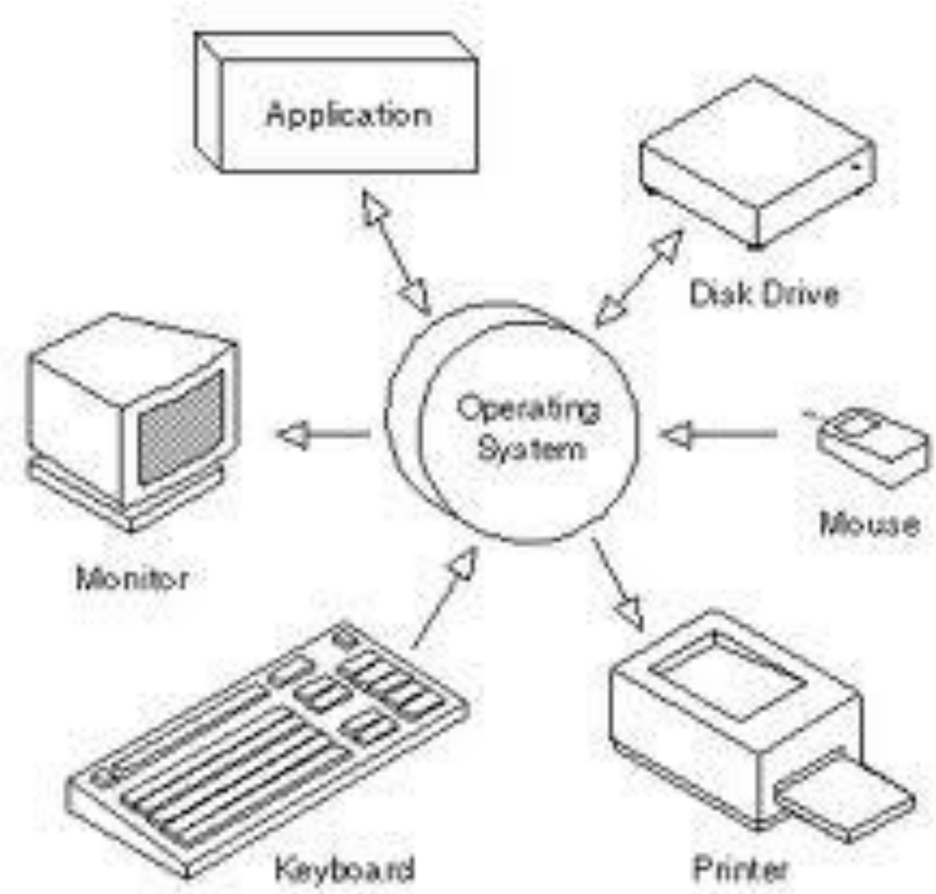
Operating Systems

I/O Part I

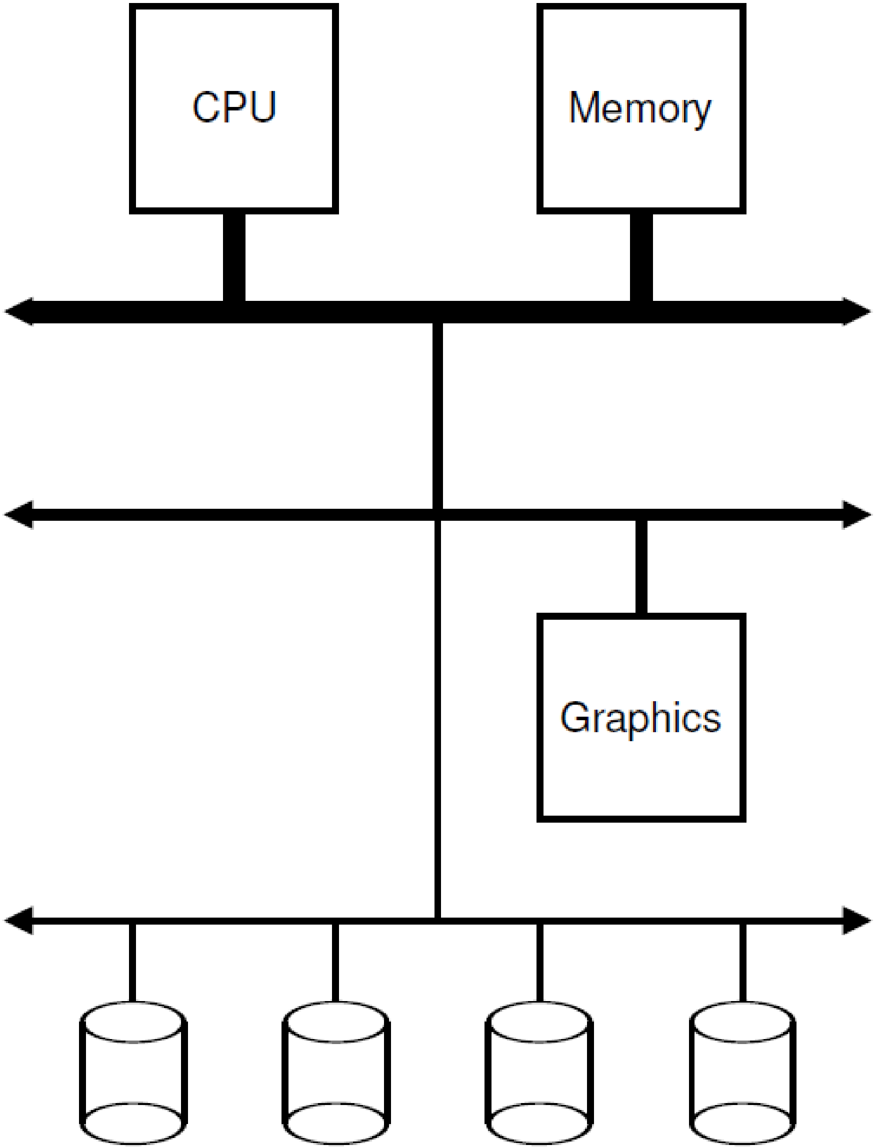
Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

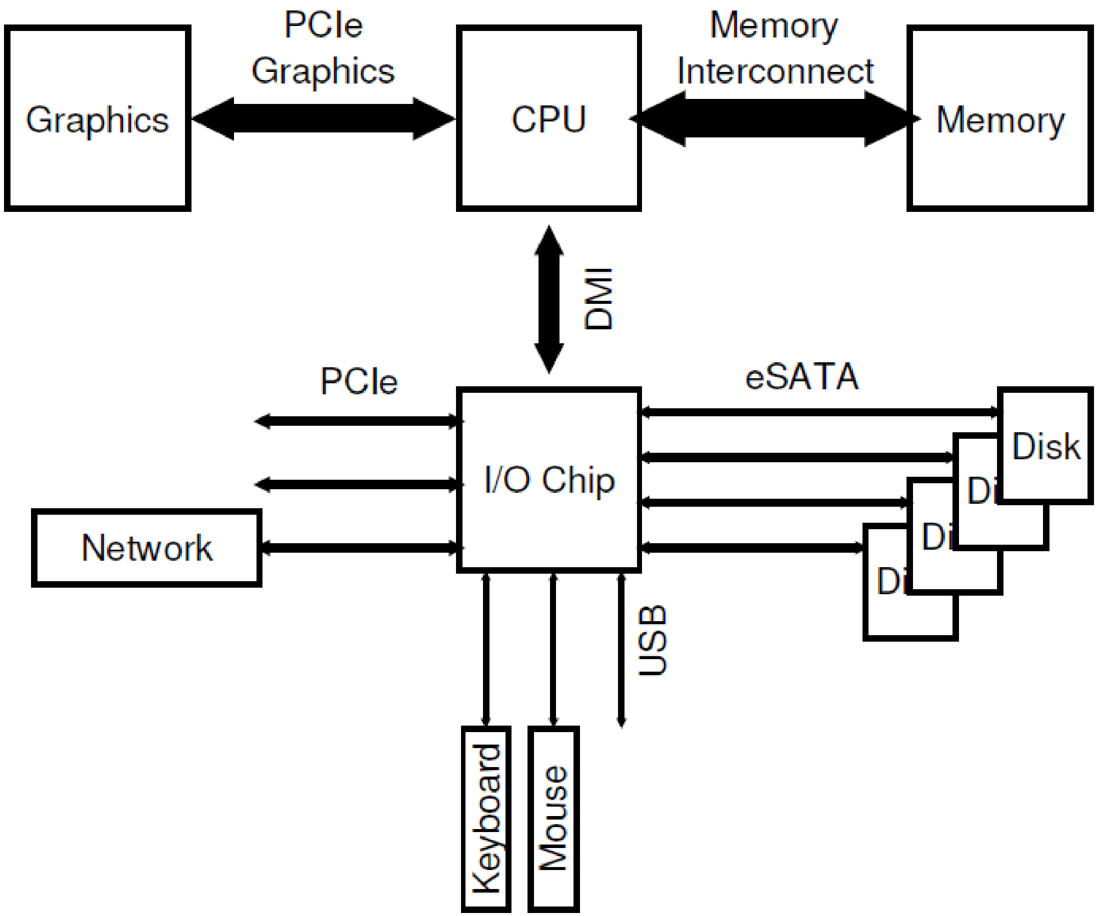
http://www.mzahran.com


Application
Disk Drive
Operating System
Monitor
Mouse
Keyboard
Printer

The faster a bus is,
the shorter it must be,
and it is expensive to design too.



