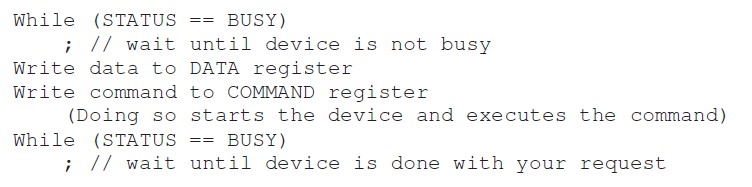
Why a hierarchical structure?

* Put simply: physics & cost.
* The faster a bus is, the shorter it must be.
  + Thus, a high-performance memory bus does not have much room to plug devices and such into it.
* Engineering a bus for high performance is quite costly.
* Thus, system designers have adopted this hierarchical approach:
  + components that demand high performance (such as the graphics card) are nearer to the CPU.
  + Lower performance components are further away.
* The benefits of placing disks and other slow devices on a peripheral bus are manifold.
  + in particular, you can place many devices on it.
* Device interface comprised of 3 registers:

1. **status register**: can be read to see device status.
2. **command register**: to tell device to perform a certain task.
3. **data register**: to pass data to device or get data from device.

* By reading and writing these registers, the operating system can control device behavior.
* A typical interaction that OS might have with the device to get the device to do something on its behalf:
* The protocol has 4 steps:
  1. OS waits until device ready to receive a command by repeatedly reading status register → check/**polling** the device.
  2. OS sends some data down to the data register.
* E.g., if this were a disk, multiple writes would need to take place to transfer a disk block (say 4KB) to the device.
* When the main CPU is involved with data movement → **programmed I/O (PIO).**

1. OS writes a command-to-command register.

• implicitly lets the device know that both the data is present and that it should begin working on the command.

1. The OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished. • It may then get an error code to indicate success or failure.

• Pros and Cons

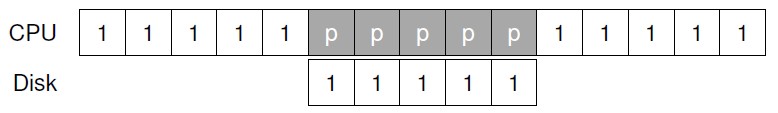
* Simple & it works.
* Inefficiencies & inconveniences waste a great deal of CPU time.

## Lowering CPU Overhead with Interrupts

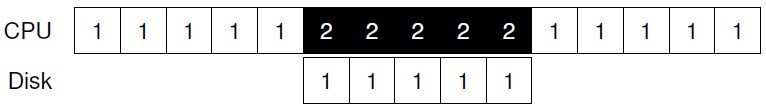
* Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task.
* When device finished with operation, it will raise hardware **interrupt**, causing CPU to jump into OS at a **pre-determined** **interrupt service routine (ISR)** or **interrupt handler**.
  + The handler: a piece of OS code that will finish the request.
  + E.g., by reading data and perhaps an error code from device and waking the process waiting for the I/O, which can then proceed as desired.
* Interrupts thus allow for **overlap** of computation & I/O, which is key for improved utilization.

## Lowering CPU Overhead with Interrupts

Without interrupts



With interrupts



Note:

* “1”: Process 1
* “2”: Process 2
* “p”: polling

When interrupts shouldn’t be used

* Interruption is not *always* the best solution.
* Fast devices
* The first poll usually finds devices to be done with task.
* Using an interrupt slow down the system:
  + switching to another process, handling interruptions, and switching back to the issuing process is expensive.
  + Thus, for fast devices, it may be best to poll.
  + If the device’s speed is unknown, or sometimes fast and sometimes slow, it best to use a **hybrid** that polls for a little while and then, if the device is not yet finished, uses interrupts.
* This **two-phased** approach may achieve the best of both worlds.

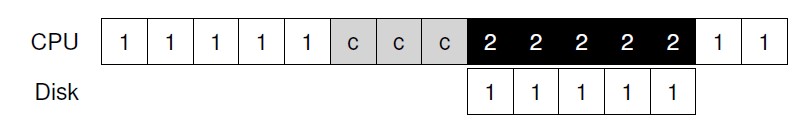
**Coalescing**: Sometimes, it’s better for the device to wait for a bit before delivering interrupt to CPU.

* While waiting, other requests may soon be completed.
* Thus, multiple interrupts can be *combined* into a single interrupt delivery.
* Lowers overhead of interrupt processing.
* Waiting too long will increase latency of request → a trade-off.

## More Efficient Data Movement with DMA

* With **programmed I/O** (even with interruptions), to transfer large chunk of data to device, CPU overburdened with a rather trivial task.

– wastes a lot of time & effort that could better be spent running other processes.



Note:

* “1”: Process 1 • “2”: Process 2
* “c”: CPU copies data from memory to device one word at a time.

## More Efficient Data Movement with DMA

* Solution: **Direct Memory Access (DMA)**
  + A device within a system that can orchestrate transfers between devices & main memory without much CPU intervention.
* To transfer data to device, works as follows:
  + OS programs DMA engine by telling it where data lives in memory, how much data to copy, and which device to send it to.
  + At that point, the OS is done with transfer and can proceed with other work.
  + When DMA complete, DMA controller raises an interrupt, and OS thus knows transfer is complete.

What is DMA and Why it is used?

Direct memory access (DMA) is a **mode of data transfer** between the memory and I/O devices. This happens **without the involvement** of the processor. We have two other methods of data transfer, **programmed I/O** and **Interrupt driven I/O**.

