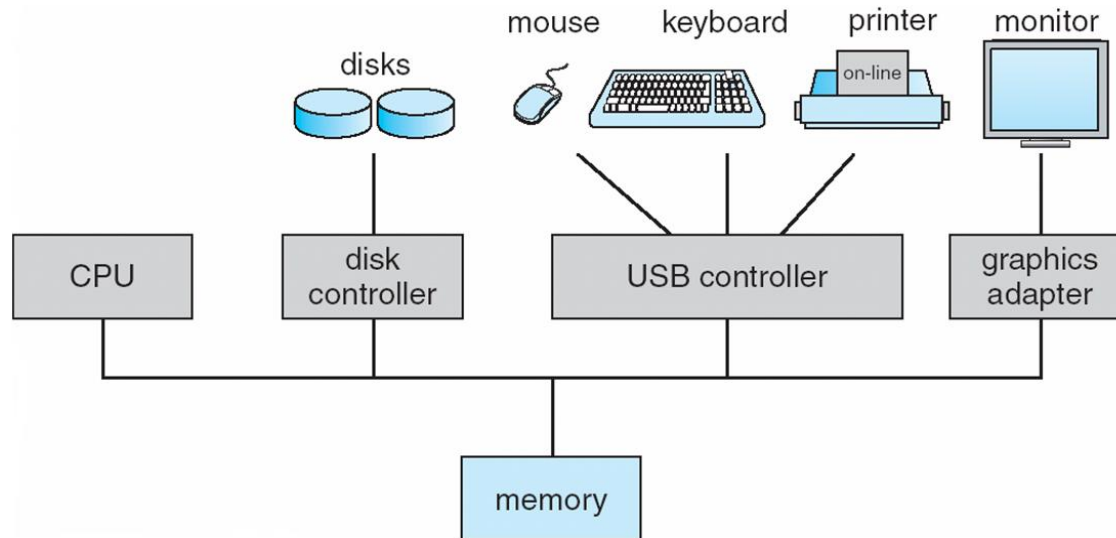# Interrupt

Computer Startup

Bootstrap program is loaded at power-up or reboot time

- Typically stored in ROM or EPROM, generally known as firmware
- Initializing all aspects of system
- Loading the most fundamental control software (note: operating system kernel) and starts execution

How does OS manage?

Synchronous:

      The OS checks the system status every a certain time period

Asynchronous:

      When there is a problem coming up, the OS solves it

Ideally, OS should run all the time on a CPU (like a responsible manager always stays in office and alert!) …. But CPUs are scarce resources, they should mostly be used for running applications … So …

A Sleeping OS

Once the OS has started an application (i.e., the app gets to run on the CPU), how can the OS continue managing the system?

- Need to pause the application and regain the CPU every now and then.

- How? Interrupt mechanism

How can OS regain control of CPU?

OS sets an alarm clock to ring after 10ms.

OS starts a program.

When the alarm rings, the running program is paused, OS regains CPU, do some housekeeping work if needed, and decides which program should run in the next 10 ms.

OS sets the alarm to ring after 10ms.

OS runs the program it chooses.

