I/O Devices

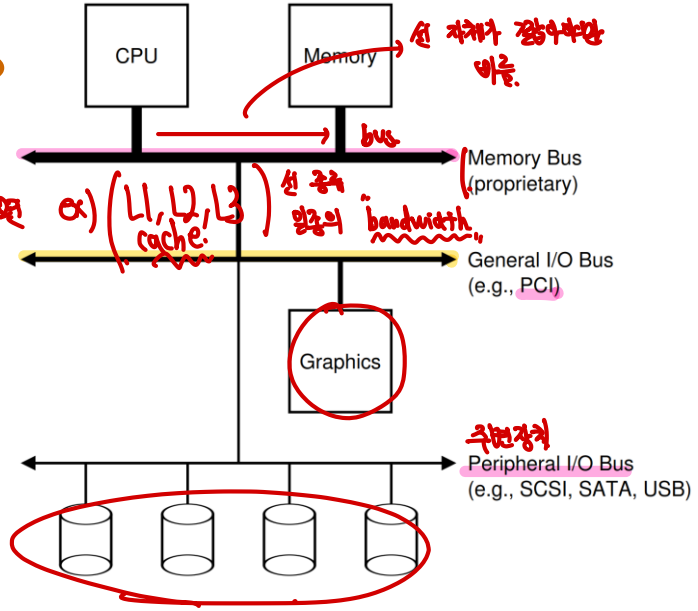
기계로, 하드웨어 프로그램 모든 입출력 장치
↓
이들을 OS에서 사용할 수 있도록 해주는
Device Driver.
=
Linux 70%

