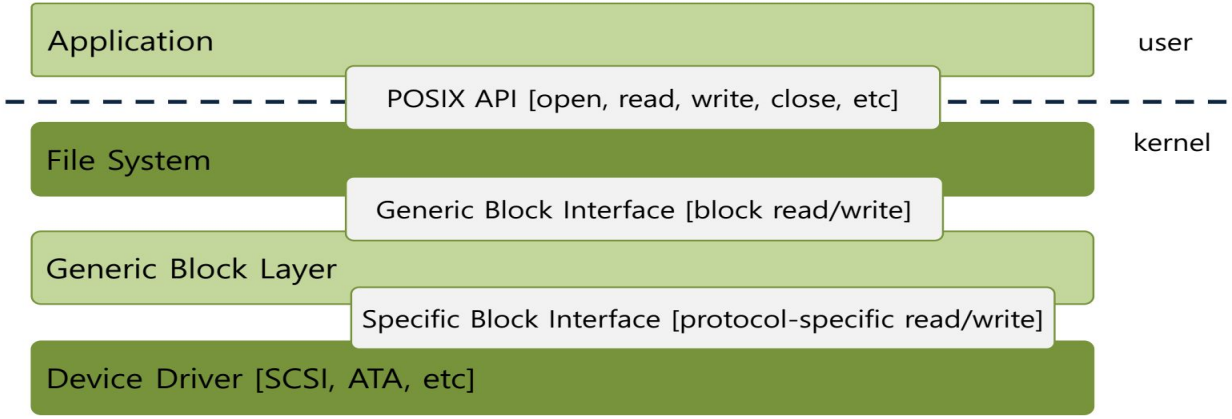
Device Drivers

•Device driver: part of OS code that talks to specific device, gives commands, handles interrupts etc.

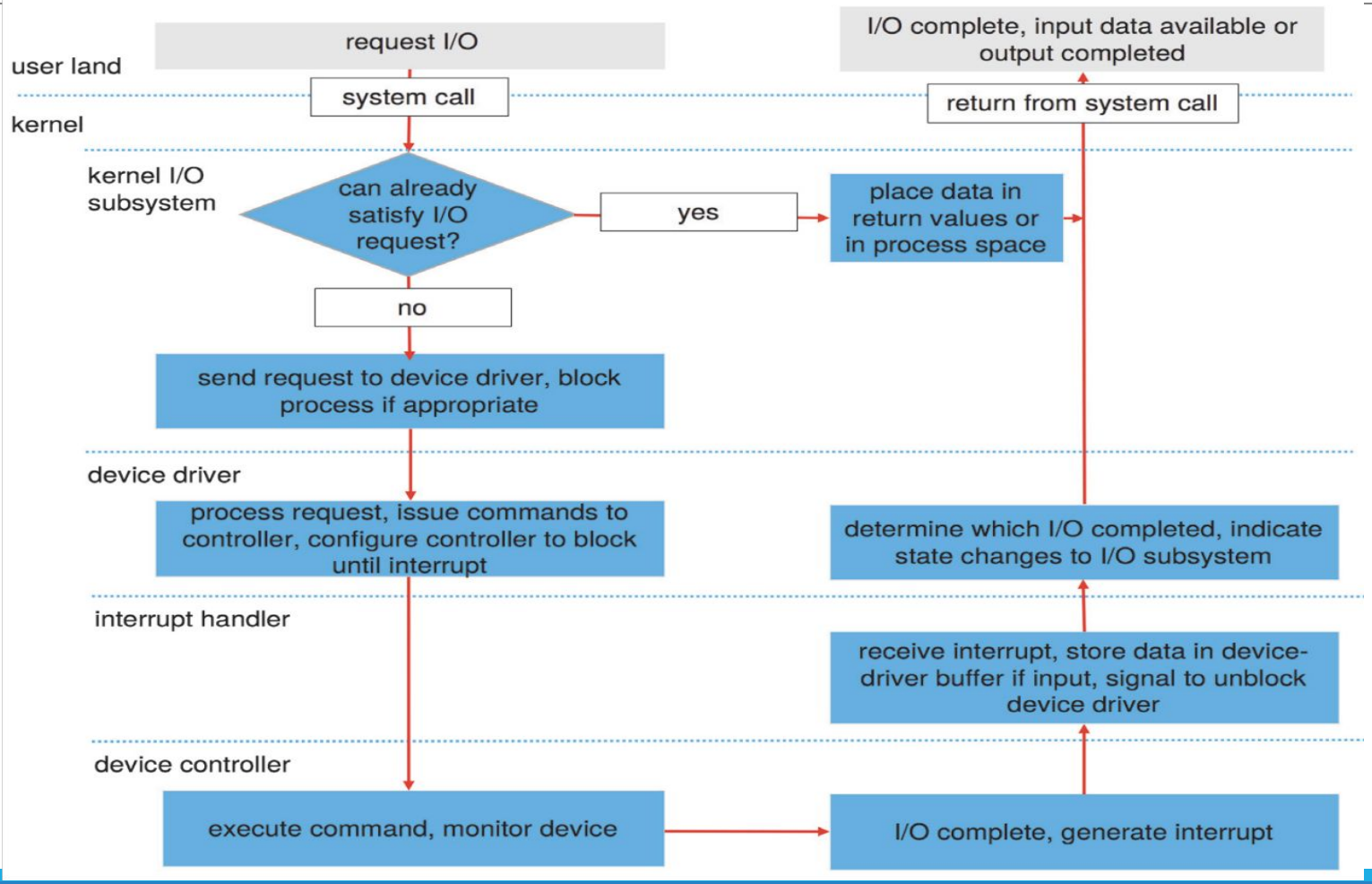
•Most OS code abstracts the device details

