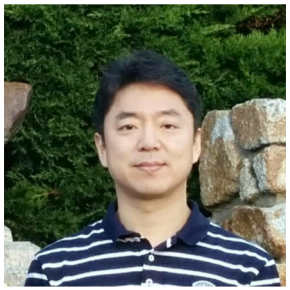
CSCI 333 : Storage Systems

Ch 36: I/O Devices

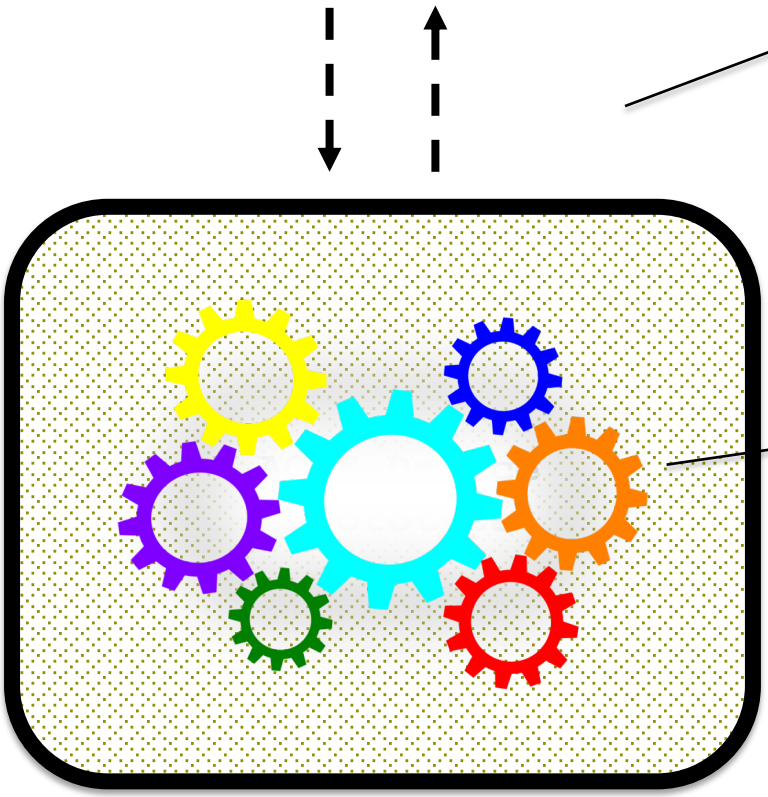
(Adapted from content by Youjip Won, Associate Professor at Dept. of Electrical and Computer Engineering, Hanyang University, Seoul, Korea)



I/O Devices

- Input/Output is arguably what makes our programs interesting. We need to be able to send and receive data to/from various hardware.
- Issues discussed in this video:
  - How should I/O be integrated into system hardware designs?
  - What are the general mechanisms for communication?
  - How can we efficiently communicate between software and I/O hardware?
- Questions to think about as we discuss:
  - Which things can we, as software developers, change?
  - What things can we, as software developers, actually know?

What Do We Actually Need to Know As Software Designers?



Device Interface:

- Message formats/parameters
- Return types/codes
- Error conditions/failure behavior
- Physical interface specification
  - Data/Power/Ports

Device Internals:

