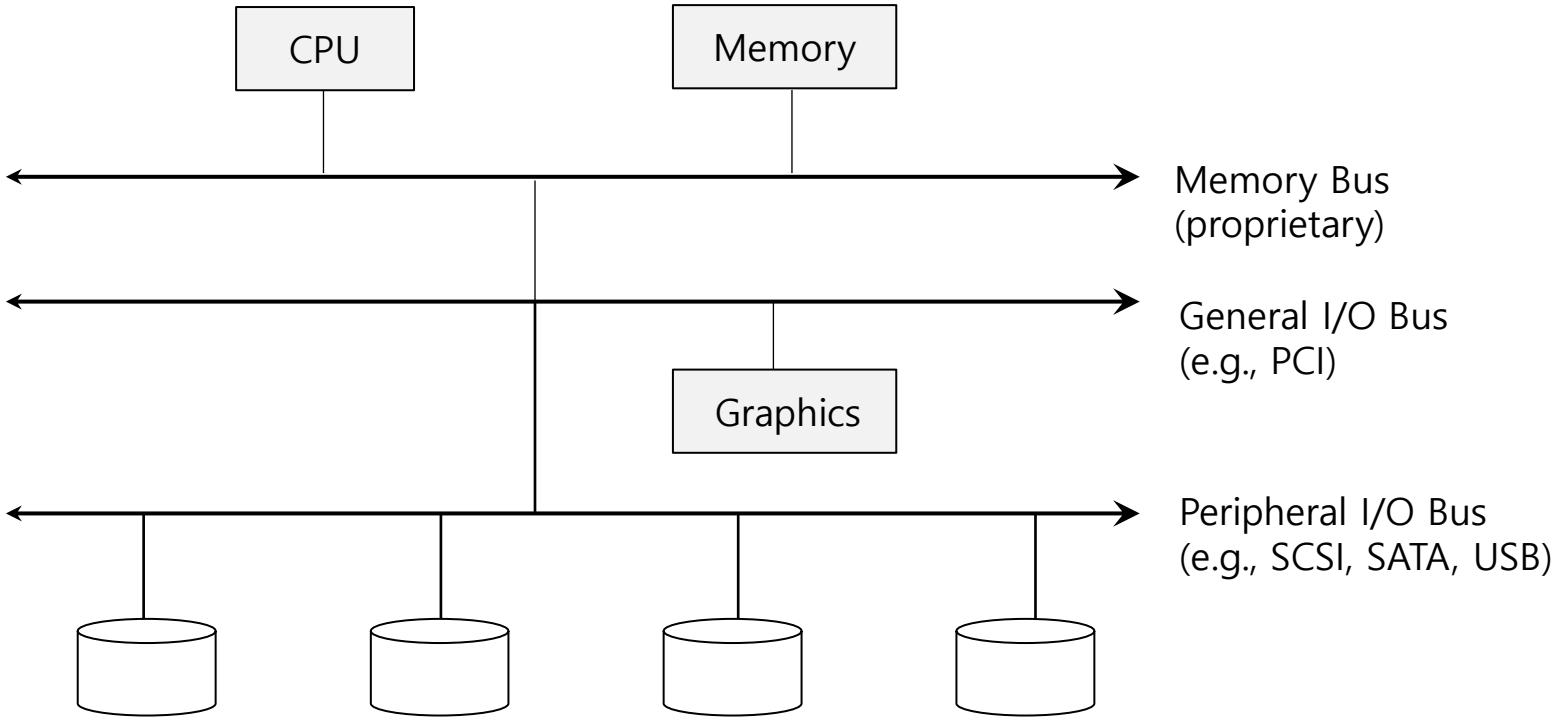
36. I/O Devices

Operating System: Three Easy Pieces

I/O Devices

- I/O is **critical** to computer system to **interact with systems.**
- Issue :
  - How should I/O be integrated into systems?
  - What are the general mechanisms?
  - How can we do this efficiently?

Structure of input/output (I/O) device



Prototypical System Architecture

CPU is attached to the main memory of the system via some kind of memory bus.

