Lecture

Operating System

36. I/O Devices

Professor Dr. Michael Mächtel

36. I/O Devices

1. **System Architecture**
2. **Devices**
3. **Case Study**

36. I/O Devices

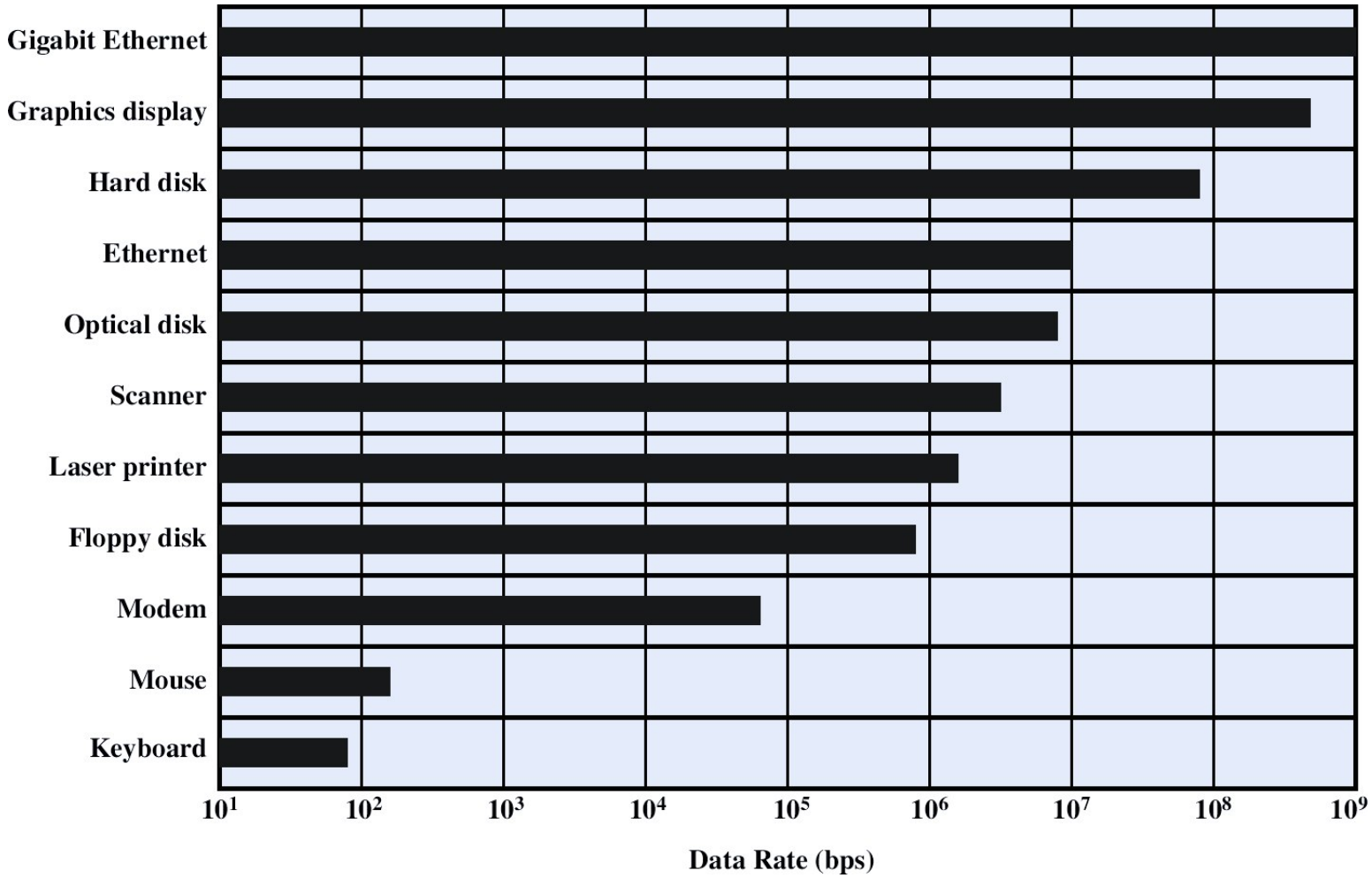
1. **System Architecture**
2. **Devices**
3. **Case Study**