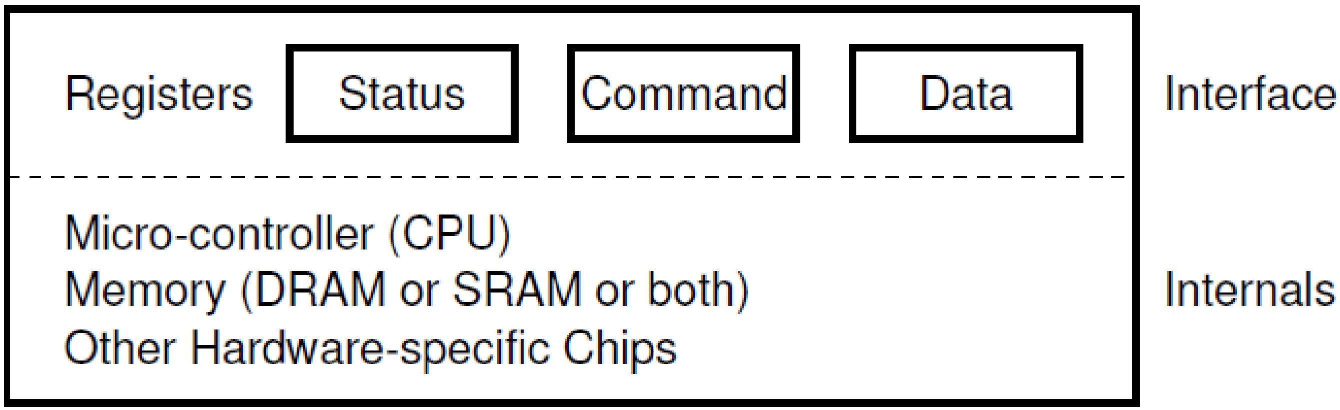
Modern System Architecture



Example of one of Intel Chipsets.

DMI: Intel's proprietary Direct Media Interface.

How Does a Generic I/O Device Look Like?



The OS can deal with the above device as follows:

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

Polling is not efficient.

Categories of I/O Devices

External devices that engage in I/O with computer systems can be grouped into three categories:

Human readable

- suitable for communicating with the computer user
- printers, terminals, video display, keyboard, mouse

Machine readable

- suitable for communicating with electronic equipment
- disk drives, USB keys, sensors, controllers

Communication

- suitable for communicating with remote devices
- ethernet cards, wifi adapters

A Simple Definition

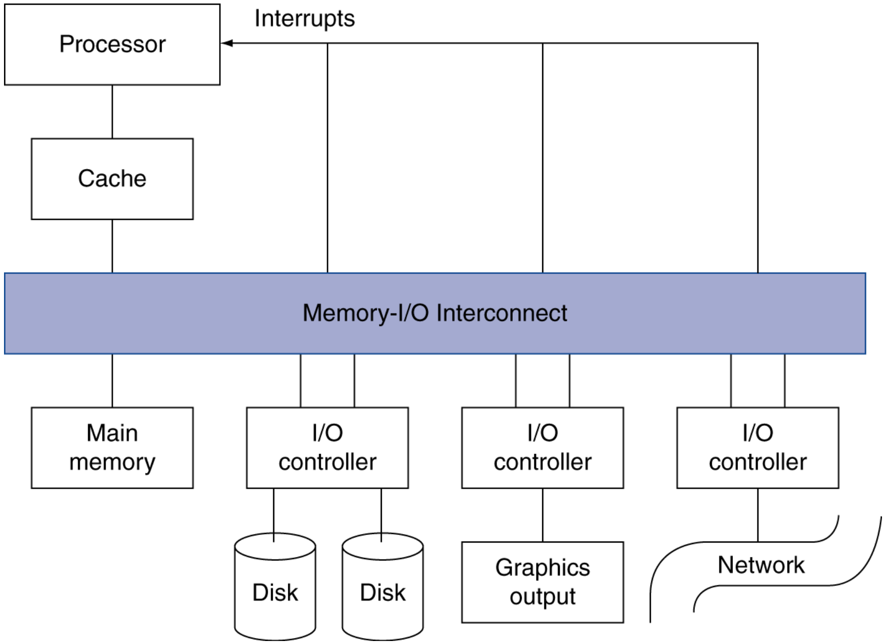
- The main concept of I/O is to move data from/to I/O devices to the processor using some modules and buffers.
- This is the way the processor deals with the outside world.

The OS and I/O

- The OS controls all I/O devices
  - Issue commands to devices
  - Catch interrupts
  - Handle errors
- Provides an interface between the devices and the rest of the system.

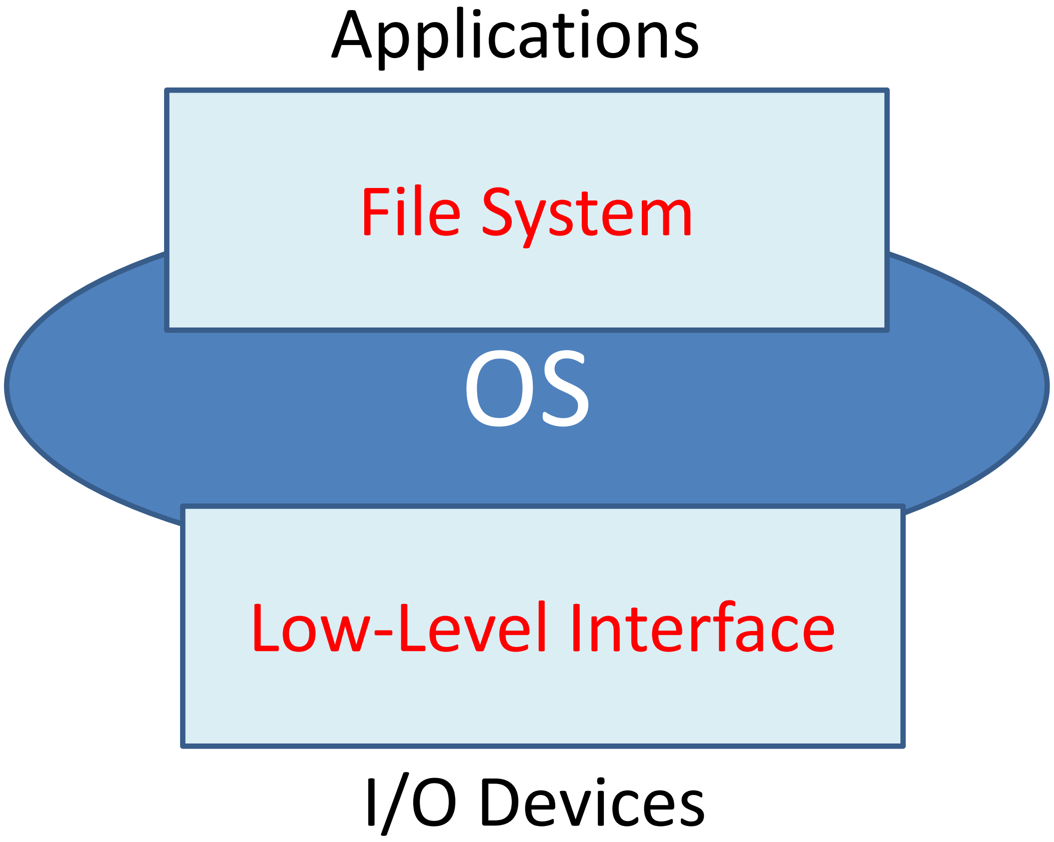
I/O Devices: Challenges

- Very diverse devices
  - behavior (i.e., input vs. output vs. storage)
  - partner (who is at the other end?)
  - data rate
- I/O Design affected by many factors (expandability, resilience)
- Performance:
  - access latency
  - throughput
  - connection between devices and the system
  - the memory hierarchy
  - the operating system
- A variety of different users