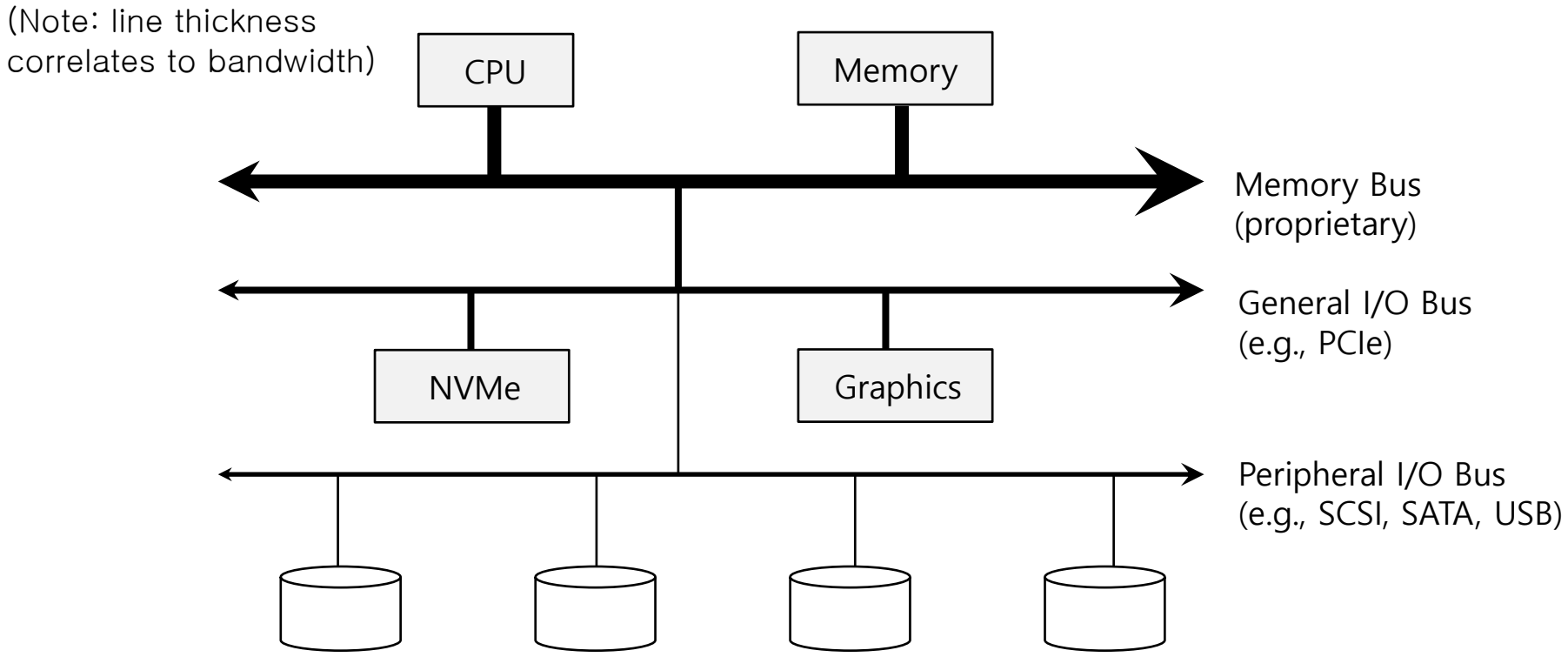
- Data structures used
- Firmware design/op. complexity
- Physical component details
- Lifetime (wear-out) and failure modes (fail-stop or degradation)

What Do We Actually Need to Know As Software Designers?

- Knowing the interface is necessary to USE the device
- Knowing the details of the device internals can be helpful to optimize system performance.
  - Both factor into the way we design software on top of the device.
  - Device heterogeneity and the increasing internal complexity of newer devi ces makes it very hard to build a general-purpose system that performs o ptimally

Question 1: How do we connect to I/O devices?

Structure of input/output (I/O) device



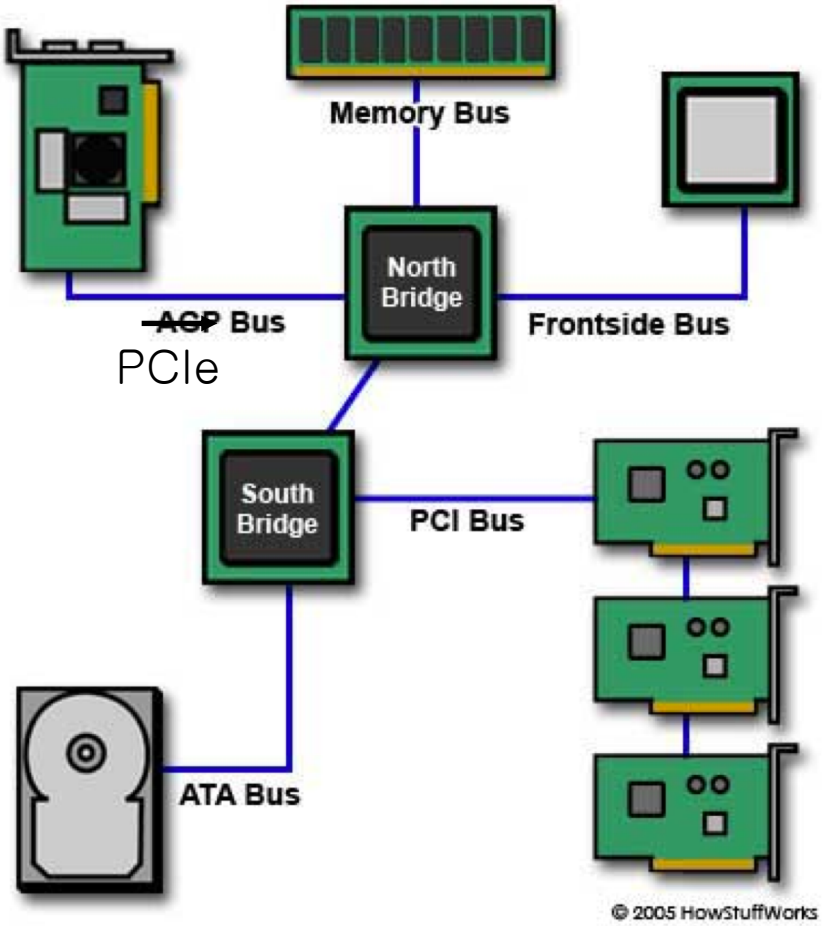
Prototypical System Architecture

CPU is attached to the main memory of the system via a fast memory bus.