Input/Output (I/O) Devices

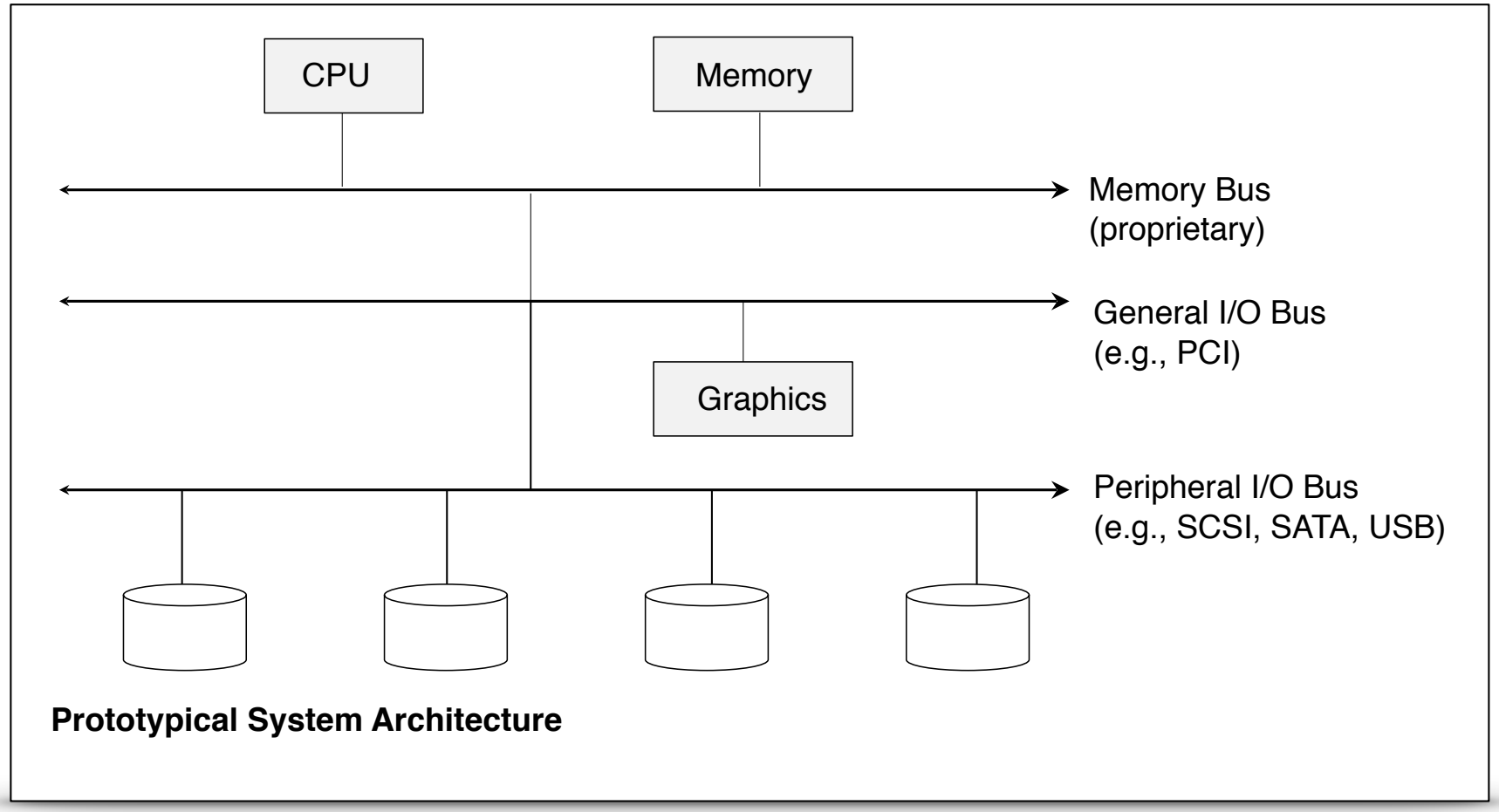
- What good is a computer without any I/O devices?
  - keyboard, display, disks
  - I/O is **critical** to computer system to **interact with systems**.
- We want:
  - H/W that will let us plug in different devices
  - OS that can interact with different combinations
- Issue :
  - How should I/O be integrated into systems?
  - What are the general mechanisms?
  - How can we make it efficiently?

I/O Devices are different!

- e.g. data rate:



System Architecture



Prototypical System Architecture

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**.

Bus Architecture

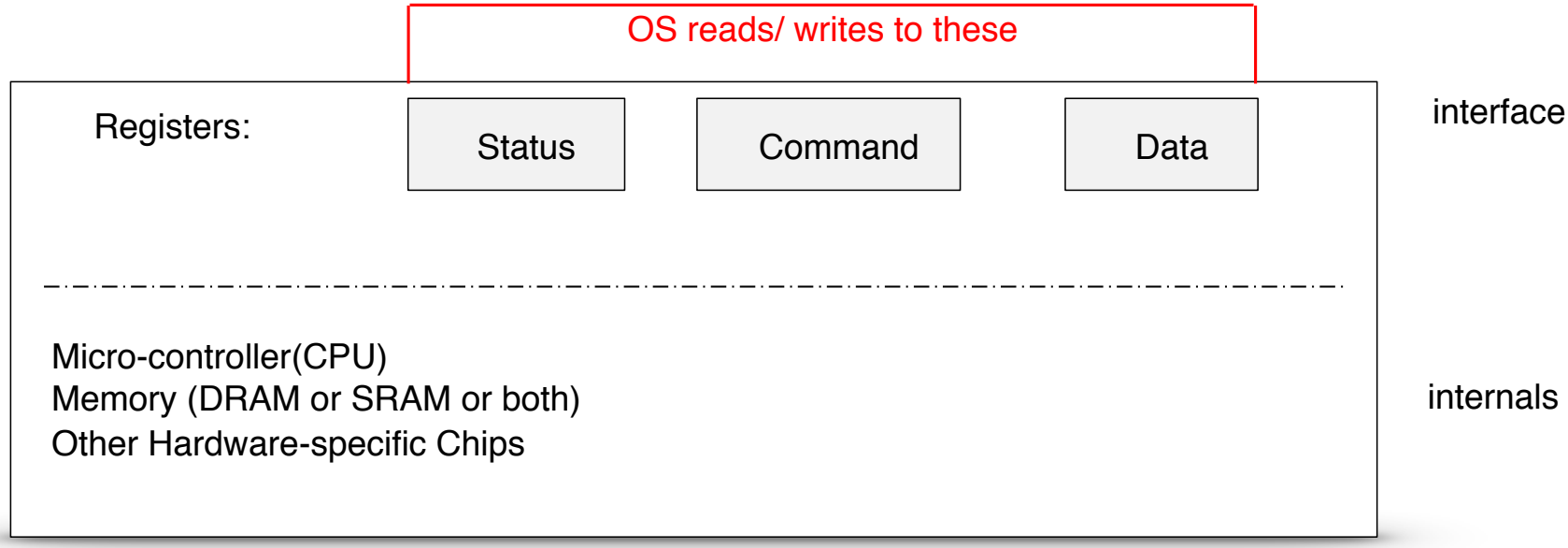
- **Buses**
  - **Data paths** that provided to enable information between CPU(s), RAM, and I/O devices.
- **I/O bus**
  - Data path that **connects** a **CPU** to an **I/O** device.
  - I/O bus is connected to I/O device by three hardware components:
    - I/O ports,
    - interfaces and
    - device controllers.