I/O Devices

- Block device
  - Stores information in fixed-size blocks
  - Each block has its own address
  - Transfers in one or more blocks
  - Example: Hard-disks, USB sticks
- Character device
  - Delivers or accepts stream of character
  - Is not addressable
  - Example: mice, printers, network interfaces


Applications
File System
OS
Low-Level Interface
I/O Devices

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner	400 KB/sec
Digital camcorder	3.5 MB/sec
802.11g Wireless	6.75 MB/sec
52x CD-ROM	7.8 MB/sec
Fast Ethernet	12.5 MB/sec
Compact flash card	40 MB/sec
FireWire (IEEE 1394)	50 MB/sec
USB 2.0	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
SATA disk drive	300 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec

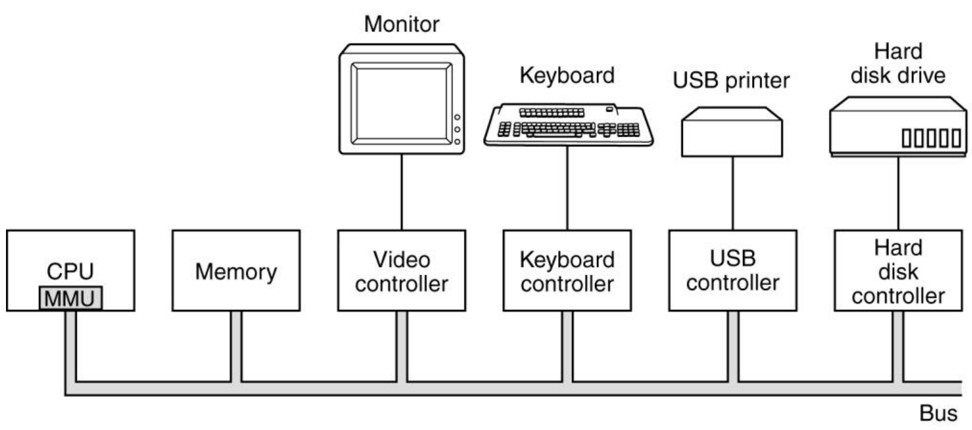
I/O Units

Mechanical component

The Device Itself

Electronic Component

Device Controller



Controller and Device

- Each controller has few registers used to communicate with CPU
- By writing/reading into/from those registers, the OS can control the devices.
- There are also data buffers in the device that can be read/written by the OS.

How does the OS deal with controllers and devices?

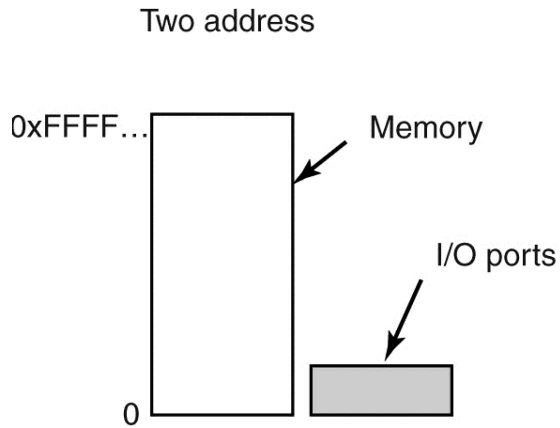
How does CPU communicate with control registers and data buffers?

Two main approaches

- I/O port space
- Memory-mapped I/O

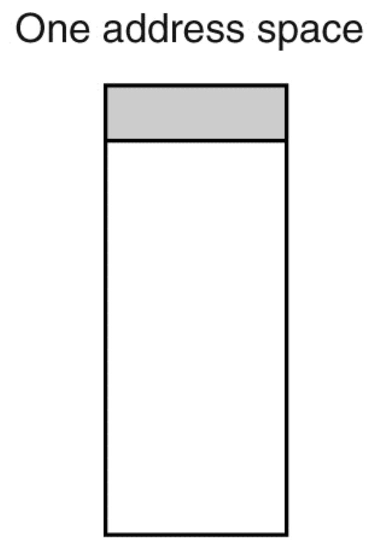
I/O Port Space

- Each control register is assigned an I/O port number
- The set of all I/O ports form the I/O port space
- I/O port space is protected



Memory-Mapped I/O

- Map control registers into the memory space
- Each control register is assigned a unique memory address