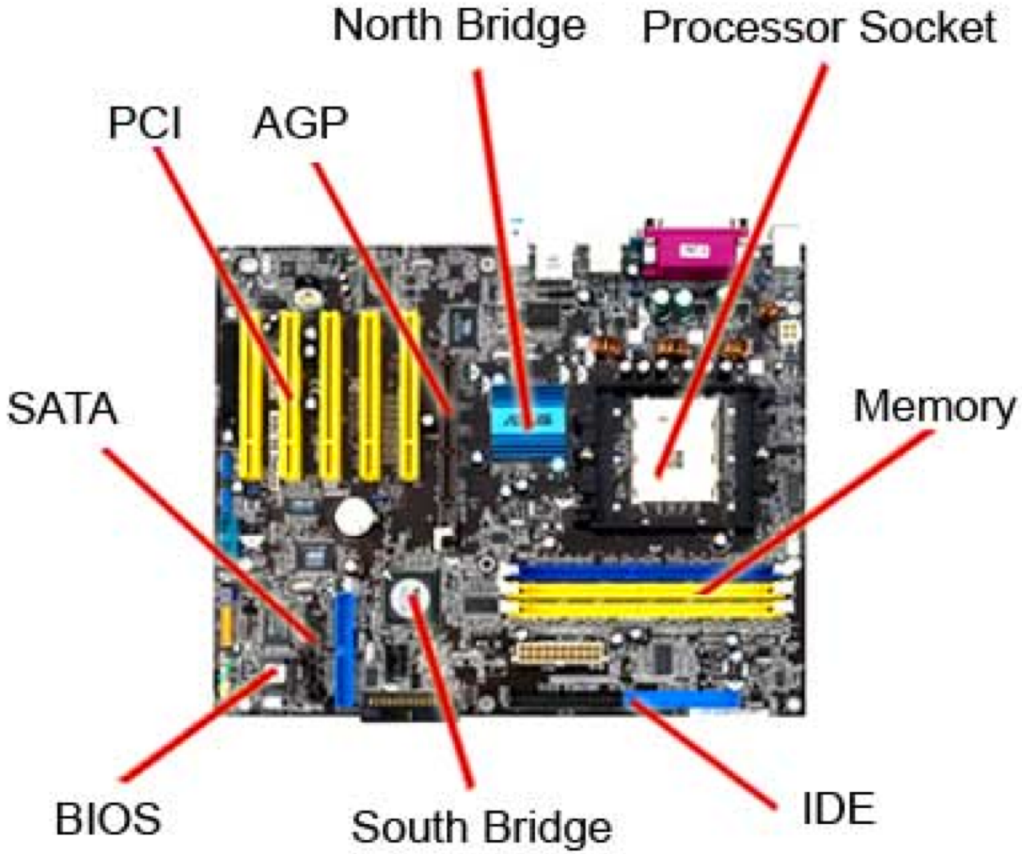
A small number of fast devices are connected to the system via a general I/O bus.

(Potentially many) slower devices are connected via a Peripheral bus.