Some devices are connected to the system via a general I/O bus.

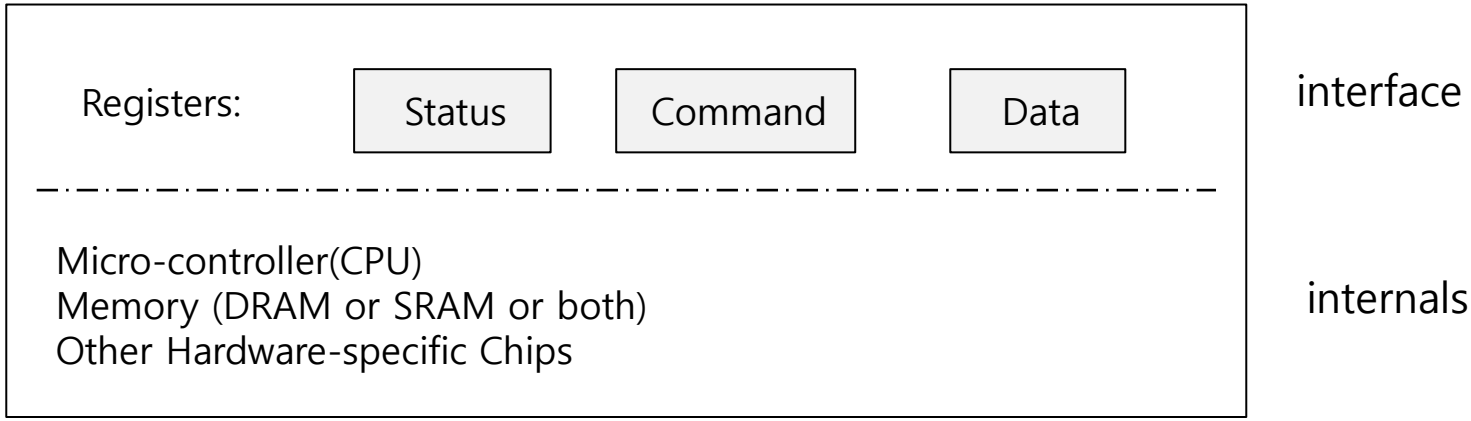
I/O Architecture

- Buses
  - Data paths that enable information transfer between CPU(s), RAM, and I/O devices.

- I/O bus
  - Data path that connects a CPU to an I/O device.
  - The I/O bus is connected to an I/O device by three hardware components: I/O ports, interfaces and device controllers.

Canonical Device

- A typical device has two important components.
  - **Hardware interface** allows the system software to control its operation.
  - **Internals** are implementation specific and do the work.



Canonical Device

Hardware interface of Canonical Device

- **status register**
  - The current status of the device
- **command register**
  - Tells the device to perform a certain task
- **data register**
  - Used to pass data to the device, or get data from the device

By reading and writing these three registers,
the operating system can control device behavior.

Hardware interface of Canonical Device (Cont.)

- Typical interaction example

```
while ( STATUS == BUSY)                    <-- polling
    ; //wait until device is not busy
write data to data register
write command to command register
    Doing so starts the device and executes the command
while ( STATUS == BUSY)                    <-- polling
    ; //wait until device is done with your request
```

Polling

- The operating system waits until the device is ready by **repeatedly** reading the status register (polling).
  - Its simple to do and works OK if the device is fast.
  - **However, it wastes CPU time waiting for the device**.
    - Switching to another ready process makes for better CPU utilization.