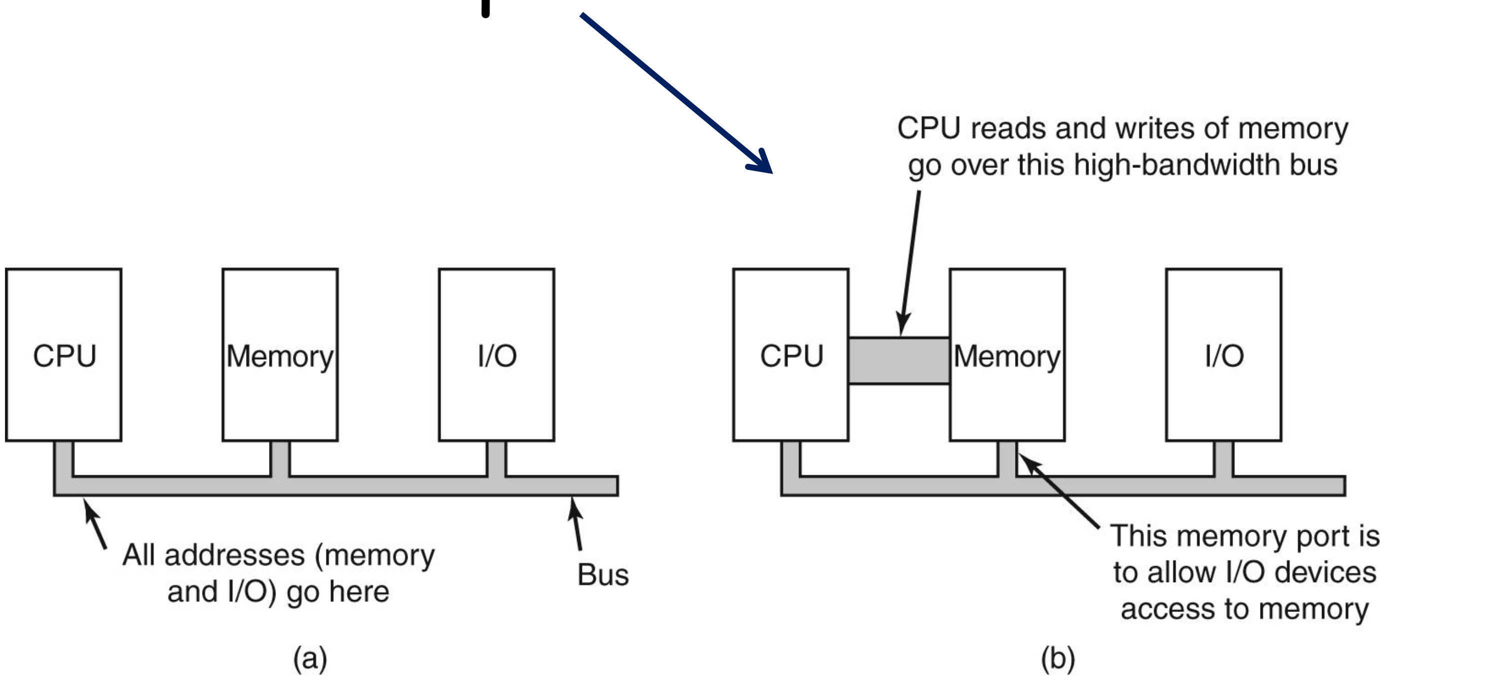


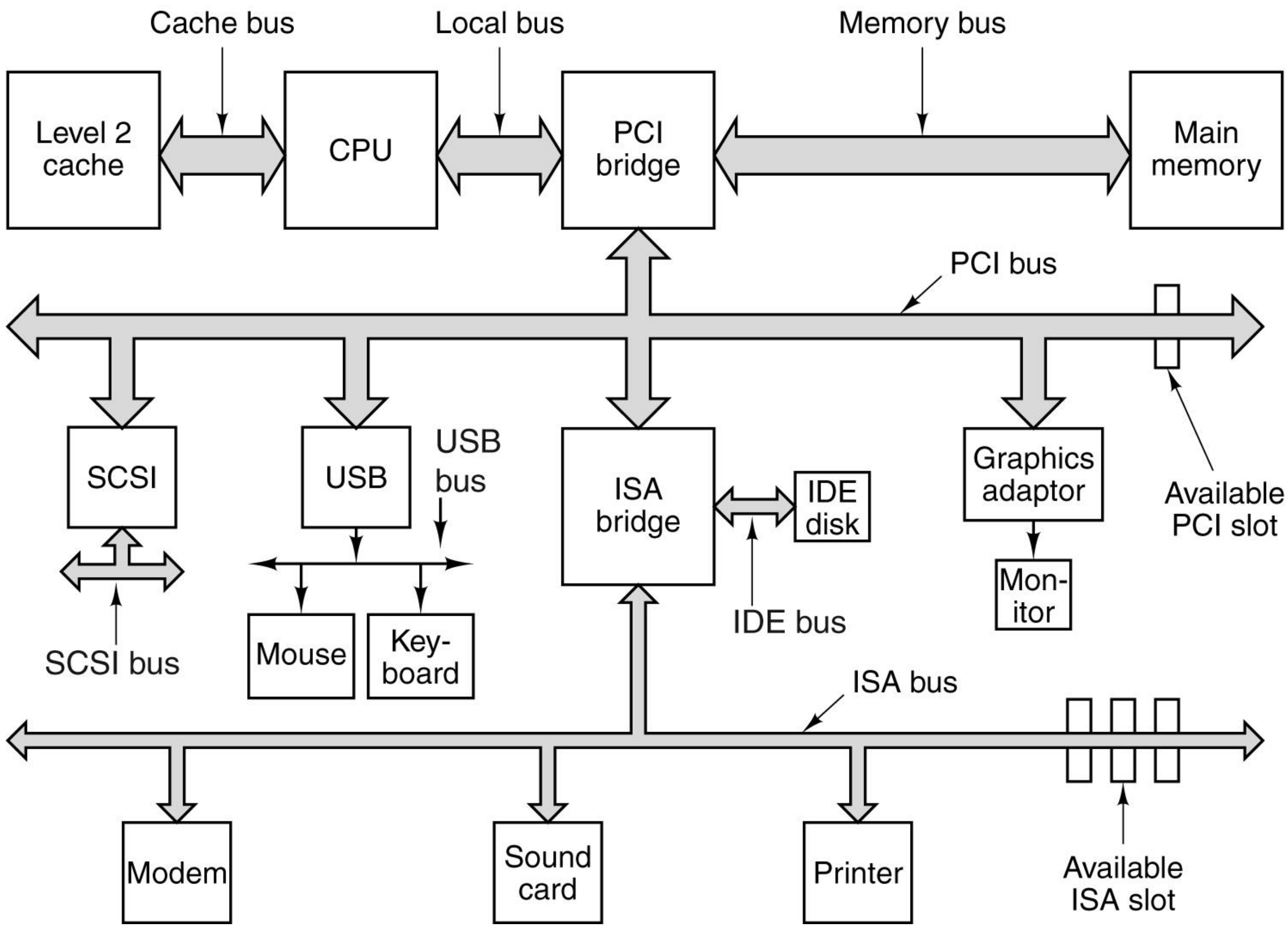
Advantages of Memory-Mapped I/O

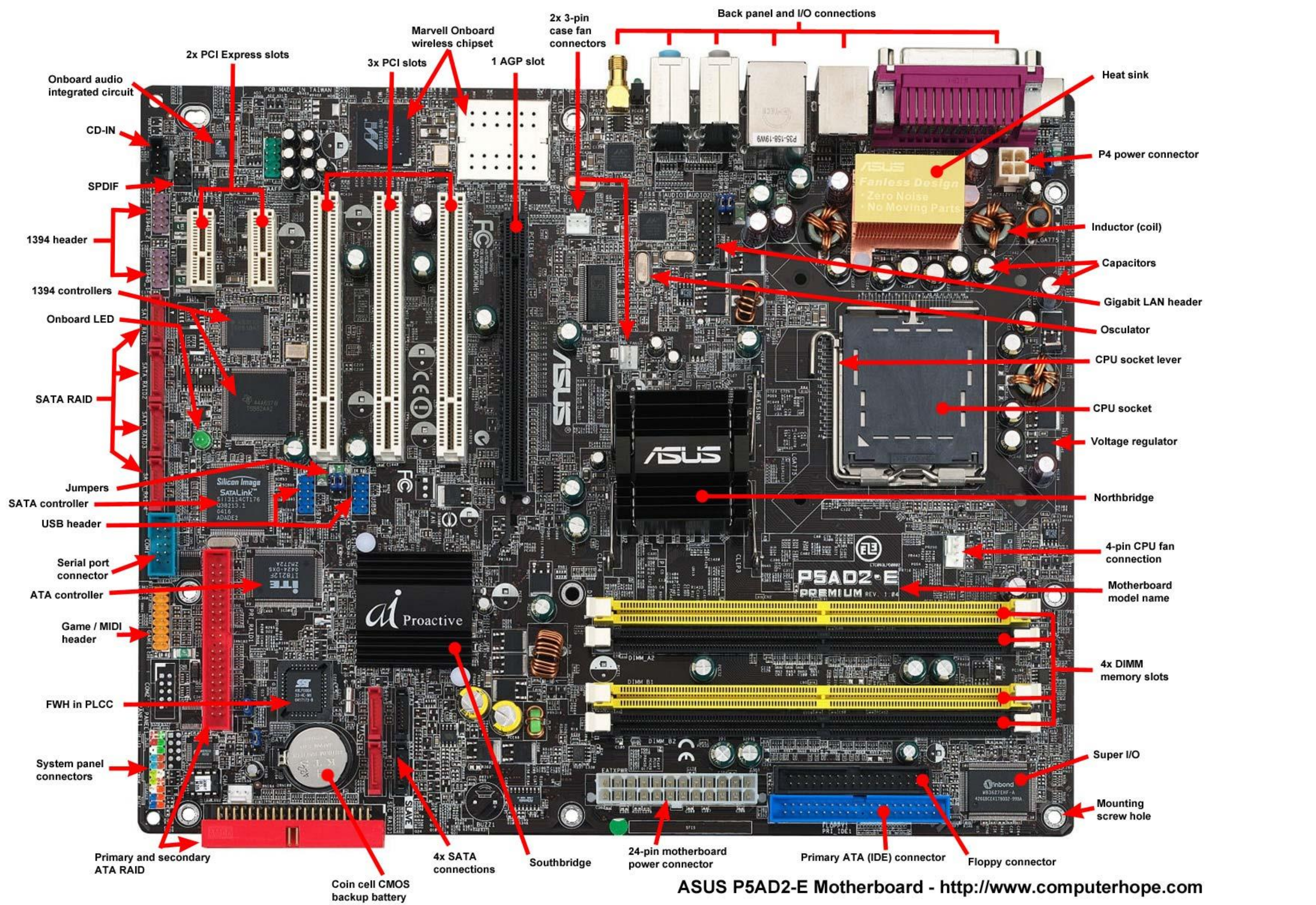
- Device drivers can be written entirely in C (since no special instructions are needed)
- No special protection is needed from OS, just refrain from putting that portion of the address space in any user's virtual address space.
- Every instruction that can reference memory can also reference control registers.