Structure of a Northbridge/Southbridge Chipset

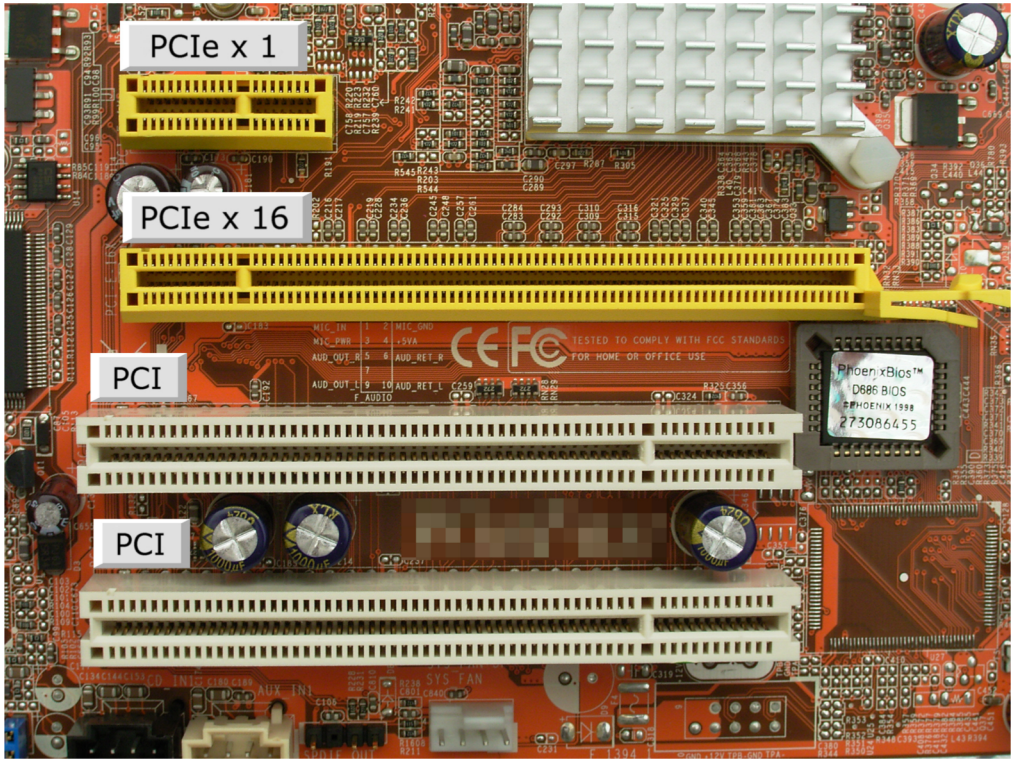


https://s.hswstatic.com/gif/motherboard-busses.jpg



https://turbofuture.com/computers/the-motherboard-components

Interfaces are constantly evolving: e.g., PCI vs. PCIe



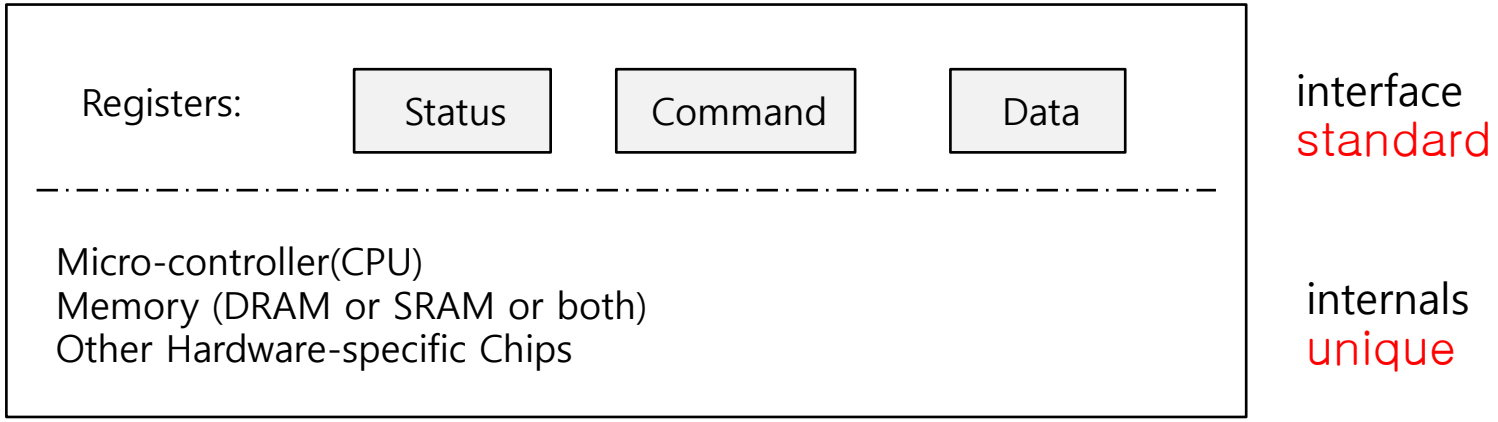
A motherboard with two 32-bit PCI slots and two sizes of PCI Express slots

The original uploader was Smial at German Wikipedia. [CC BY-SA 2.0 DE (https://creativecommons.org/licenses/by-sa/2.0/de/deed.en)]

Question 2: How do we communicate with I/O devices?