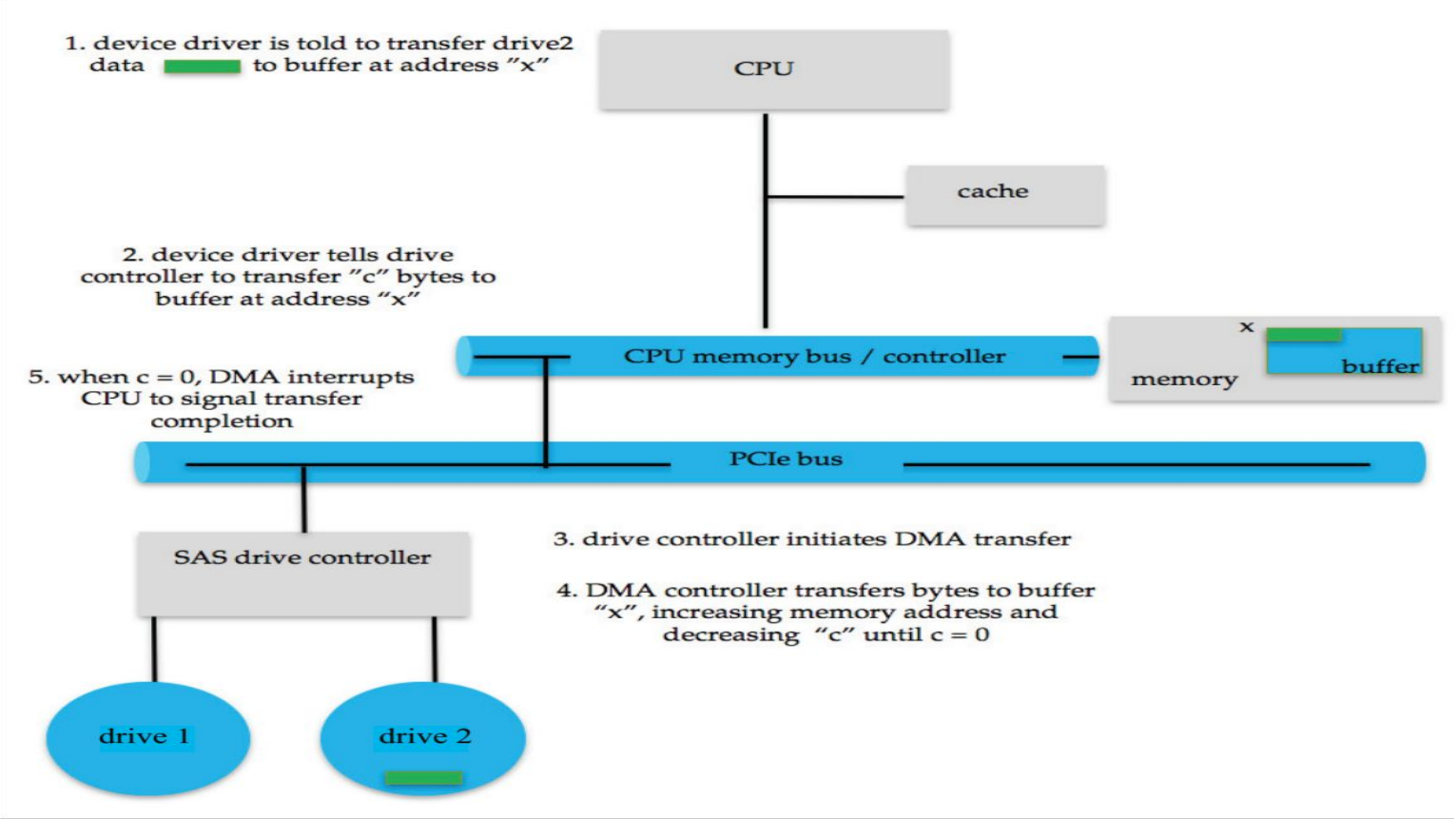
–E.g., file system code is written on top of a generic block interface