Disadvantages of Memory-Mapped I/O

- Caching a device control register can be disastrous.
- Hardware complications




Cache bus
Local bus
Memory bus
Level 2 cache
CPU
PCI bridge
Main memory
PCI bus
SCSI
USB
USB bus
ISA bridge
IDE disk
Graphics adaptor
Available PCI slot
Monitor
SCSI bus
Mouse
Keyboard
IDE bus
ISA bus
Modem
Sound card
Printer
Available ISA slot

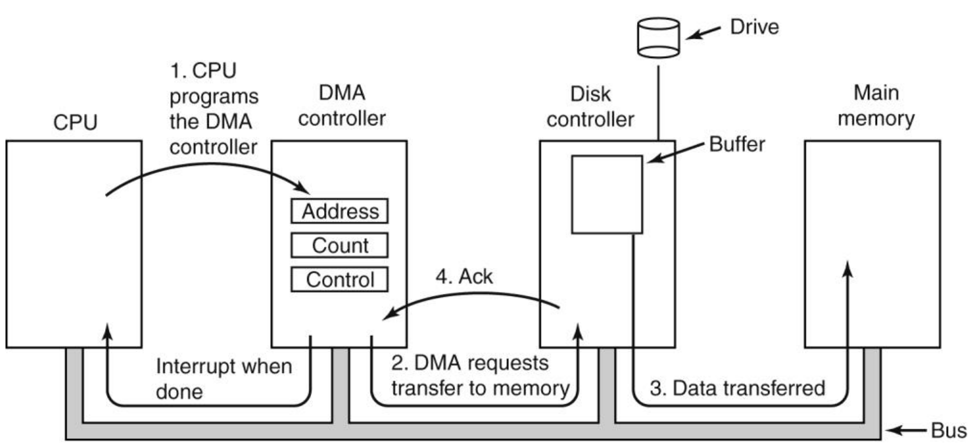


Source: https://www.computerhope.com/cdn/bigmb.jpg

Interrupts

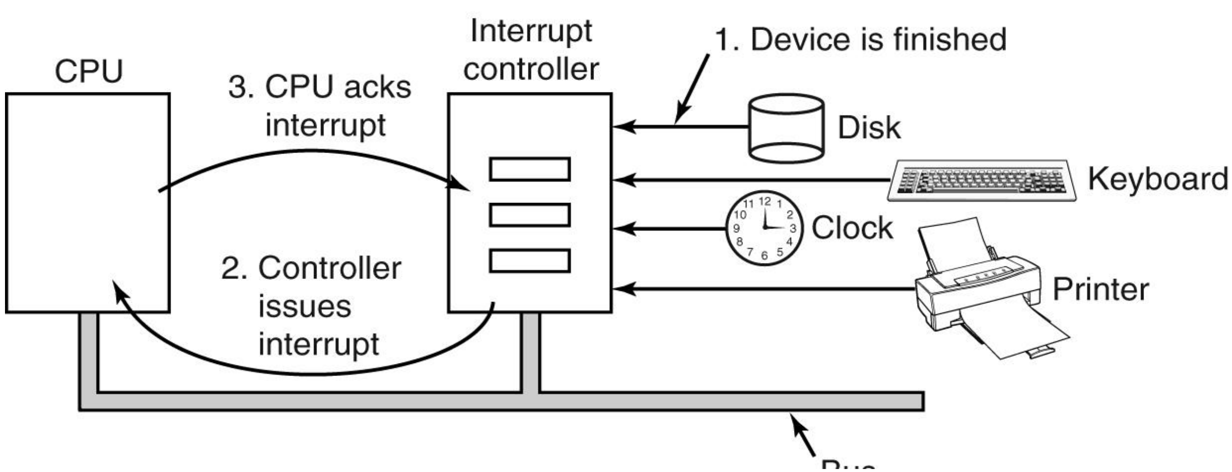
Direct Memory Access (DMA)

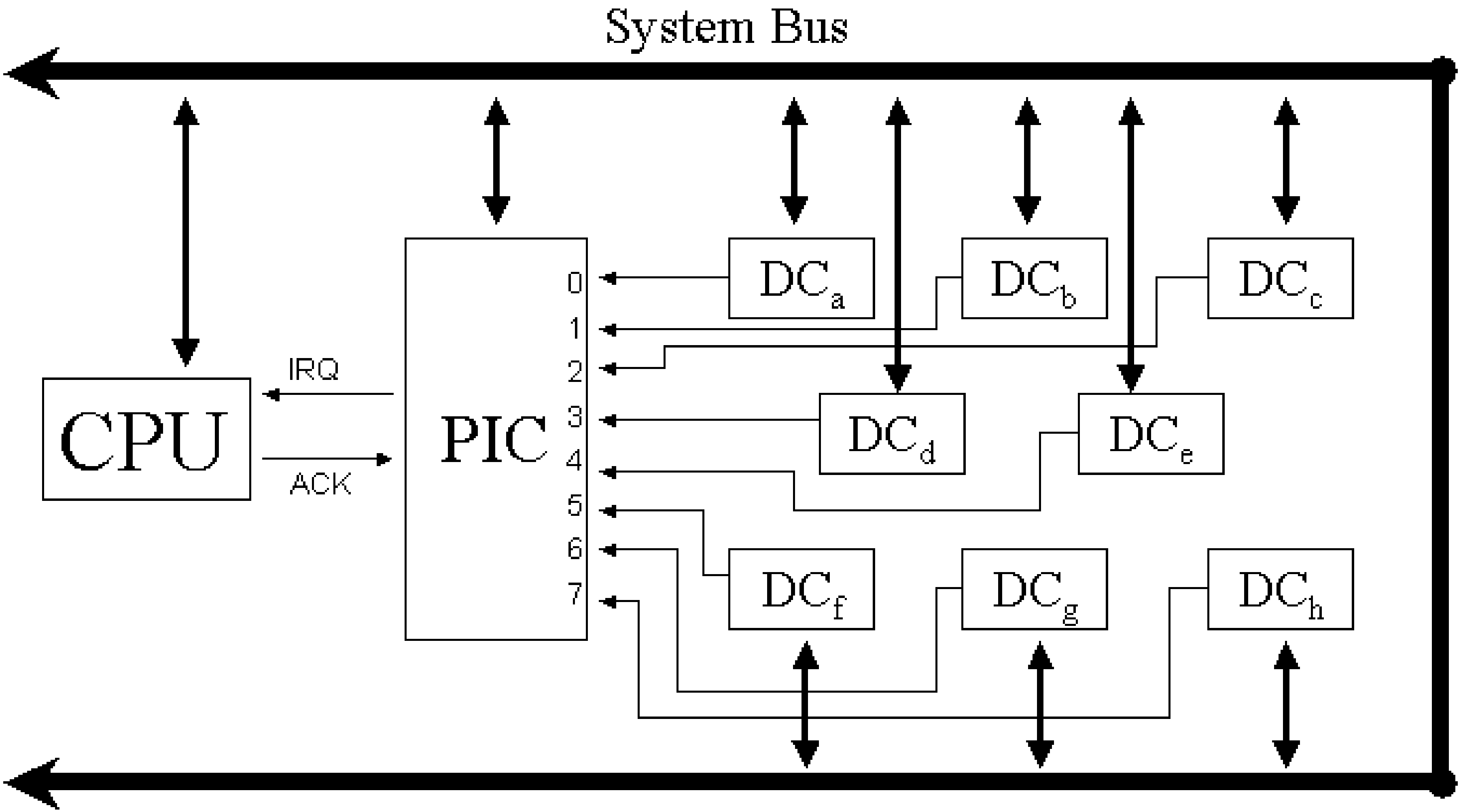
- It is not efficient for the CPU to request data from I/O one byte at a time.
- DMA controller has access to the system bus independent from the CPU



Interrupts

- When an I/O device has finished the work given to it, it causes an interrupt.
- The interrupt makes the OS stop the current process, switch to kernel mode, and start handling that interrupt.