36. I/O Devices

1. System Architecture
2. **Devices**
3. Case Study

Canonical Device

- Canonical Devices has two important components.
  - **Hardware interface** allows the system software to control its operation.
  - **Internals** which is implementation specific.



Canonical Device

Hardware interface
of 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

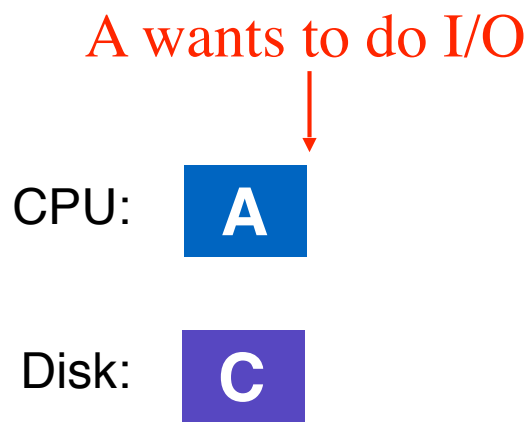
By reading and writing above **three registers**,

the operating system can **control device behavior**.

Hardware interface
of Canonical Device

Typical interaction example

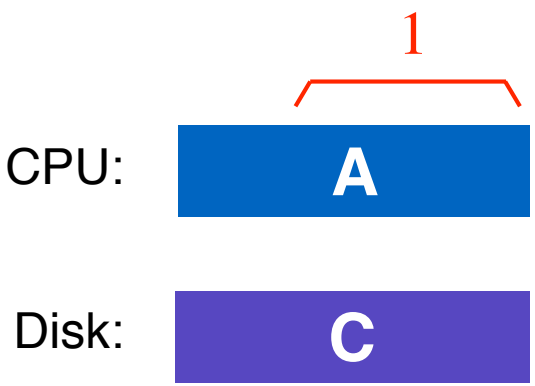
```
while ( STATUS == BUSY)
    ; //wait until device is not busy
write data to data register
write command to command register
    Doing so starts the device and executes the command
while ( STATUS == BUSY)
    ; //wait until device is done with your request
```



Hardware interface
of Canonical Device

Typical interaction example

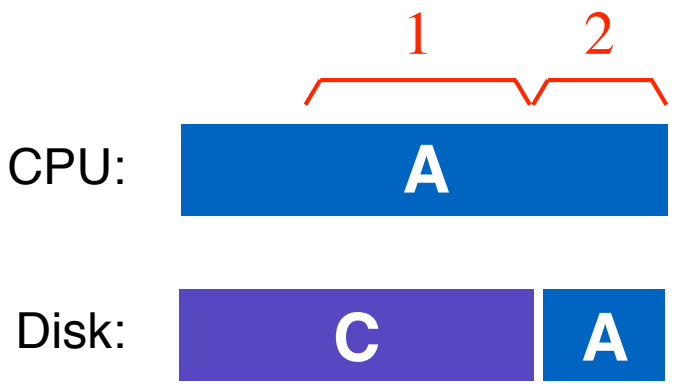
```
while ( STATUS == BUSY)                                    (1)
    ; //wait until device is not busy
write data to data register
write command to command register
    Doing so starts the device and executes the command
while ( STATUS == BUSY)
    ; //wait until device is done with your request
```



Hardware interface
of Canonical Device

Typical interaction example

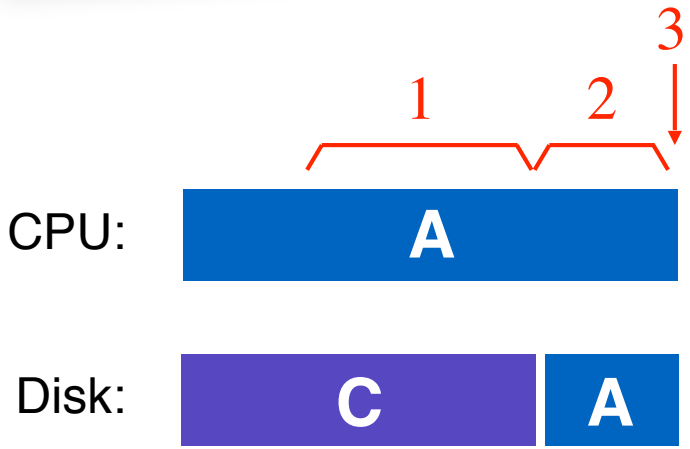
```
while ( STATUS == BUSY)
    ; //wait until device is not busy
write data to data register                    (2)
write command to command register
    Doing so starts the device and executes the command
while ( STATUS == BUSY)
    ; //wait until device is done with your request
```