DMA Advantages and Disadvantages

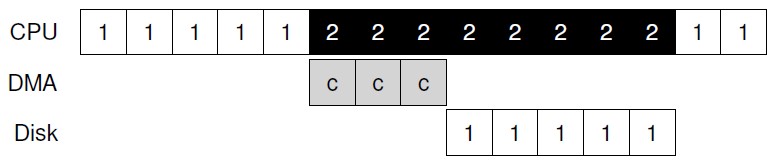
* Advantages:

1. Transferring the data without the involvement of the processor will **speed up** the read-write task.
2. DMA **reduces the clock cycle that** requires to read or write a block of data.
3. Implementing DMA also **reduces the overhead** of the processor.

• Disadvantages

1. As it is a hardware unit, it would **cost** to implement a DMA controller in the system.
2. Cache **coherence** problems can occur while using DMA controller.

## More Efficient Data Movement with DMA



* Copying of data is now handled by DMA controller.
* Because the CPU is free during that time, OS can do something else, here choosing to run Process 2.
* Process 2 thus gets to use more CPU before Process 1 runs again.

Methods Of Device Interaction

* Unanswered questions so far:
  + How should hardware communicate with a device?
  + Should there be explicit instructions? – Or are there other ways to do it?
* Two primary methods of device communication:

1. Explicit **I/O instructions**.

* First and oldest (used by IBM mainframes for many years)
* Specify a way for OS to send data to specific device registers.
* E.g., on x86, *in* and *out* instructions used to communicate with devices.

– Source = register, destination = specific port which names the device.

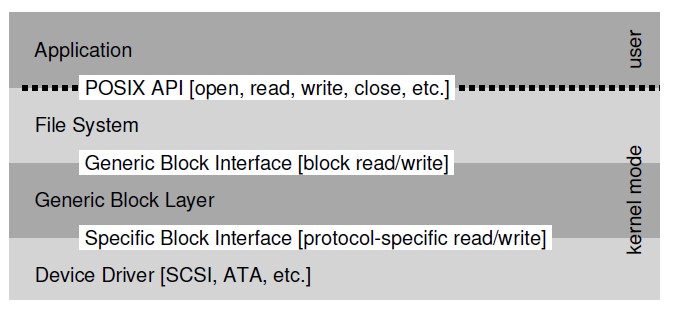
**2. Memory-mapped I/O**

* Hardware makes device registers available as if they were memory locations.
* To access a particular register, the OS issues a *load* (to read) or *store* (to write) the address.
* Hardware then routes load/store to device instead of main memory.
* Not great advantage to one approach or the other.

## Fitting Into The OS: The Device Driver

* Remaining problem:
  + How to fit devices with very specific interfaces into OS, which to be kept as general as possible?
  + How to keep most of OS device-neutral, thus hiding details of device interactions from major OS subsystems?
* Problem solved through **abstraction**:
  + At lowest level, a piece of software in OS must know in detail how a device works → **device driver**.
  + Device interaction specifics are *encapsulated* within it.

## Fitting Into The OS: The Device Driver



* Example of how a device driver can help (in Linux file system software stack).
* A file system (and certainly, an application above) is completely oblivious to specifics of which disk class it is using.
* It simply issues block read & write requests to the generic block layer, …
* … which routes them to the appropriate device driver, …
* … which handles the details of issuing the specific request.
* Although simplified, the diagram shows how such detail hidden from most of OS.

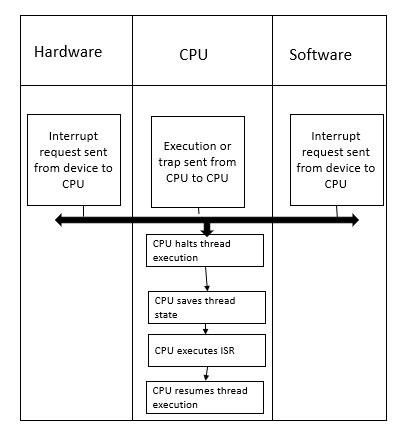
# Downside of encapsulation

* Such encapsulation (provided by device drivers) described in previous slides can have its downside.
* If there is a device that has many special capabilities, but must present (i.e., *expose*) a generic interface to the rest of the kernel, those special capabilities will go unused.
* E.g., of above scenario:
  + In Linux with SCSI devices, which have very rich error reporting.
  + Because other block devices (e.g., ATA/IDE) have much simpler error handling, all that higher levels of software ever receive is a generic IO error code
  + any extra detail that SCSI may have provided is thus lost to the file system.

**What are interrupts and how interrupt handling is done in modern operating systems?**

## Interrupts are generally called signals which are generated by software or hardware when a particular event or process requires immediate attention. So, the signal informs the processor about a high-priority and urgent information demand causing an interruption in the current working process.

* Thus, whenever an interruption occurs the processor finishes the current instruction execution and starts the execution of the interrupt known as interrupt handling. Moreover, for every interrupt handling to occur there is an Interrupt service routine (ISR) or interrupt handler.
* The interrupt handling mechanism of an operating system accepts a number which is an address and then selects what specific action to be taken which is already mentioned in the interrupt service routine. In most architecture, the address is stored in a table known as a vector table.



# Summary

* You should now have a basic understanding of how an OS interacts with a device.
* Two techniques, the interrupt and DMA, have been introduced to help with device efficiency.
* Two approaches to accessing device registers, explicit I/O instructions and memory mapped I/O, have been described.
* Finally, the notion of a device driver has been presented, showing how OS itself can encapsulate low-level details and thus make it easier to build the rest of the OS in a device neutral fashion.