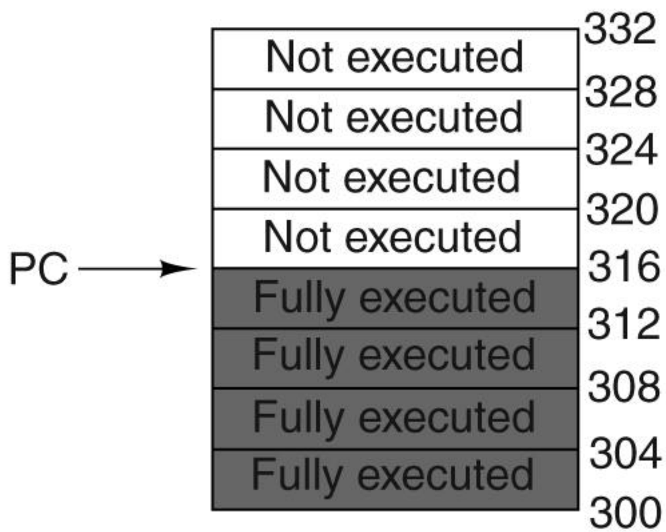


DC = Device Controller

PIC = Programmable Interrupt Controller

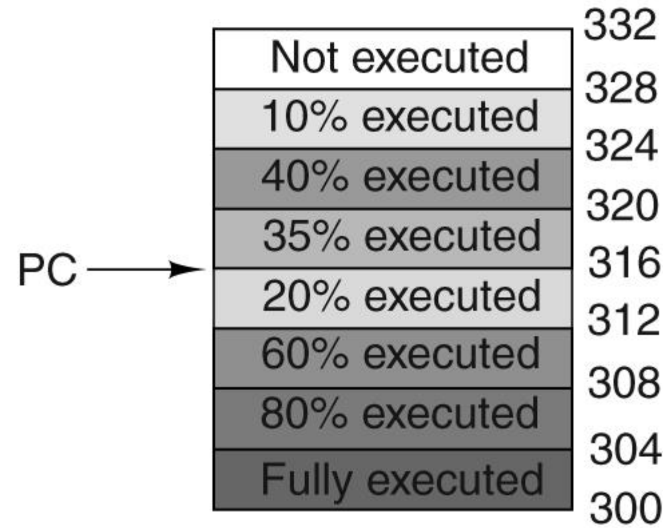
Precise Interrupts

- Makes handling interrupts much simpler.
- Has 4 properties
  - The program counter (PC) is saved in known place
  - All instructions before the one pointed by PC have fully executed
  - No instruction beyond the one pointed by PC has been executed
  - The execution state of the instruction pointed to by the PC is known



(a)

Precise Interrupt



(b)

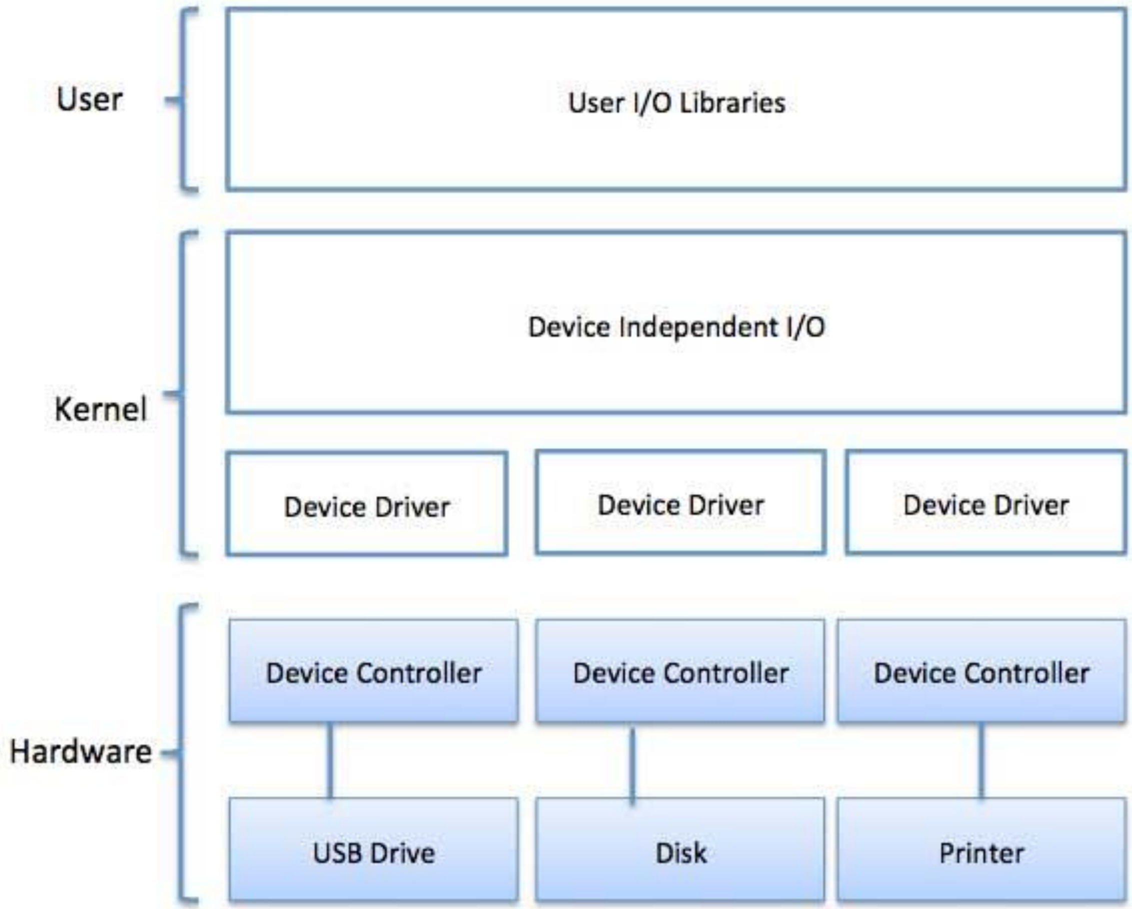
Imprecise Interrupt

Important!

There are other scenarios where interrupts happen as we will see in interrupt-driven I/O.

The Software Part

The Big Picture



Source: https://www.tutorialspoint.com/operating_system/images/io_software.jpg

I/O Software

- Device independence:
  - We should be able to write programs that can access any I/O
- device without having to specify the device in advance.
- Error handling
  - Should be handled as close to the hardware as possible
- Synchronous (blocking) vs asynchronous (interrupt-driven)
- Buffering

Do not confuse I/O software with device driver.

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

- Uniform interfacing for device drivers
  - Trying to make all devices look the same
  - For each class of devices, the OS defines a set of functions that the driver must supply.
  - This layer of OS maps symbolic device names onto proper drivers

Three Ways for Doing I/O

- Programmed I/O
- Interrupt-driven I/O
- I/O Using DMA

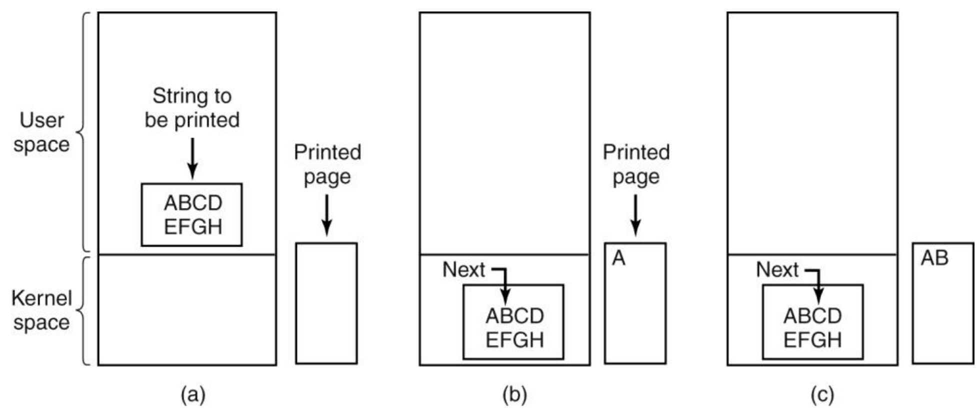
	No Interrupts	Use of Interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

Programmed I/O

- CPU does all the work
- Busy-waiting (polling)

Example:

```
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( );
```

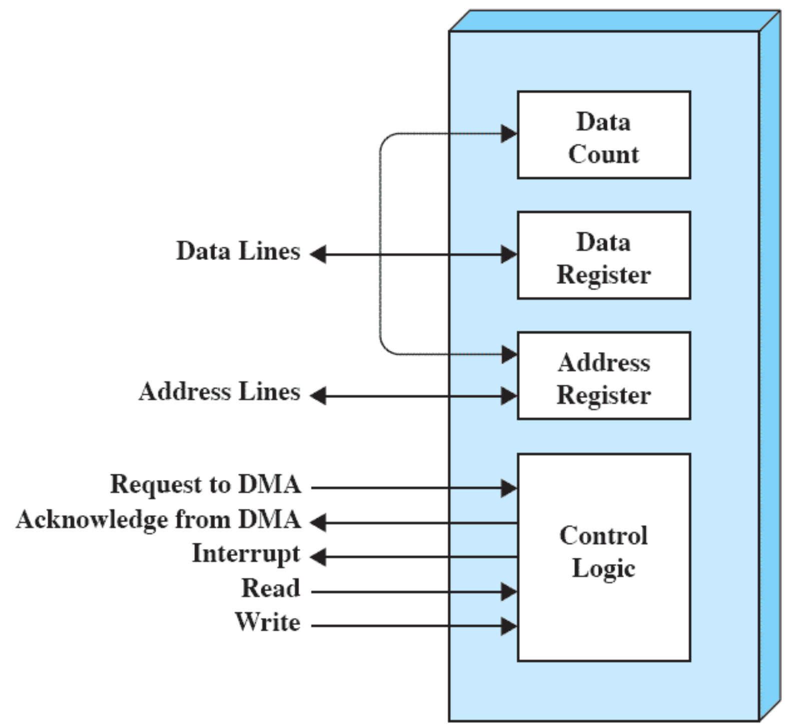


Interrupt-Driven I/O

- Waiting for a device to be ready, the process is blocked and another process is scheduled.
- When the device is ready it raises an interrupt.