Hardware interface
of Canonical Device

Typical interaction example

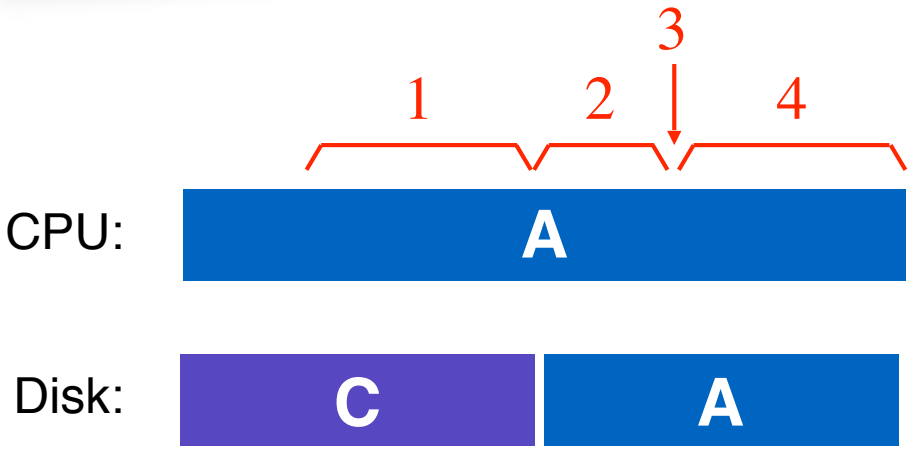
```
while ( STATUS == BUSY)
    ; //wait until device is not busy
write data to data register
write command to command register                    (3)
    Doing so starts the device and executes the command
while ( STATUS == BUSY)
    ; //wait until device is done with your request
```



Hardware interface
of Canonical Device

Typical interaction example

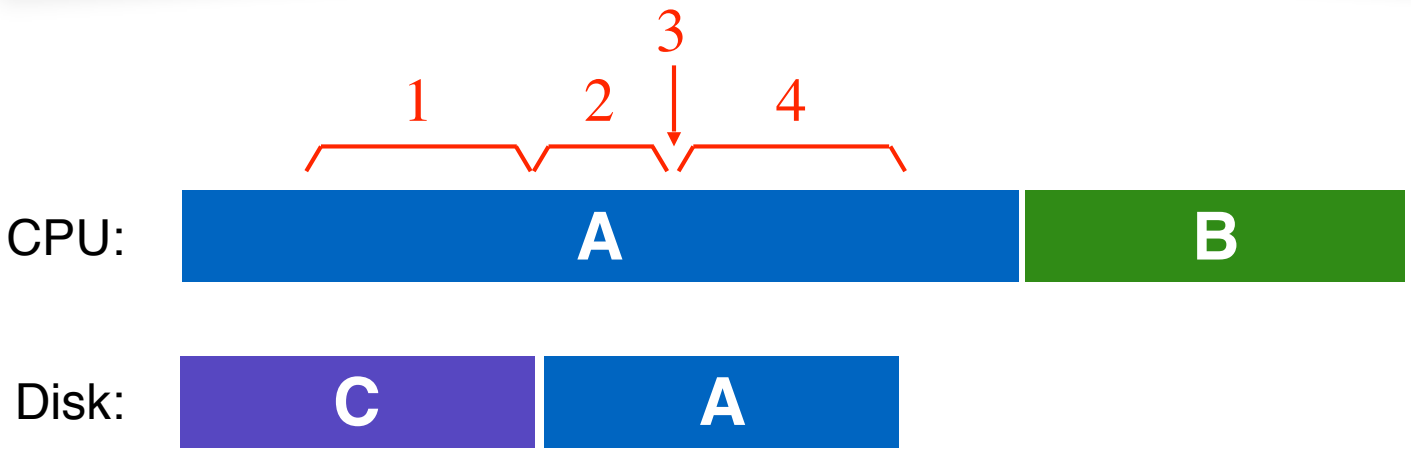
```
while ( STATUS == BUSY)
    ; //wait until device is not busy
write data to data register
write command to command register
    Doing so starts the device and executes the command
while ( STATUS == BUSY)                                    (4)
    ; //wait until device is done with your request
```



Hardware interface
of Canonical Device

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

- 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.

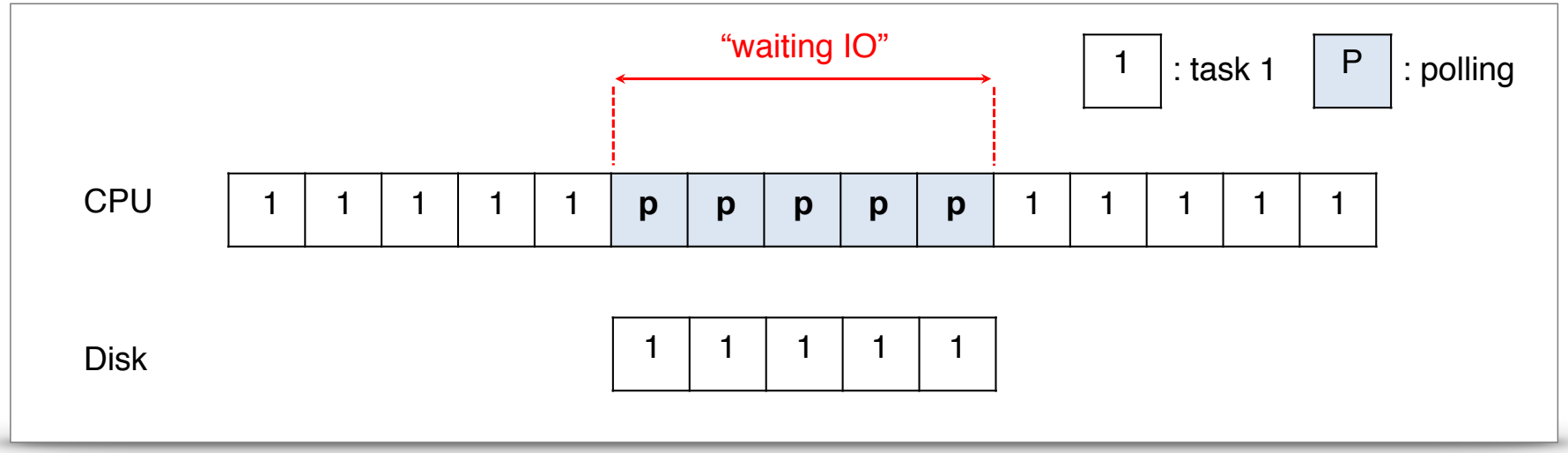


Diagram of CPU utilization by polling

Interrupts

- 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.