When the alarm rings, the running program is paused, OS regains CPU, …
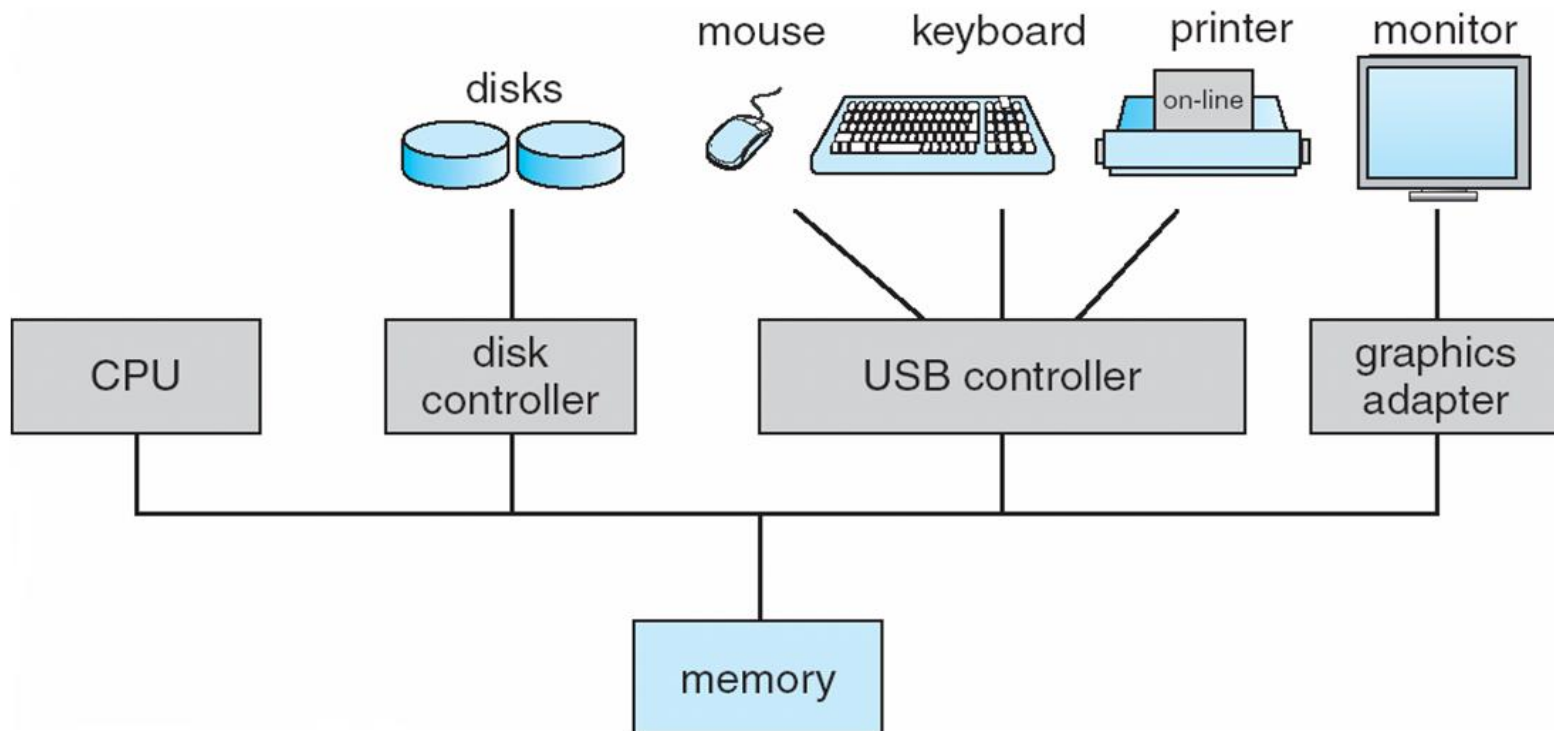
….

…

Interrupts

Who can generate interrupts?

- Hardware (I/O devices, timer, etc.)
- Software

Why I/O Devices generate Interrupt?

- I/O devices and the CPU can operate concurrently

- The CPU runs code

Why I/O Devices generate Interrupt?