I/O Using DMA

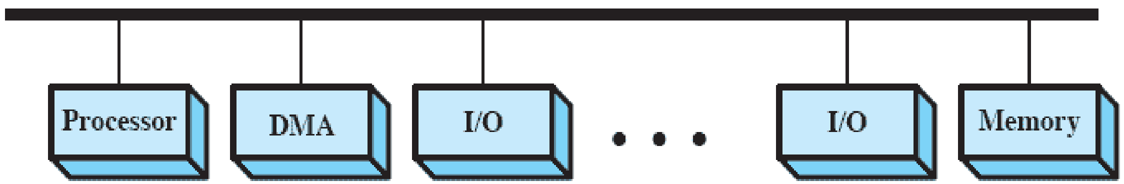
- DMA does the work instead of the CPU
- Let the DMA do its work and then interrupts



General Design of DMA

DMA Alternative Configurations

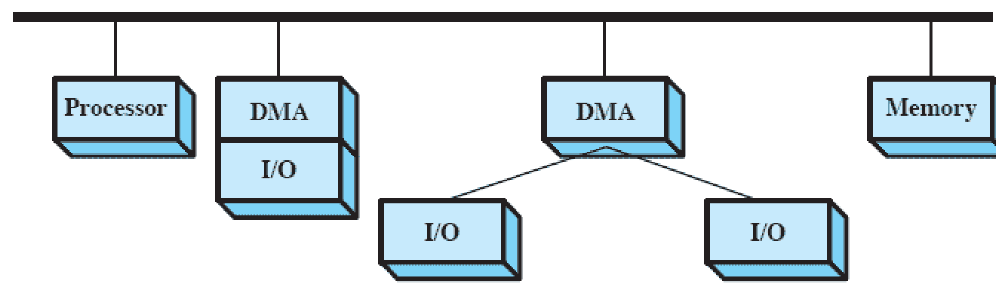
Inefficient



(a) Single-bus, detached DMA

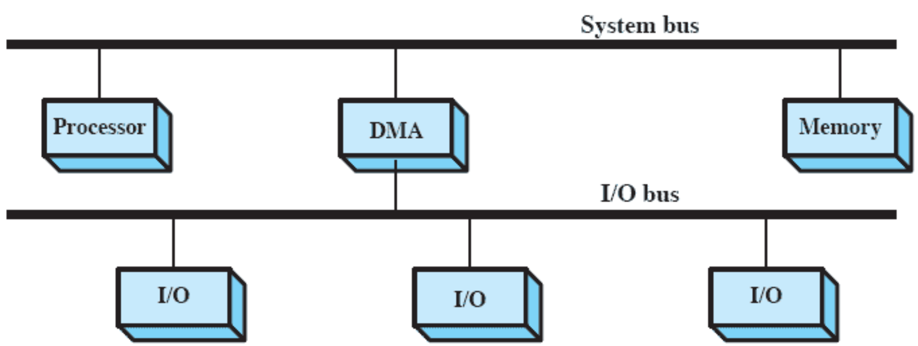
I/O Module

Better



(b) Single-bus, Integrated DMA-I/O

Best

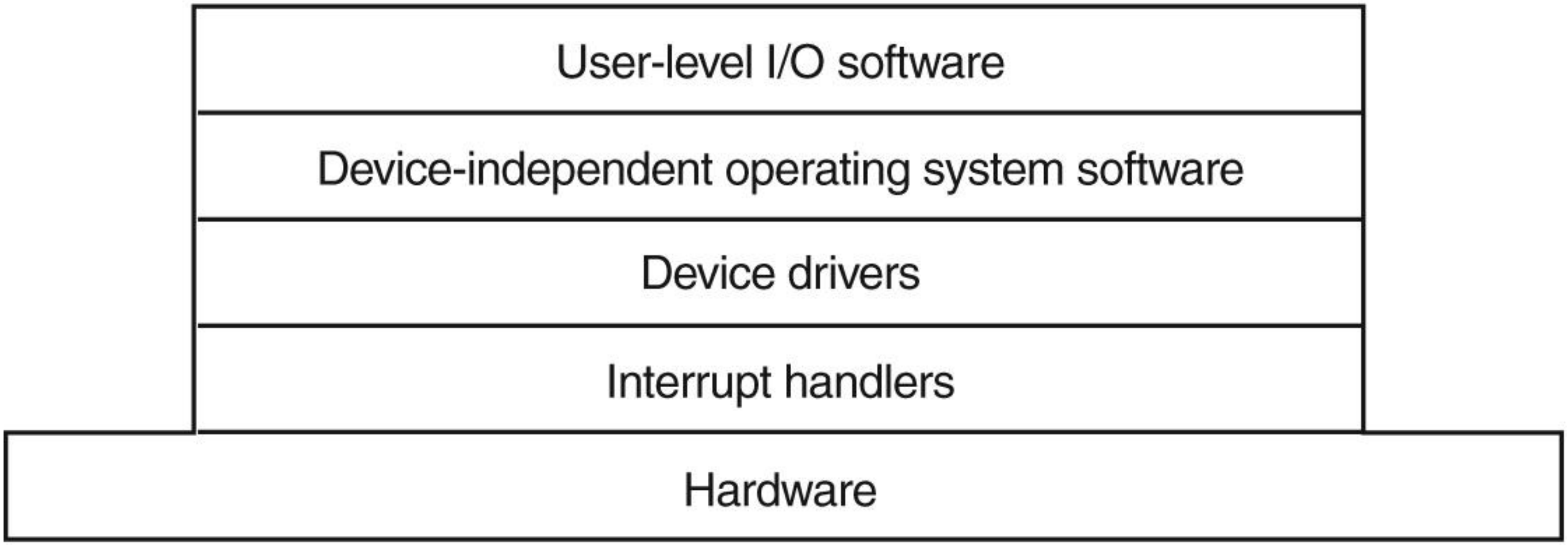


(c) I/O bus

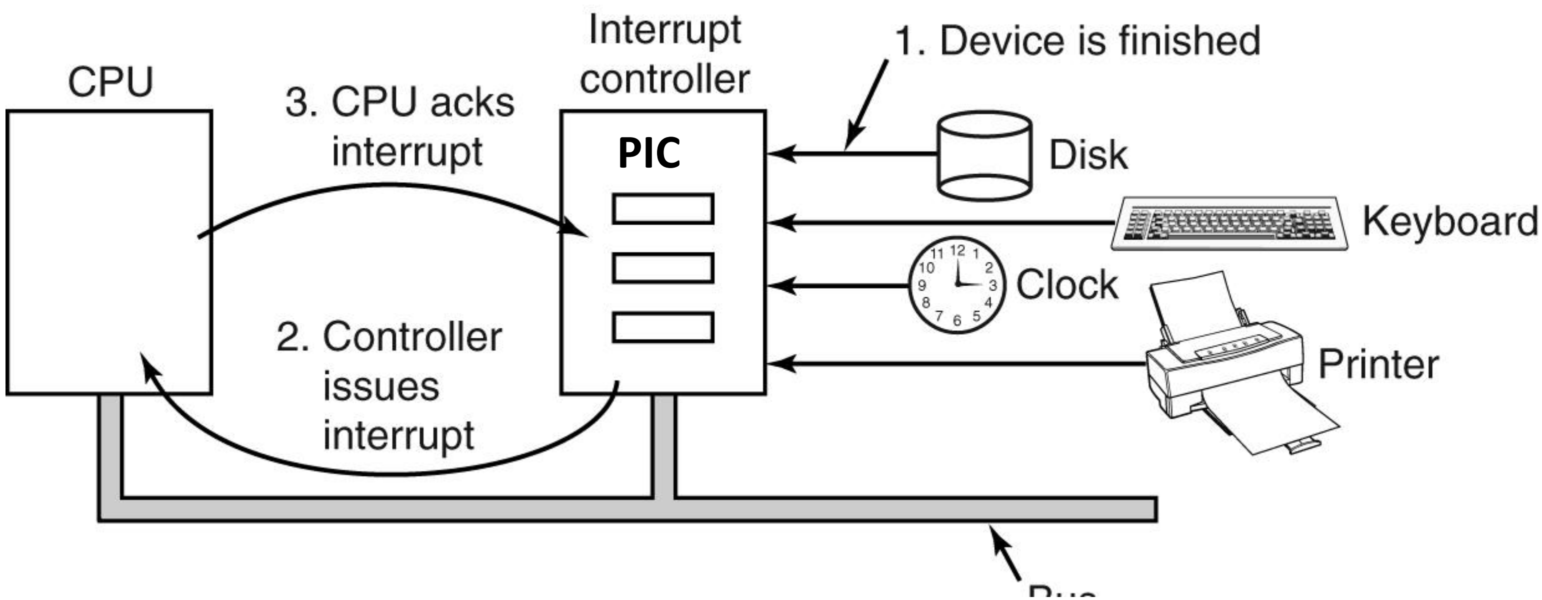
What is and I/O Module?

- A hardware module.
  - Can be device controller or a more advanced.
- Connected to the CPU (or DMA) from one side.
- Connected to the device and its device controller from the other side.
- Translates commands between processor (or DMA) and the device.
- Buffers data due to speed between two sides.

OS Software Layers for I/O



Interrupt Handlers



Interrupt Handlers in More Details

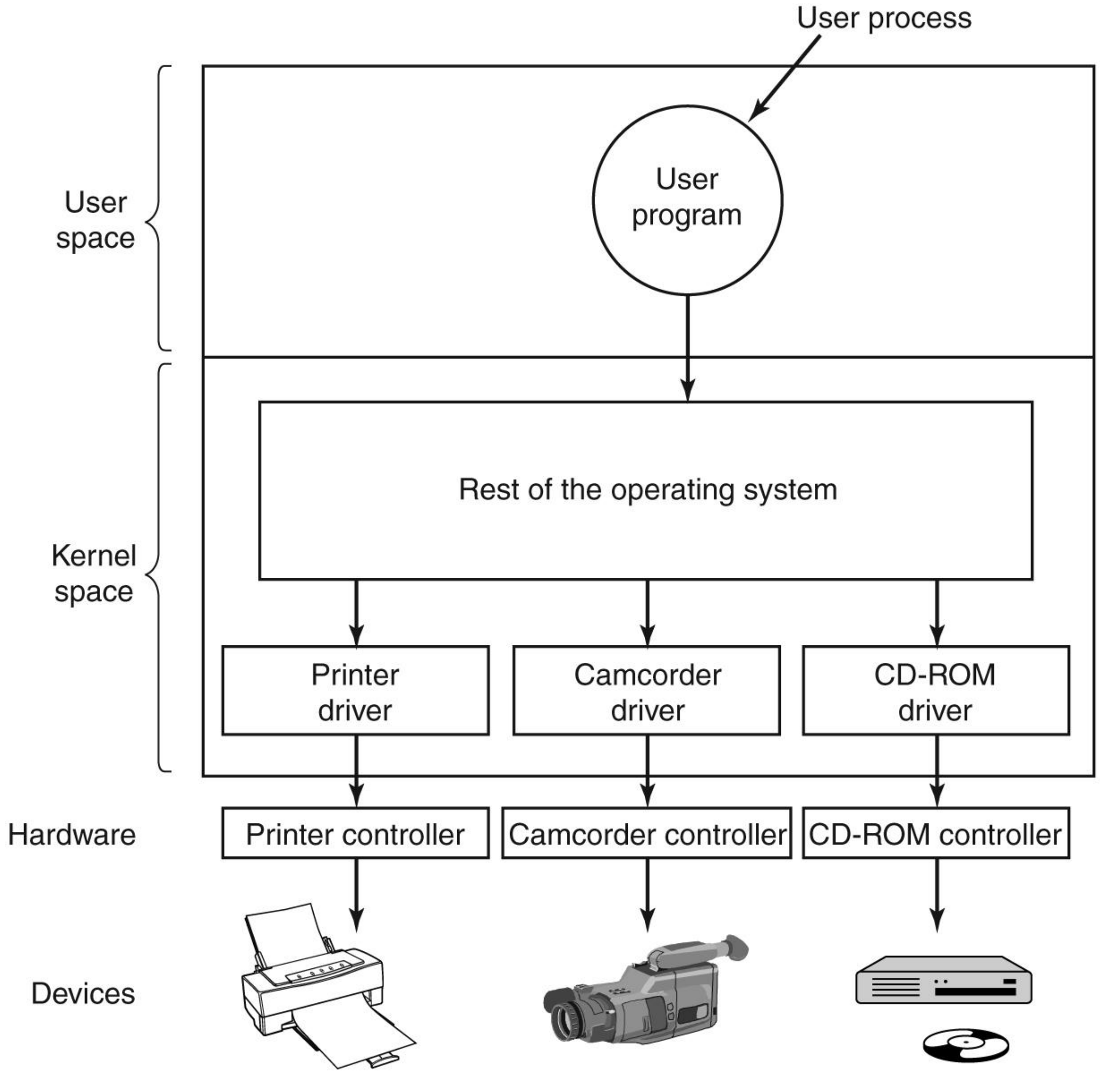
1. Save any registers not already saved
2. Set up context for interrupt service routine
3. Set up a stack
4. Acknowledge interrupt controller
5. Copy registers to process table
6. Run interrupt service routine
7. Choose process to run next