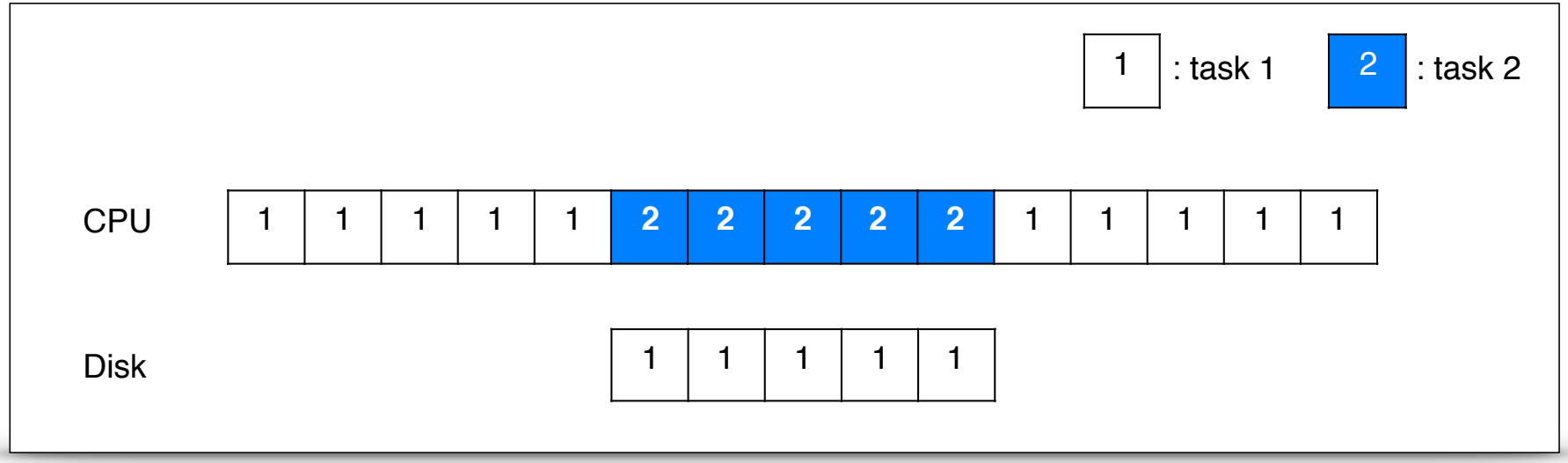


Diagram of CPU utilization by interrupt

Polling vs interrupts

- However, "**interrupts is not always the best solution**"
  - If, device performs very quickly, interrupt will "slow down" the system.
  - Because **context switch is expensive** (switching to another process)

If a device is fast ➔ **poll** is best.
If it is slow ➔ **interrupts** is better.

CPU is once again over-burdened

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

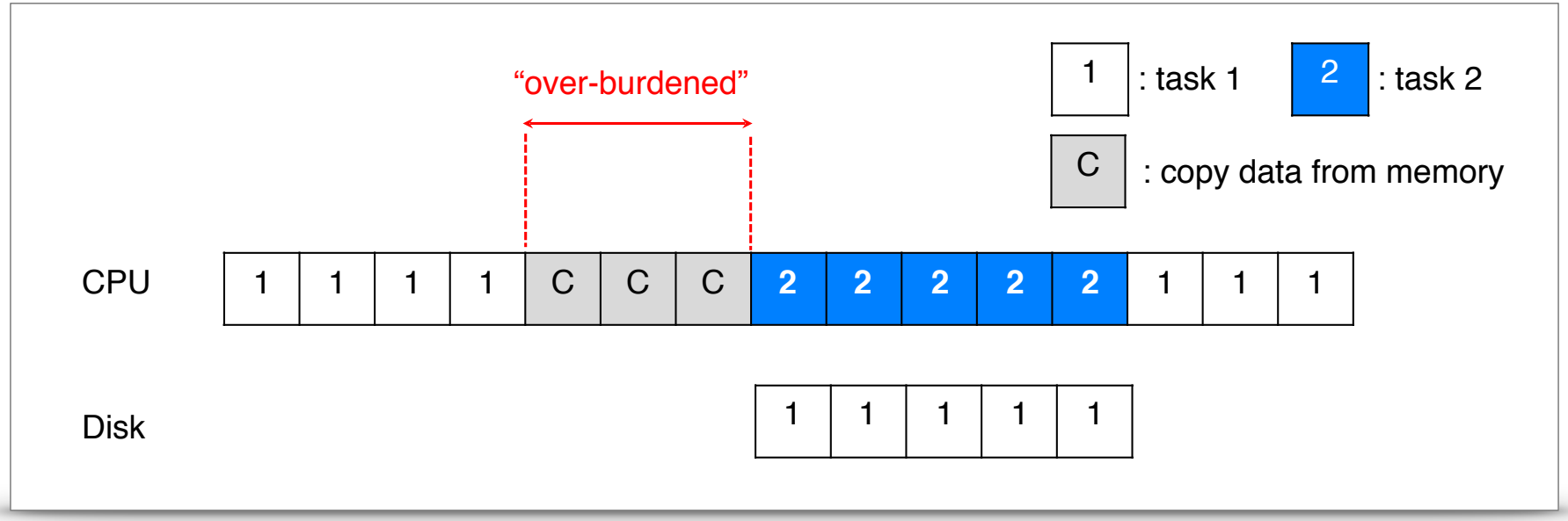


Diagram of CPU utilization

DMA (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.