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

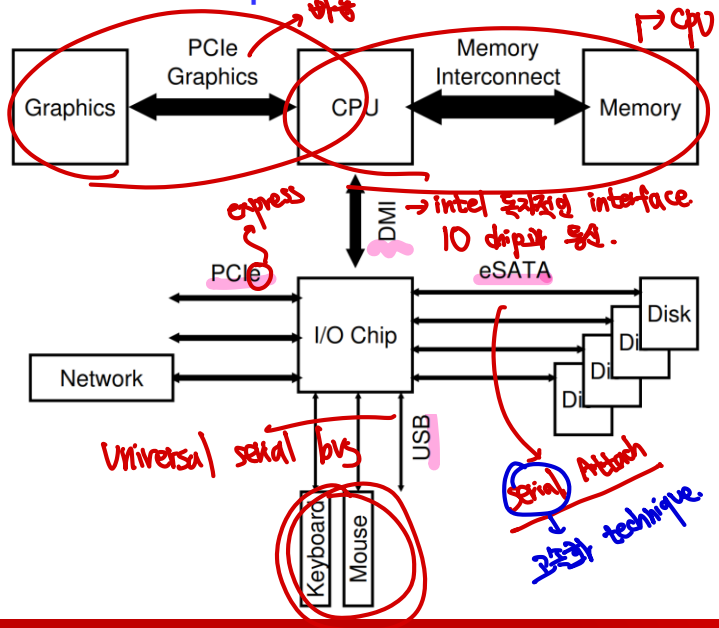
System Architecture

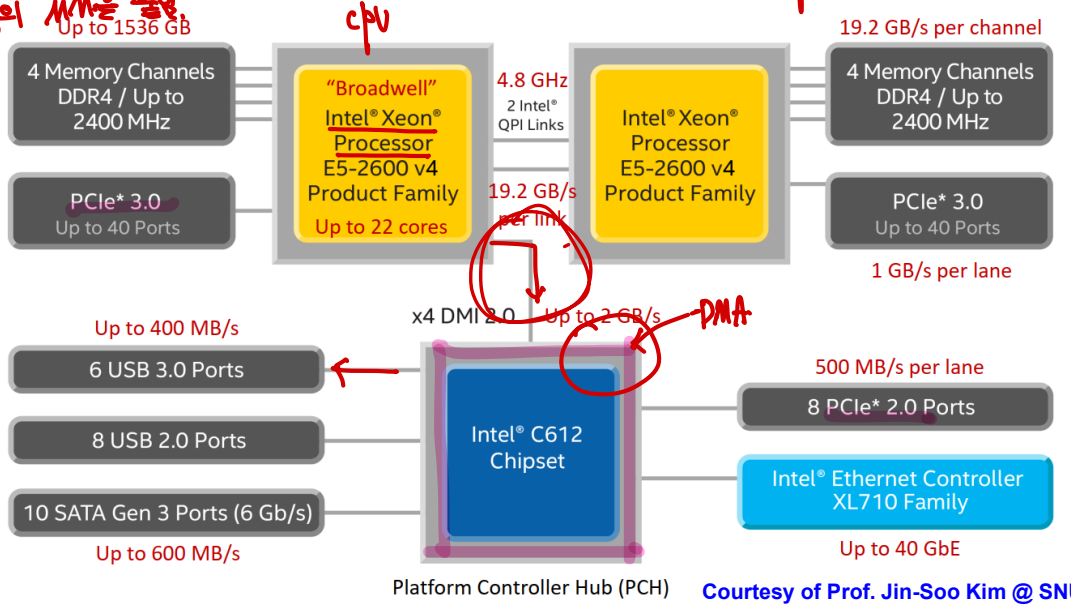
- **We introduce the concept of an input/output (I/O) device**
  - I/O is quite critical to computer systems; if there is no keyboard and monitor?
- **Let's look at a classical diagram of a typical system**
  - A single CPU attached to the main memory via memory bus or interconnect
  - Some devices are connected to the system via a general I/O bus (e.g. PCI); graphics and some other higher-performance I/O devices might be found here
  - Slow devices (e.g. keyboard, mouse, disk) are connected to the system via a peripheral bus, such as SCSI, SATA, or USB
- **Why do we need hierarchical structure?**
  - The answer is simple: physics and cost
  - The faster a bus is, the shorter it must be
  - Engineering a bus for high performance is quite costly
  - Thus, system designers have adopted this hierarchical approach



Modern System Architecture

- **Modern systems increasingly use specialized chipsets and faster point-to-point interconnects to improve performance**
  - A CPU connects to an I/O chip via Intel's proprietary DMI (Direct Media Interface) and the rest of the devices connect to this chip via various different interconnects
  - One or more hard drives connect to the system via the eSATA interface
  - The I/O chip includes a number of USB connections for low performance devices
  - Other higher performance devices can be connected via PCIe (Peripheral Component Interconnect Express)

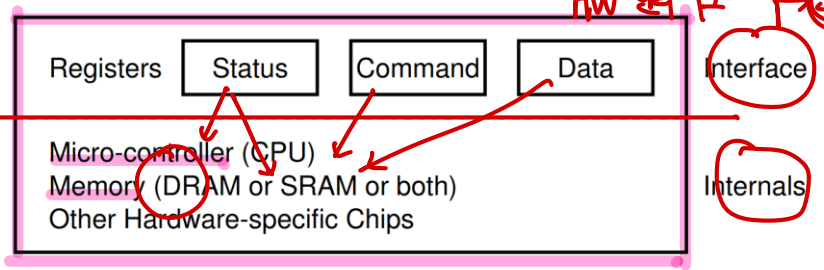




Courtesy of Prof. Jin-Soo Kim @ SNU

A Canonical Device & Protocol

- **A device has two important component**
  1) The hardware interface allows the system software to control its operation; this canonical device's hardware interface is comprised of three registers:
     (1) status register, which can read to see the current status of the device
     (2) command register, to tell the device to perform a certain task
     (3) data register, to pass data to the device, or get data from device
  2) Its internal structure is implementation specific and responsible for implementing the abstraction the device that presents to the system (e.g. chips, firmware)