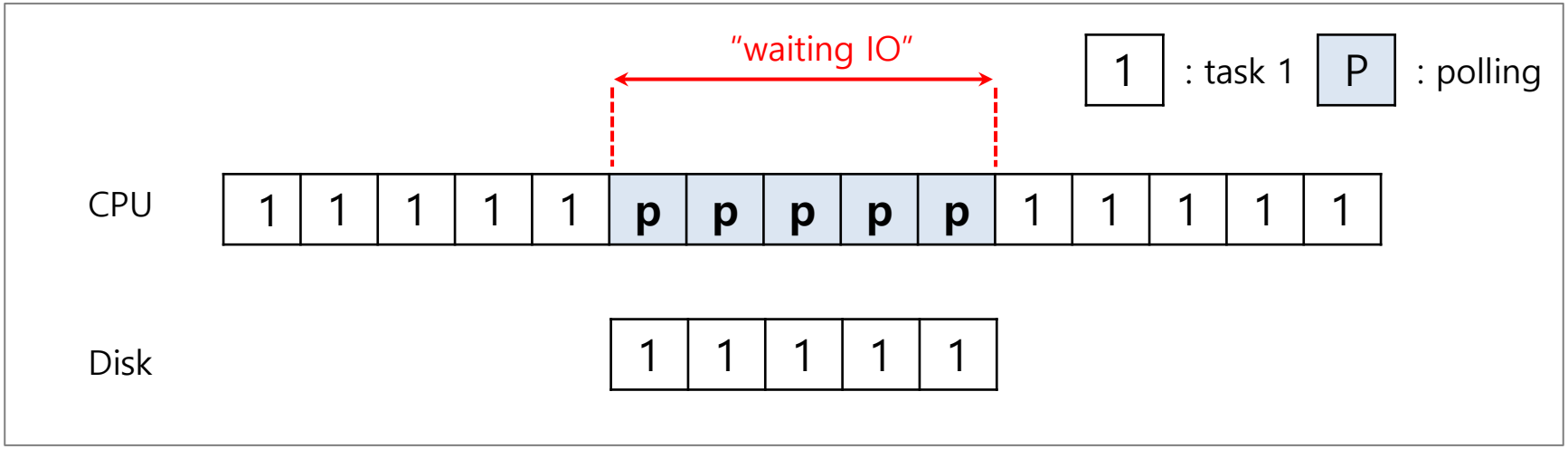


Diagram of CPU utilization by polling

interrupts

- **So lets put the I/O requesting process to sleep** and context switch to another process.
- When the I/O device is finished, wake the process waiting for the I/O by **interrupt**.
  - Ensures efficient utilization of both the **CPU and the disk.**

