I/O Devices and Drivers

For I/O devices, two components are:

- **Bus:** signals for data transfer and communication
- **Controller:** electronics operate on hardware ports, bus, devices etc.

Device Drivers:

- Software that *plugs* into OS interface, tells OS about relevant hardware
- Required to *translate* OS commands to device commands via controller.
- **Windows Support:**
    - Allows drivers to run in *Kernel Mode*
- **UNIX Support:**
    - Able to run *some* drivers in *User Mode.*

Subsystem:

- A *Driver* is connected to the OS via an *interface* called a sub-system.
- *Sub-system* will translate OS commands to *generalized* commands
- *Driver* converts commands to device instructions.

Device Variations:

- Devices can vary in multitude of ways.
    - **Data Transfer Mode:** Devices operate on different data-sizes. *e.g.* keyboard vs. Hard-disk
    - **Access Method:** Devices may provide Sequential, or Random Access
    - **Transfer Schedule:** Devices may communicate, transfer data *Synchronosuly*, or *Asynchronously*
    - **Dedication:** Devices may be concurrently shareable, or *Dedicated* (*i.e.* no concurrent use).
    - **Device Speed:** Device speeds vary according to device requirements.
- Devices need loose categarization, and then issued only the appropirate commands
    - *e.g.* reading a hard-disk charcter-by-character is stupid

Data Transfer Mode:

- **Block Interface:**
    - used for devices such as hard-disks, CDs
    - Must support *read, write* commands; *seek* commands too if *Random-Access.*
- **Character Interface:**
    - used for devices like keyboards, printers etc. that require *sequential data-transfer*.
    - Useful for devices erratically producing small amounts of data *e.g.* keyboards
    - Must support *get, put* commands.

Bufferring:

- Area of memory storing data to be transferred.
- Good way to deal with speed mismatch b/w producer and consumer.
- **Double Buffering:**
    - Data arriving while buffer writes to another device needs to be handled
    - Two buffers are used:
        - One for storing incoming data, and another for transferring old data.
            - *e.g.* A keyboard might have two buffers; while one is emptied, second buffer records the keystrokes.
        - Excellent way to handle speed mismatch between producer/consumer

- **Spool:** Buffer-like system where all incoming requests are stored for devices that can only serve one job at a time.

Network Devices:

- Network communication tends to be serial
- Block read commnds *read*, *write*, *seek* not appropriate
- *Sockets* model used to communicate b/w client-server:
  - Server opens a *socket*
  - A client can *plugin* to a socket
  - *Packets* of data transferred b/w client and server
- Server uses *select* command to manage a set of clients
  - Displays what *sockets* are waiting to send or receive packets.

I/O Protection:

- All user acesses to I/O *mediated* through OS (since they invoke *system calls*) to confirm validity.
  - *e.g.* when a user tries to cancel someone's request on the printer
- Extra checking implemented at cost of overhead
- Special circumstances (*e.g.* high-performance applications and GPUs) might require waiving this condition to improve performance

Kernel I/O Data Structures:

- In Linux, everything is a file
- Kernel has a *global open-file table i.e.* table of any resources under use.
  - Each resource file has pointers to is important functions. *e.g. seek, read, write*
- Each process records which *files* or *resources* it has 'opened'
  - ...and holds pointer to *files* in the *global open-file table.*

I/O Scheduling:

- Kernel maintains a global structure of I/O requests.
- Kernel re-arranges requests to improve efficiency
- Priority *might* be accounted for; might override the importance of re-arranging it.

I/O Request to Hardware Operations:
Following example illustrates the process:

- Process issues a *read* command
  - Assume resource *file* already open
- Command *invokes* a system call
  - Returns data from *cache* or *buffer* if its been alredy read
  - Otherwise, *block* the process, schedule I/O operation, *sub-system* command the *device driver*
- Device-driver uses *buffer* to store incoming data; performs device singalling (*i.e.* writing into device registers)
- Controller *operates* device as commanded by driver
- Controller will *interrupt* driver or permit *polling* till request is finished.
- *Interrupt Handler* stores the incoming data, notifies *device-driver,* which notifies Kernel
- Kernel transfers data to *address space* of process, process gets un-blocked.
- Scheduler chooses process, execution resumes.