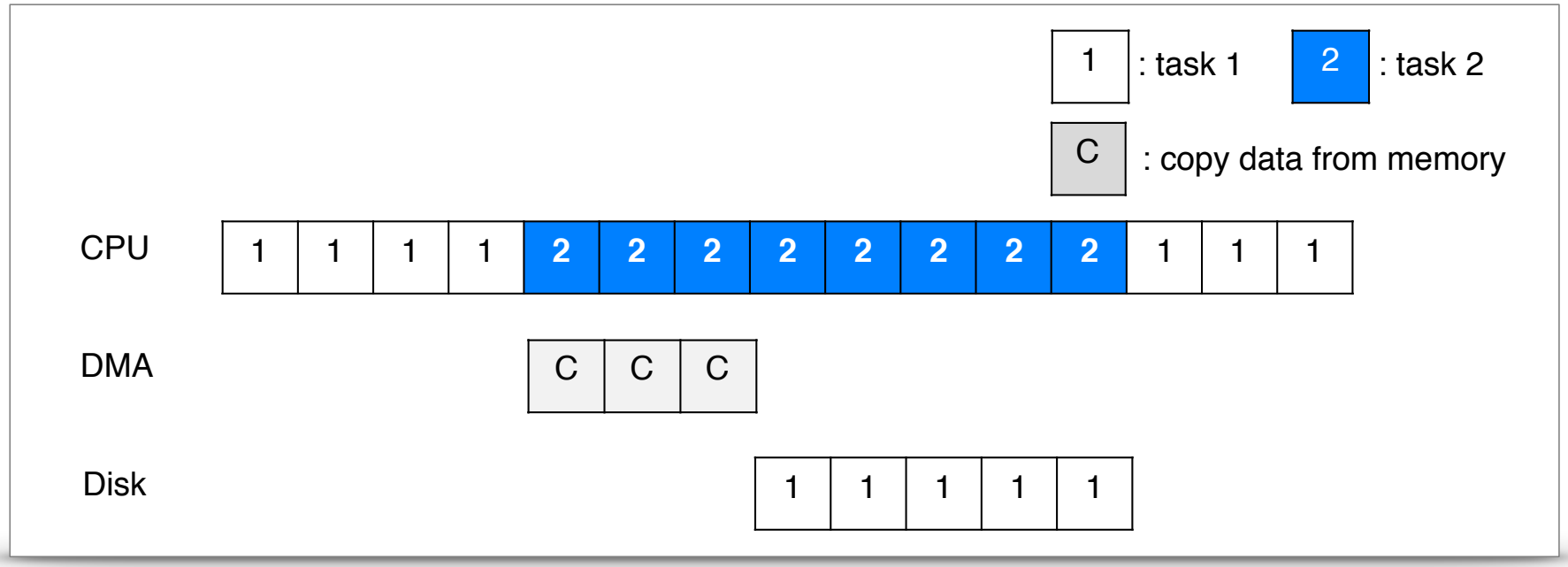
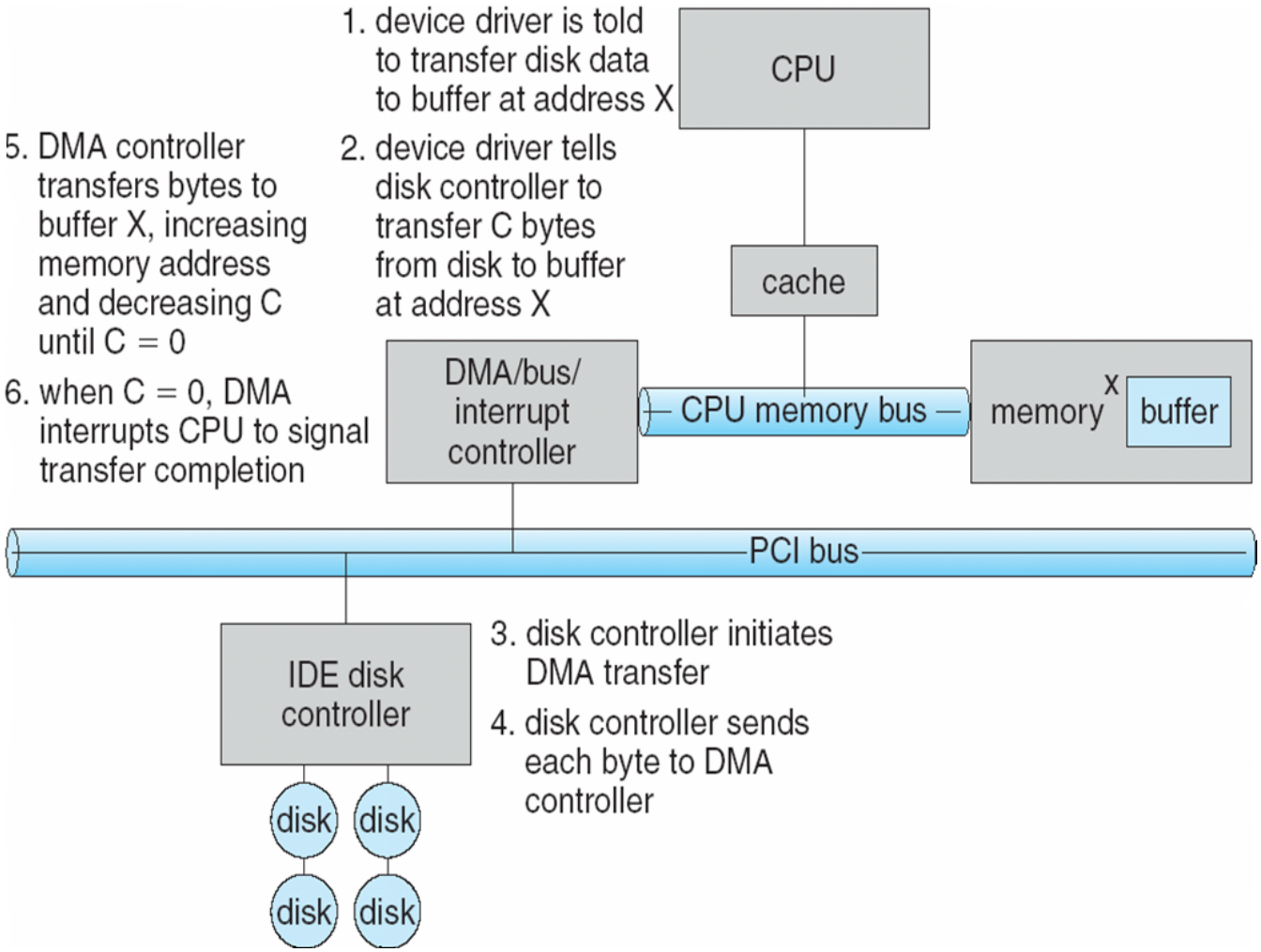