- Each device controller is in charge of a particular device type
  - It has a local buffer and status/command registers
  - Input: device ➔ local buffer; output: local buffer ➔ device
- CPU also moves data between main memory and local buffers; controller CANNOT.
- Device controller informs CPU that it has finished an I/O operation by causing an *interrupt;* then, CPU can move data between memory and controller's local buffer.
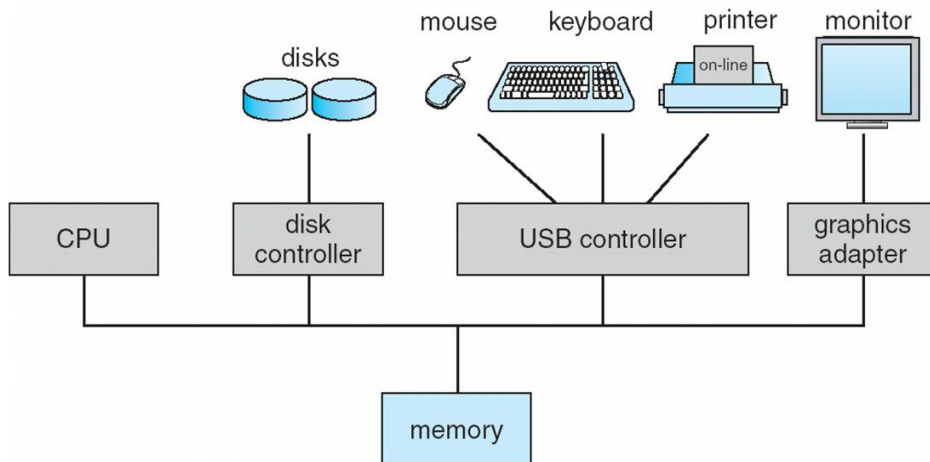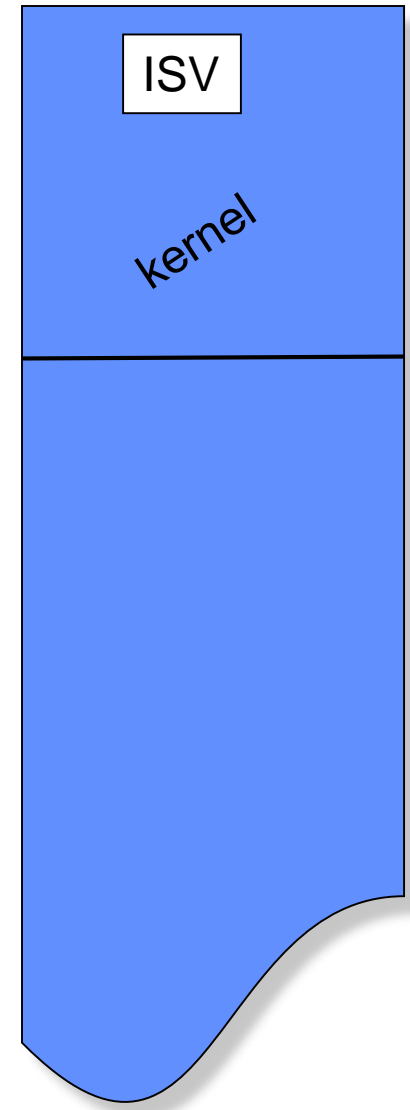
Interrupt Handling

CPU is normally running code. When it is interrupted:
1. Completes current instruction
2. Transfers execution to **Interrupt Service Routine (ISR)**
3. ISR usually disables interrupts and processes current one
4. When ISR completes, resume interrupted computation

Interrupt Handling

- When CPU is interrupted:
  1. Completes current instruction
  2. Transfers execution to **Interrupt Service Routine (ISR)**
  3. ISR usually disables interrupts and processes current one
  4. When ISR completes, resume interrupted computation

2. How to know location of ISR?
  - Small # of interrupt types are pre-defined, each type having an ISR
  - An array storing pointers to the ISRs
  - Array stored at fixed location, called I**nterrupt Service Vector (ISV)**
  - If interrupt type i occurs, jump to ISV[i]

ISV

kernel

Memory

Interrupt Handling

- When CPU is interrupted:
  1. Completes current instruction (if possible)
  2. Transfers execution to **Interrupt Service Routine (ISR)**
  3. ISR usually disables interrupts and processes current one
  4. When ISR completes, resume interrupted computation

4. How to resume?
- Before ISR is executed
  - CPU automatically saves address of interrupted instruction in system stack (a portion of memory used by the OS)
  - If ISR needs registers, must save old values (also on stack)
- After completing, ISR
  - Restores register values
  - Loads interrupted instruction address into PC (Program Counter, which is a register)