A Canonical Device

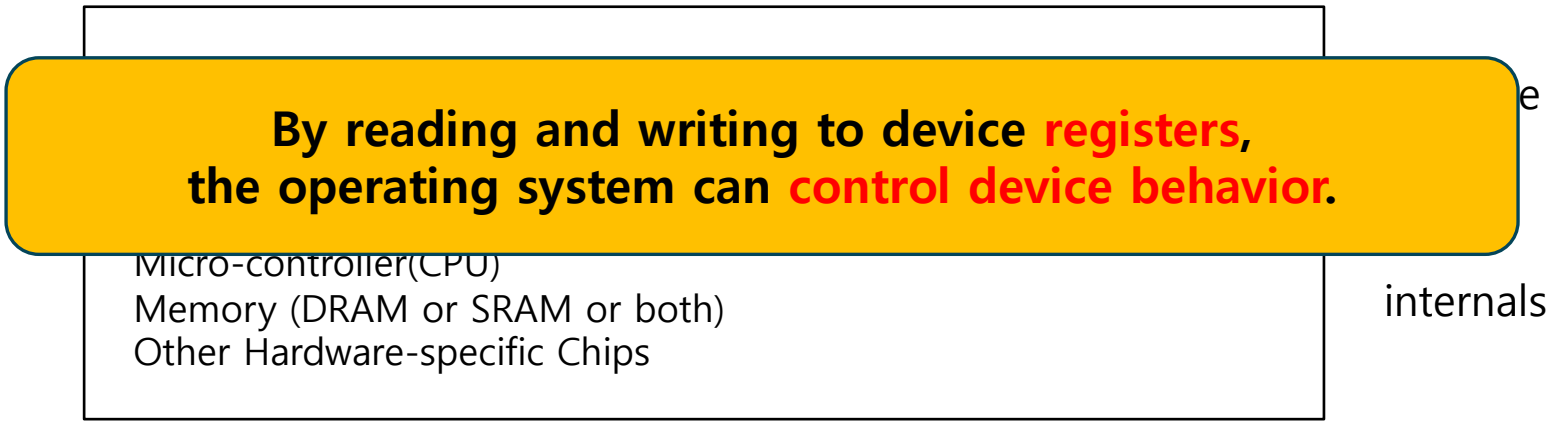
- **Hardware interface** allows system software to control device operation.
- **Device Internals** are device/implementation specific.



Canonical Device

Hardware Interface for a Canonical Device

- **status register**
  - reflects the current status of the device (e.g., BUSY, READY)
- **command register**
  - instructs the device to perform a certain task
- **data register**
  - Passes data: OS->device (write), or device->OS (read)



Canonical Device

Device interaction option 1: polling

- One common technique to interact with a device is **polling**.
- Typical **polling** interaction:

```
while ( STATUS == BUSY)
    ; //wait until device is not busy
<write data to data register>
<write command to command register> //starts device & execs command
while ( STATUS == BUSY)
    ; //wait until device is done with your request
```

- What is a disadvantage?

Downsides of Polling

- The OS waits until the device is ready by **repeatedly** reading the status register.
  - Simple and easy to understand/implement.
  - **Wastes CPU time just waiting for the device**.
    - System could be making progress if CPU were to switch to another task.

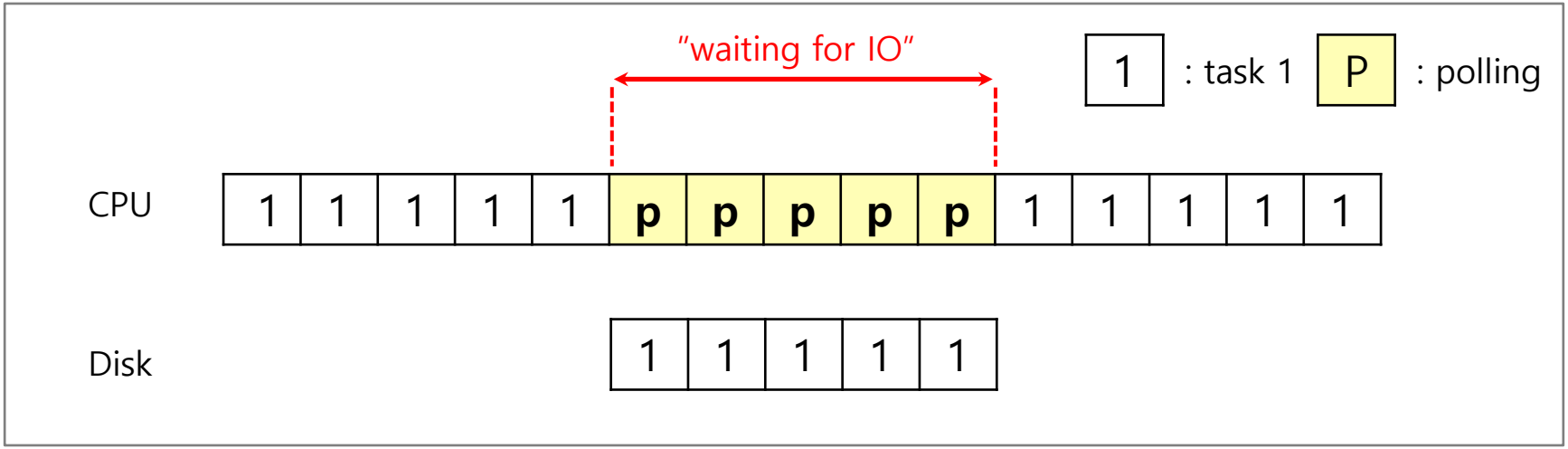


Diagram of CPU utilization using polling

Device Interaction Option 2: interrupts

- **The OS puts the process that is requesting I/O** (1) **to sleep** and then context switches to another process (2).
- When the device is finished, an interrupt wakes the sleeping process
  - **Advantage: CPU and the disk are both utilized.**