Diagram of CPU utilization by DMA

DMA Handshake



Example: OS Code

Polling

```
copy_from_user(buffer, p, count);              /* p is the kernel bufer */
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( );
```

Interrupt

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY)
*printer_data_register = p[0];
scheduler( );
```

ISR

```
if (count == 0) {
      unblock_user( );
} else {
      *printer_data_register = p[i];
      count = count – 1;
       i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

DMA

```
copy_from_user(buffer, p, count)
set_up_DMA_controller( );
scheduler( );
```

ISR

```
acknowledge_interrupt( )
unblock_user( );
return_from_interrupt( );
```

Device interaction

- How the OS communicates with the **device**?
- Solutions
  - **I/O instructions**: a way for the OS to send data to specific device registers.
    - Ex) `in` and `out` instructions on x86
  - **memory-mapped I/O**
    - Device registers available as if they were memory locations.
    - The OS `load` (to read) or `store` (to write) to the device instead of main memory.
- Doesn't matter much (both are used)

Protocol Variants

- **Status** checks:
  - polling vs. interrupts
- **Data**:
  - PIO vs. DMA
- **Control**:
  - special instructions vs. memory-mapped I/O

Device interaction (Cont.)

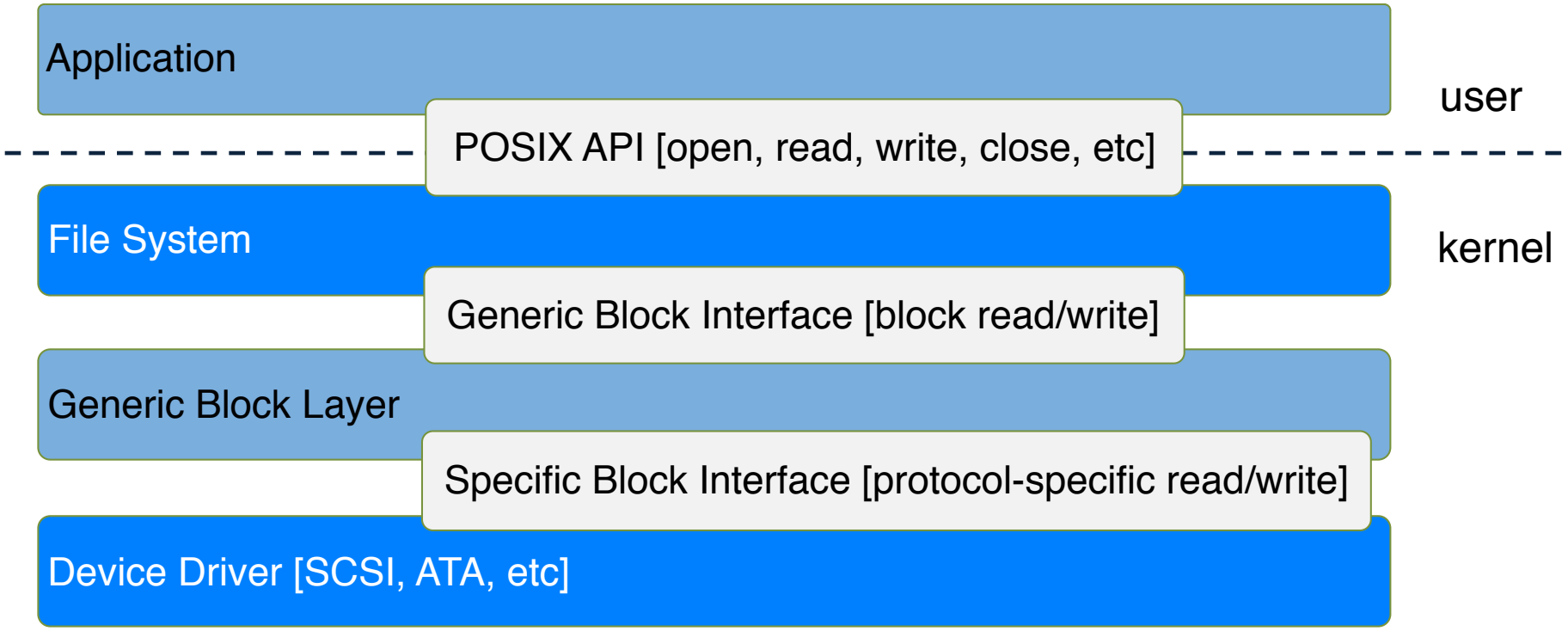
- How the OS interact with **different specific interfaces**?
  - Ex) We'd like to build a file system that worked on top of SCSI disks, IDE disks, USB keychain drivers, and so on.
- Solutions: **Abstraction**
  - Abstraction encapsulate **any specifics of device interaction**.