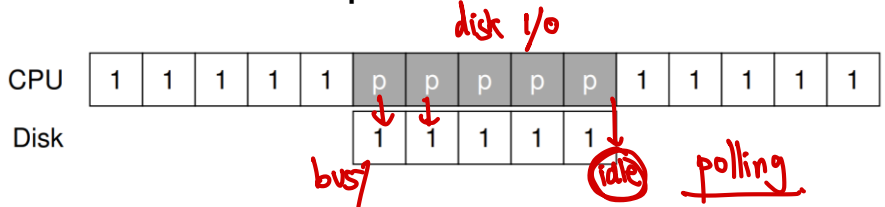


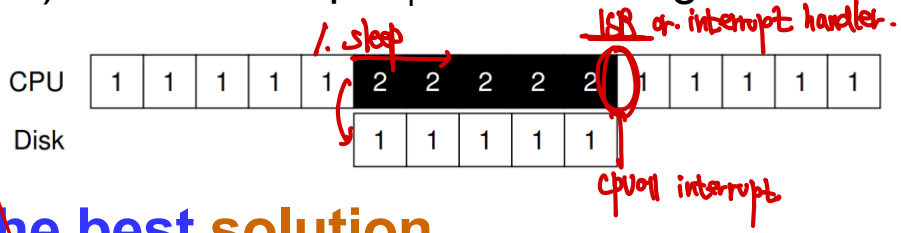

```
1) While (STATUS == BUSY)
       ; // wait until device is not busy
2) Write data to DATA register
3) Write command to COMMAND register
       (starts the device and executes the command)
4) While (STATUS == BUSY)
       ; // wait until device is done with your request
```

- **The protocol to interact between device and OS has four steps**
  1) OS keeps reading status register until device is ready to get command (polling)
  2) OS sends some data down to the data register (programmed I/O; PIO)
  3) OS writes a command to the command register to tell the device what to do
  4) OS waits for the device to finish by again polling it in a loop

Lowering CPU Overhead with Interrupts

- **The interrupt can improve the OS and device interaction**
  - Instead of polling the device repeatedly, OS can issue a request, put the calling process to sleep, and context switch to another task
  - When the device is finished with the operation, it will raise a hardware interrupt, causing CPU to jump into OS at a predetermined ISR or interrupt handler
- **Interrupts allow for overlap of computation and I/O**
  - Without interrupts: 1) the system simply spins, polling the status of the device repeatedly until the I/O is complete, 2) P_1 can run again
  - With interrupts: 1) OS runs P_2 on CPU while the disk services P_1's request, 2) an interrupt occurs when I/O finished, 3) OS wakes up P_1 and run it again

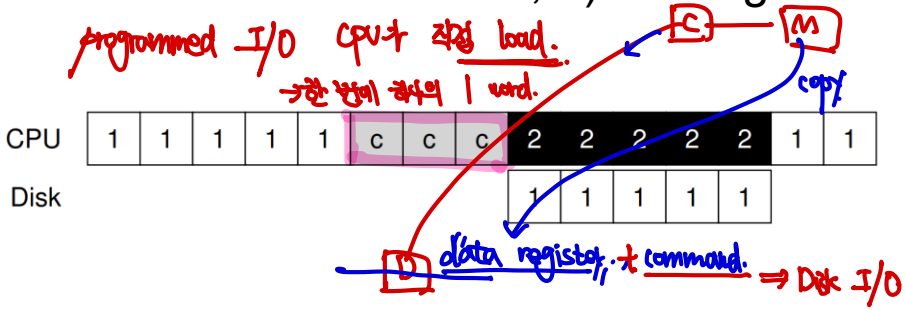


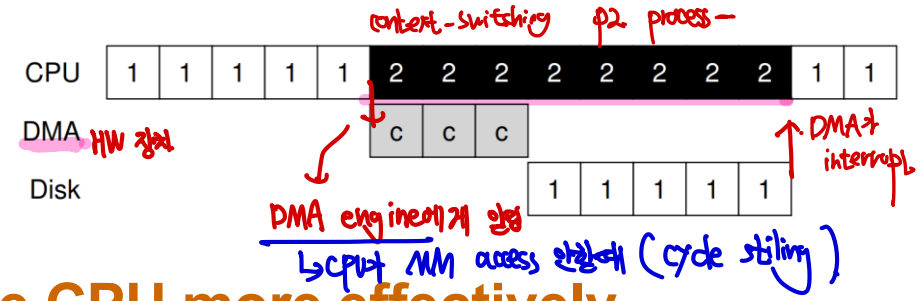


- **Using interrupts is not always the best solution**
  - Lots of interrupts will slow down the system: context switch, interrupt handling
  - The polling may be the best if a device is fast, otherwise, interrupt → hybrid
  - Livelock issue due to huge interrupts → polling or interrupt coalescing