Example: CPU both run code and facilitate I/O devices
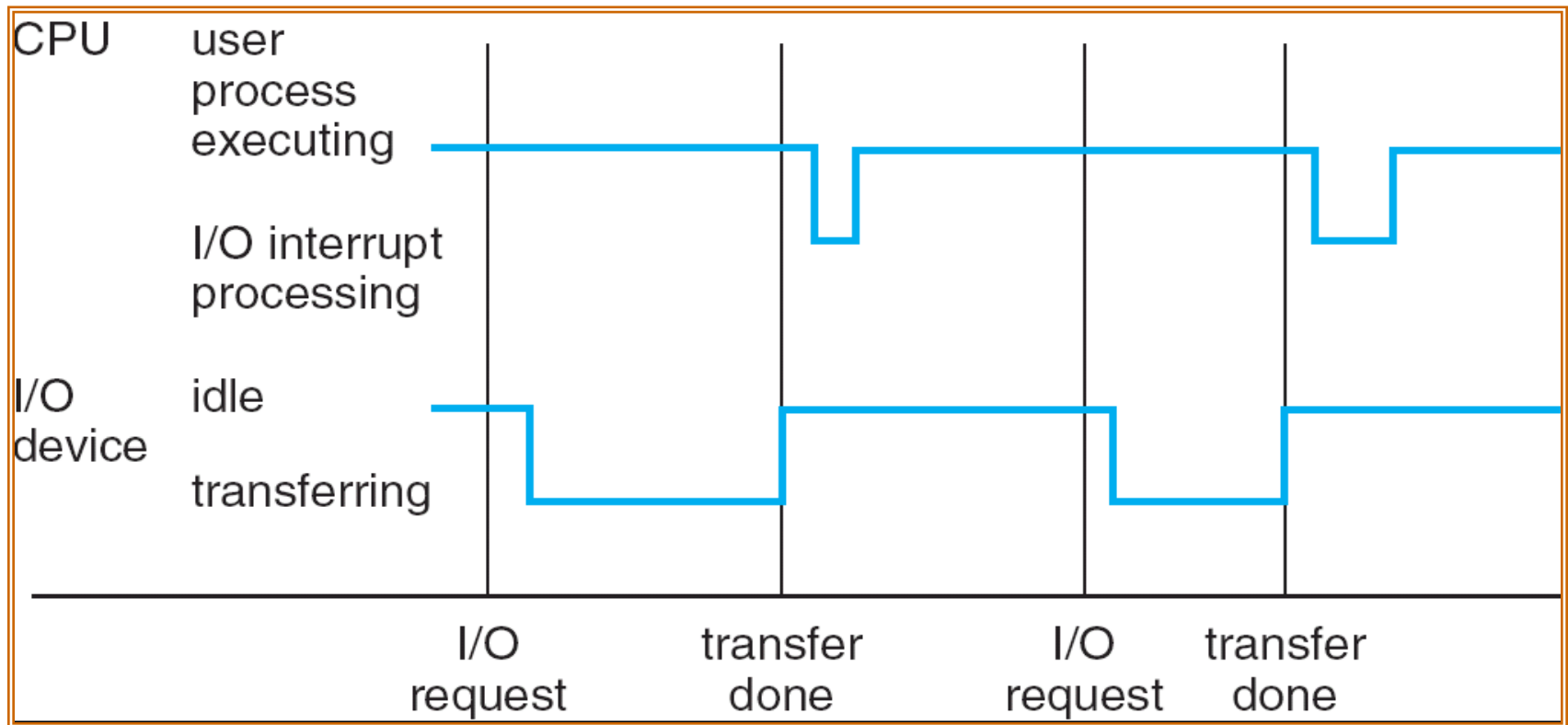
CPU computation and I/O processing may proceed concurrently

Interrupts

Who can generate interrupts?

- Hardware (I/O devices, timer, etc.)
- Software
  - Exceptions, e.g., division by zero or illegal instruction
  - System calls: user program requests the OS kernel to provide certain services (like the user program calls some functions which are part of the OS kernel)

Recap

Why do we need interrupt?

Who can trigger interrupt?

How is a hardware interrupt handled?