8. Set up context for process to run next
9. Load new process' registers
10. Start running new process

How About Device Drivers?

Device Drivers

- Device specific code for controlling the device
  - Read device registers from controller
  - Write device registers to issue commands
- Usually supplied by the device manufacturer.
- Can be part of the kernel or at user-space (with system calls to access controller registers).
- OS defines a standard interface that drivers for block devices must follow and another standard for driver of character devices.

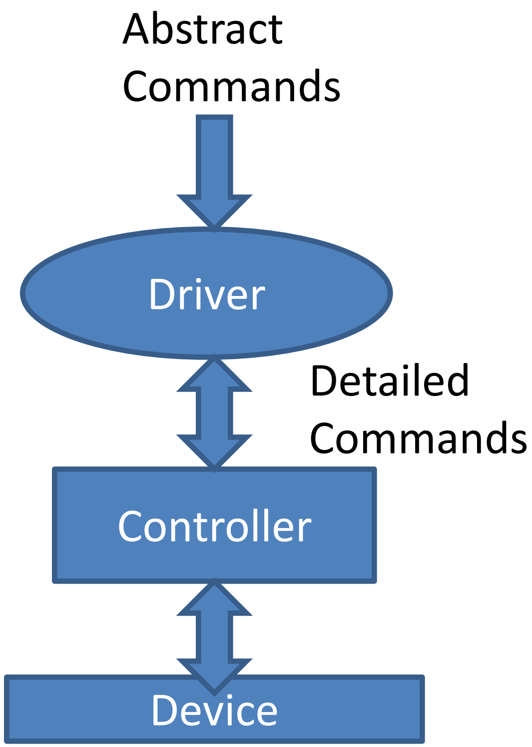
Device Drivers



Device Drivers

- Main functions:
  - Receive abstract read/write from layer above and carry them out
  - Initialize the device
  - Log events
  - Manage power requirements
- Drivers must be reentrant
- Drivers must deal with events

such as a device removed or plugged



Conclusions

- The OS provides an interface between the devices and the rest of the system.
- The I/O part of the OS is divided into several layers.
- The hardware: CPU, programmable interrupt controller, DMA, device controller, and the device itself.
- OS must *expand* as new I/O devices are added