Device Drivers

- **Block** device drivers
  - Manages devices that read and write data in blocks
    - Magnetic disc, CD-ROMS, etc.
- **Character** device drivers
  - Manages devices that require the kernel to send or receive data one byte at a time
    - terminals, modems, and printers.
- **Network** device drivers
  - Manages network interface cards and communication ports that connect to network devices such as bridges and routers

File system Abstraction

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

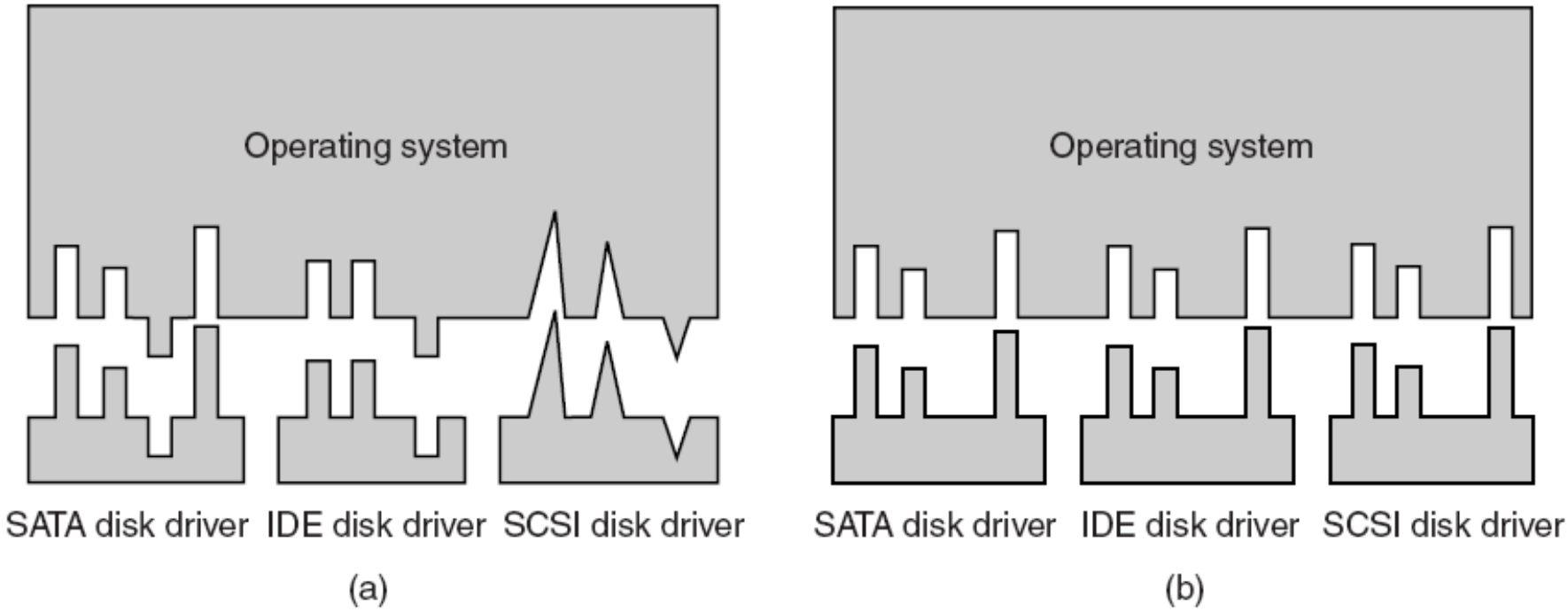


The File System Stack

Problem of File system Abstraction

- If there is a device having many special capabilities, these capabilities **will go unused** in the generic interface layer.
- **Over 70% of OS** code is found in device drivers.
  - Any device drivers are needed because you might plug it to your system.
  - They are primary contributor to **kernel crashes**, making **more bugs**.

Importance of Device Driver Interface



(a) Without a Standard Device Driver Interface

(b) With Standard Device Driver Interface

36. I/O Devices

1. System Architecture
2. Devices
3. **Case Study**

A Simple IDE Disk Driver

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

Control / Command Register

- **Control Register**:
  - Address `0×3F6` = `0×80 (0000 1RE0)`:
    R=reset, E=0 means "enable interrupt”
- **Command Block Registers**:
  - Address `0×1F0` = Data Port
  - Address `0×1F1` = Error
  - Address `0×1F2` = Sector Count
  - Address `0×1F3` = LBA low byte
  - Address `0×1F4` = LBA mid byte
  - Address `0×1F5` = LBA hi byte
  - Address `0×1F6` = 1B1D TOP4LBA: B=LBA, D=drive
  - Address `0×1F7` = Command/status

Status and Error Register

- **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

Basic Protocol

- **Wait for drive to be ready**. Read Status Register (0x1F7) until drive is not busy and READY.
- **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 IDE permits just two drives) to command registers (0x1F2-0x1F6).
- **Start the I/O**. by issuing read/write to command register. Write READ—WRITE command to command register (0x1F7).
- **Data transfer** (for writes): Wait until drive status is READY and DRQ (drive request for data); 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 transfer is complete.
- **Error handling**. After each operation, read the status register. If the ERROR bit is on, read the error register for details.

Thanks

Questions?