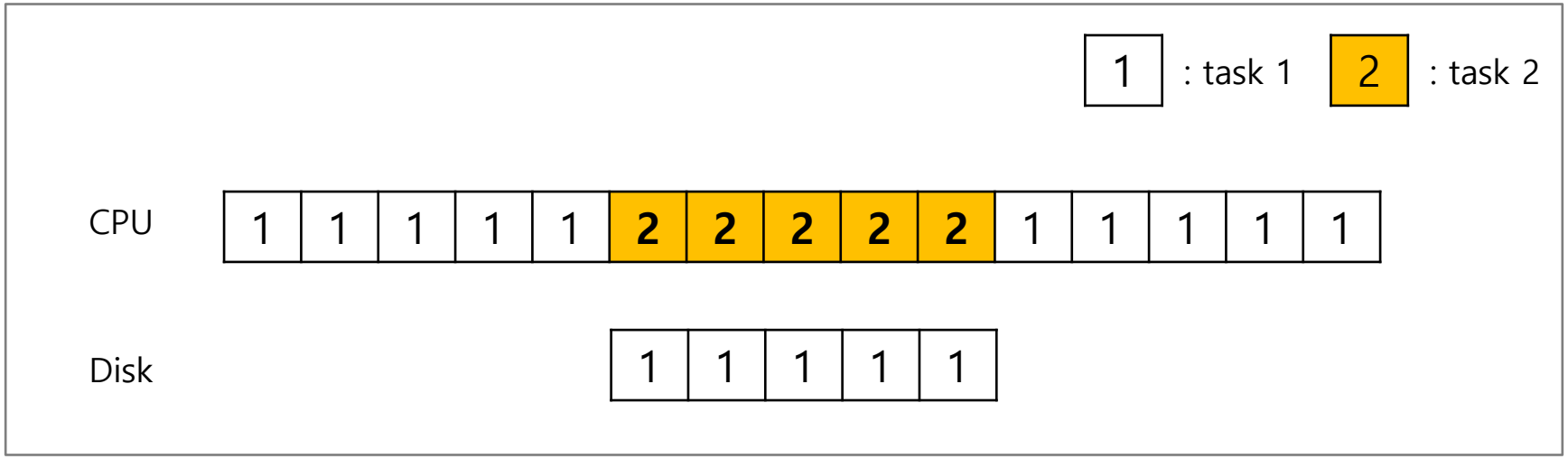


Diagram of CPU utilization using interrupts

Polling vs interrupts

- Is there one “best” solution?
  - What if device is fast?
  - What if device is slow?
  - What if performance is bimodal (sometimes fast, sometimes slow)?

Direct Memory Access (DMA)

- The CPU may waste a lot of time copying large chunks of data from memory to the device.