More Efficient Data Movement with DMA

- **When using PIO to transfer a large chunk of data to a device:**
  - CPU is once again overburdened with a rather trivial task, and thus wastes a lot of CPU time and effort that could better be spent running other processes
  - PIO: 1) P_1 initiates I/O, which must copy data from memory to device explicitly, one word at a time, 2) I/O begins when the copy is complete, 3) P_1 runs again

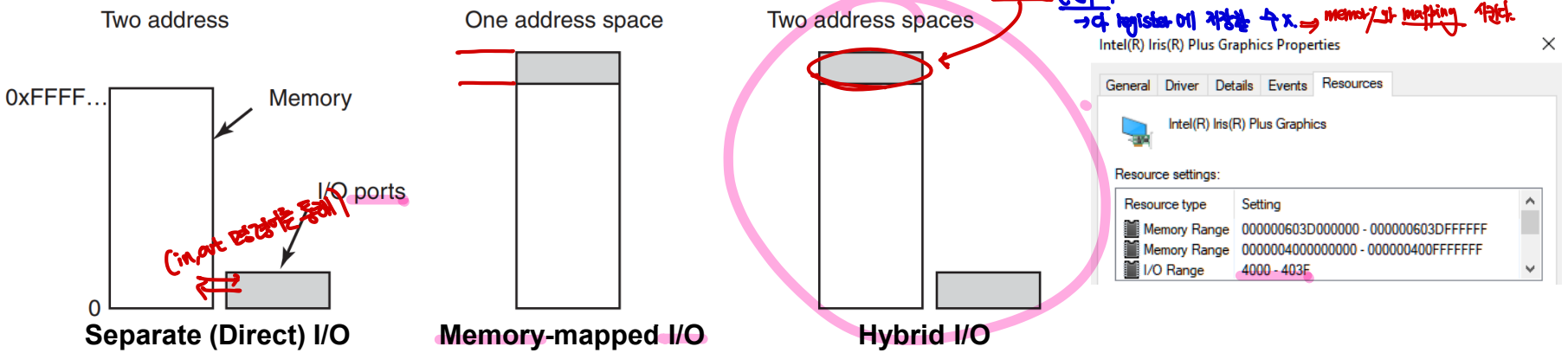




- **The DMA allows us to utilize the CPU more effectively**
  - A DMA engine is essentially a specific device in system that can orchestrate transfers between devices and main memory without much CPU intervention
  - DMA: 1) OS (P_1) programs the DMA engine by telling it the transfer information (i.e. source and target address of data, size of data, etc),
    2) OS is done with the transfer and can proceed with other work P_2
    3) DMA controller raises an interrupt upon the DMA transfer,
    4) P_1 can run again

Method of Device Interaction

- **The oldest method is to have explicit I/O instructions**
  - This is used by IBM mainframes for many years and these instructions specify a way for OS to send data to specific device registers (e.g. `in`, `out` on x86)
  - Such instructions are usually privileged and OS is the only entity to use them
- **The second one to interact with devices is memory-mapped I/O**
  - This makes device registers available as if they were memory locations
  - To access a register, OS issues a load (to read) or store (to write) the address
- **A hybrid I/O scheme is also available**
  - This scheme uses memory-mapped I/O data buffers and separate I/O ports for the control registers (e.g. graphics controller)



Fitting into the OS: The Device Driver

- **How to fit devices, each of which have very specific interfaces, into the OS, which we want to keep as general as possible?**
  - Abstraction: a device driver encapsulates any specifics of device interaction
- **Let's examine the Linux file system software stack**
  - A file system (and certainly, an application above) is completely oblivious to the specifics of which disk class it is using
  - It simply issues block read and write requests to the generic block layer
  - Then, the device driver handles the details of issuing the specific request
  - This also provides a raw interface to devices, which enables special application (e.g. disk defragmentation) to directly read and write without the abstraction