- File system specifics of which disk class it is using.
  - Ex) It issues **block read** and **write** request to the generic block layer.



Life cycle of a I/O request




1. device driver is told to transfer drive2 data to buffer at address "x"
CPU
cache
2. device driver tells drive controller to transfer "c" bytes to buffer at address "x"
x
CPU memory bus / controller
memory
buffer
5. when c = 0, DMA interrupts CPU to signal transfer completion
PCIe bus
SAS drive controller
3. drive controller initiates DMA transfer
4. DMA controller transfers bytes to buffer "x", increasing memory address and decreasing "c" until c = 0
drive 1
drive 2

Simple IDE disk driver

- Four types of register
  - Control, command block, status and error
  - Memory mapped IO
  - `in` and `out` I/O instruction

- Control Register:

  Address 0x3F6 = 0x80 (0000 1RE0): R=reset, E=0 means "enable interrupt”

- Command Block Registers:

  Address 0x1F0 = Data Port

  Address 0x1F1 = Error

  Address 0x1F2 = Sector Count

  Address 0x1F3 = LBA low byte

  Address 0x1F4 = LBA mid byte

  Address 0x1F5 = LBA hi byte

  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive

  Address 0x1F7 = Command/status

- Status Register (Address 0x1F7):

7	6	5	4	3	2	1	0
BUSY	READY	FAULT	SEEK	DRQ	CORR	IDDEX	ERROR

- Error Register (Address 0x1F1): (check when Status ERROR==1)

7	6	5	4	3	2	1	0
BBK	UNC	MC	IDNF	MCR	ABRT	T0NF	AMNF

  - BBK = Bad Block
  - UNC = Uncorrectable data error
  - MC = Media Changed
  - IDNF = ID mark Not Found
  - MCR = Media Change Requested
  - ABRT = Command aborted
  - T0NF = Track 0 Not Found
  - AMNF = Address Mark Not Found

- **Wait for drive to be ready**. Read Status Register (0x1F7) until drive is not busy and REA DY.
- **Write parameters to command registers**. Write the sector count, logical block address (LBA) of the sectors to be accessed, and drive number (master=0x00 or slave=0x10, as I DE permits just two drives) to command registers (0x1F2-0x1F6).
- **Start the I/O**. by issuing read/write to command register. Write READ—WRITE comman d to command register (0x1F7).
- **Data transfer (for writes)**: Wait until drive status is READY and DRQ (drive request for d ata); write data to data port.
- **Handle interrupts**. In the simplest case, handle an interrupt for each sector transferred; more complex approaches allow batching and thus one final interrupt when the entire t ransfer is complete.
- Error handling. After each operation, read the status register. If the ERROR bit is on, rea d the error register for details.