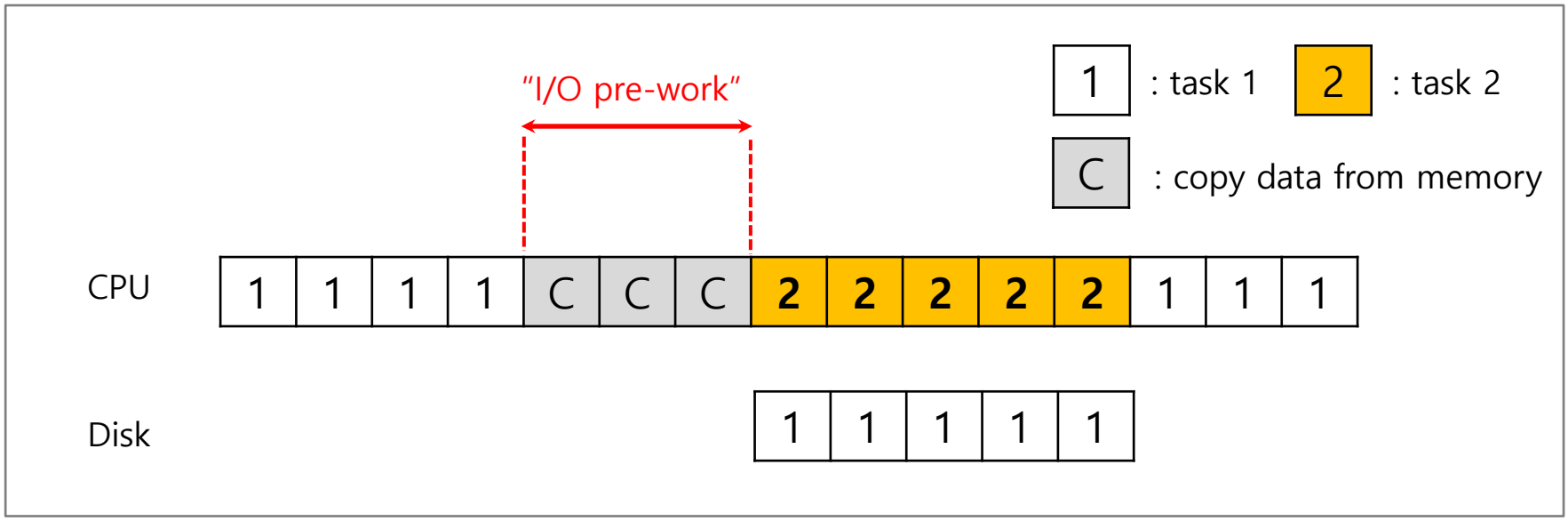


Diagram of CPU utilization without DMA

DMA (Direct Memory Access)

- **Technique: offload the copy** from memory to device
  - Really only need to know 2 things
    - where the data lives in memory (source and destination)
    - how much data to copy
- When completed, DMA raises an interrupt, I/O begins on Disk.

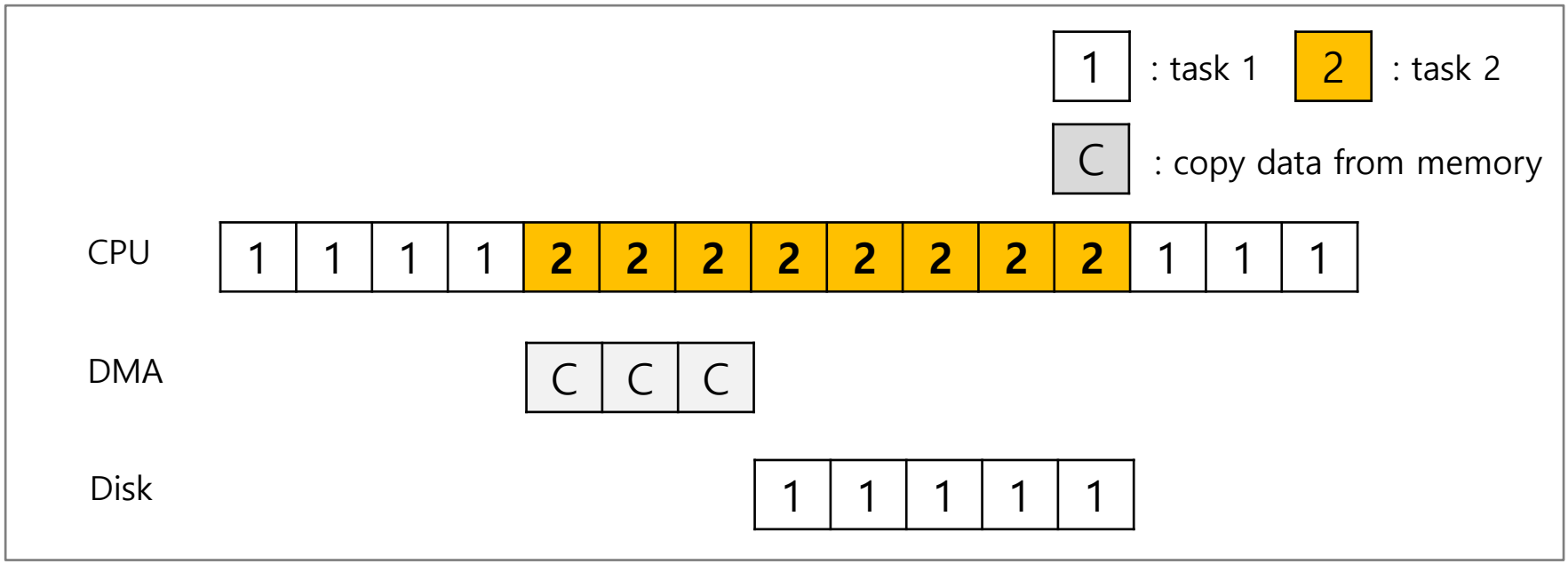
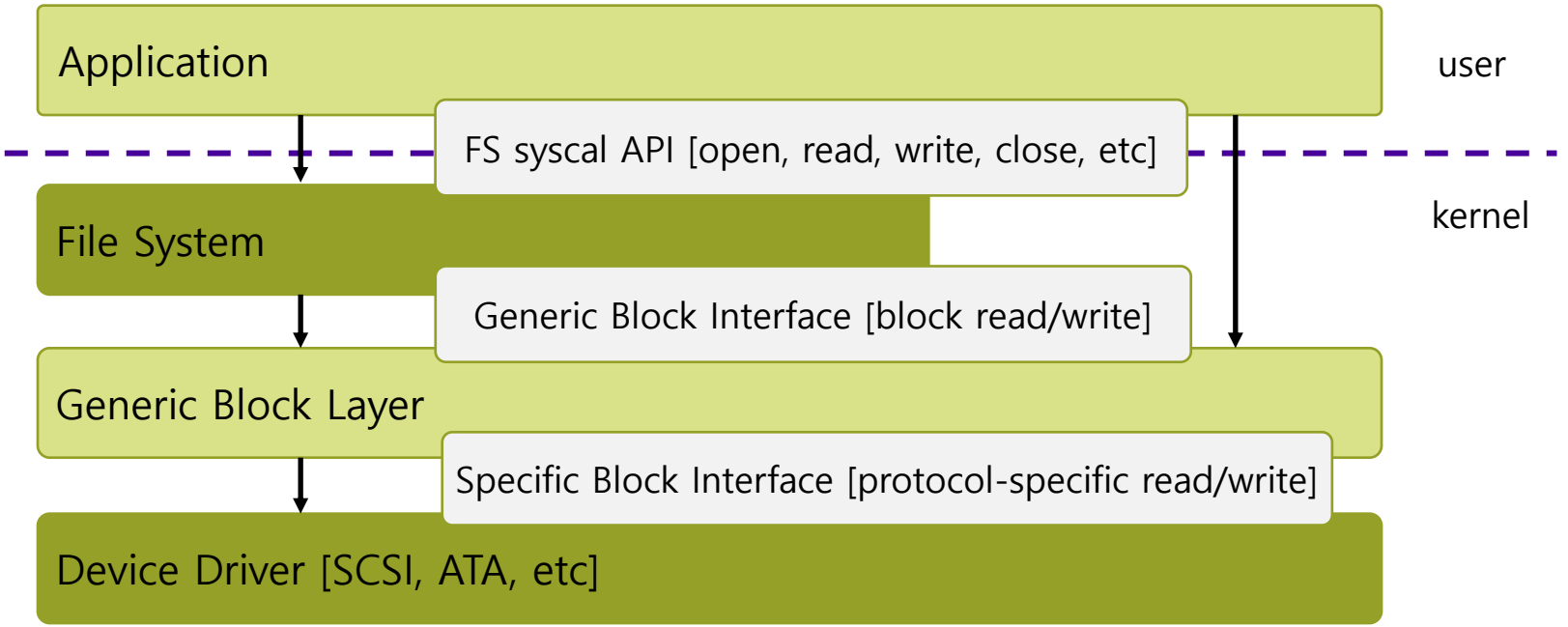


Diagram of CPU utilization using DMA

Question 3: How do we support so many different I/O devices in one system?

Layers and Abstraction

- Recall the Linux software storage stack and the interfaces between layers



The Software Storage Stack

Problems With the File system Abstraction

- If there is a device with nonstandard capabilities, those capabilities will go unused in the generic interface layer.
  - General Problem: Abstraction *hides* details, sometimes to our detriment
    - Any workarounds?

- Over 70% of OS code is found in device drivers.
  - Many device drivers are available to your OS by default because you might one day plug a device into your system
    - Standard OS distributions often contain a superset of your actual hardware's drivers, but modules can be loaded on demand
  - Device drivers are a primary contributor to **kernel crashes**, and often introduce **security vulnerabilities**

Final Thoughts

- There are things we know (device interface), things we guess (general internal structure), and things we are blind to (internal state of device)
  - We try our best to do the things that device designs do well and avoid the things that we expect will lead to bad behaviors
- There are different ways to interact with devices, with pros/cons
  - Polling is good for fast devices, but "busy waits"
  - Interrupts are good for slower devices
  - Hybrid approaches can also make sense
- The layered design of our systems is a huge boon, but as devices get faster and faster, software overheads become a problem
  - Generic design limits adoption of device-specific features
  - Sometimes "jumping over" the OS is our best path to good performance

Credit & Thanks

- This lecture slide set was initially developed for an Operating System course in the Computer Science Dept. at Hanyang University. This lecture slide set draws from the OSTEP book written by Remzi and Andrea at the University of Wisconsin. It has been modified substantially, but much credit deservingly goes to the original authors.