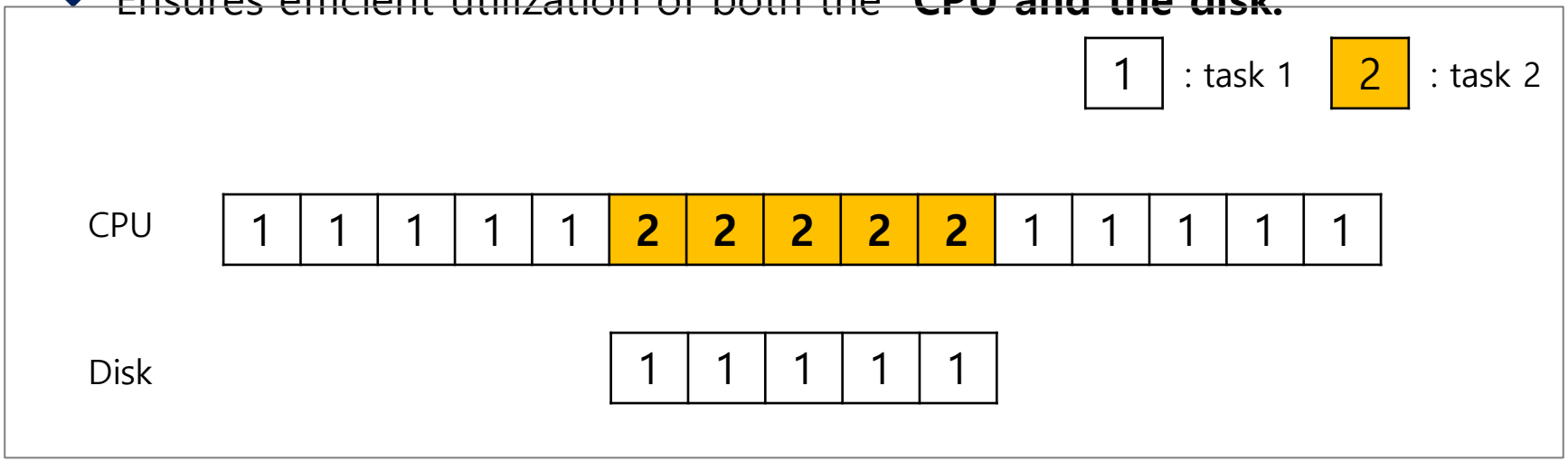


Diagram of CPU utilization by interrupt

Polling vs interrupts

- *However,* **"interrupts are not always the best solution"**
  - If, device performs very quickly, an interrupt will "slow down" the system.
  - Because **a context switch is expensive (switching to another process)**

If an I/O device is fast → polling is best.
If it is slow → interrupts are better.

CPU is once again over-burdened

- CPU **wastes a lot of time** to copy a *large chunk of data* from memory to the device.

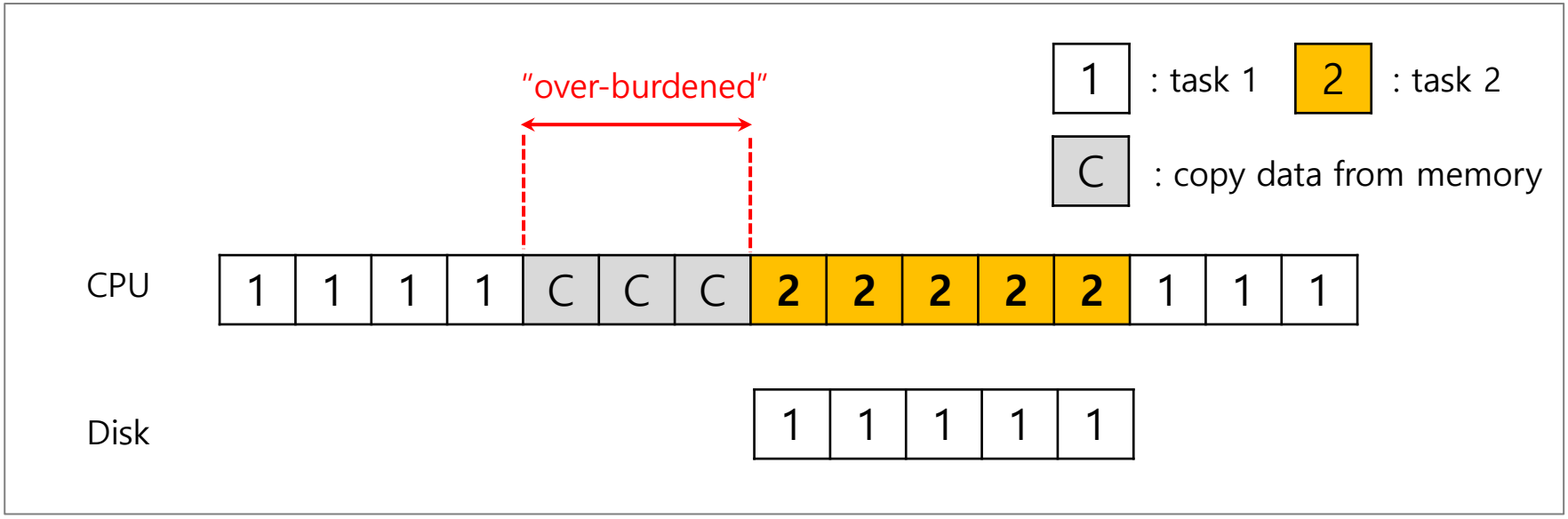


Diagram of CPU utilization
(programmed I/O- CPU involved in transferring memory)

DMA (Direct Memory Access)

- **Copy data** in memory by knowing
  - where the data lives in memory
  - how much data to copy
- When completed, DMA raises an interrupt, I/O begins on Disk.

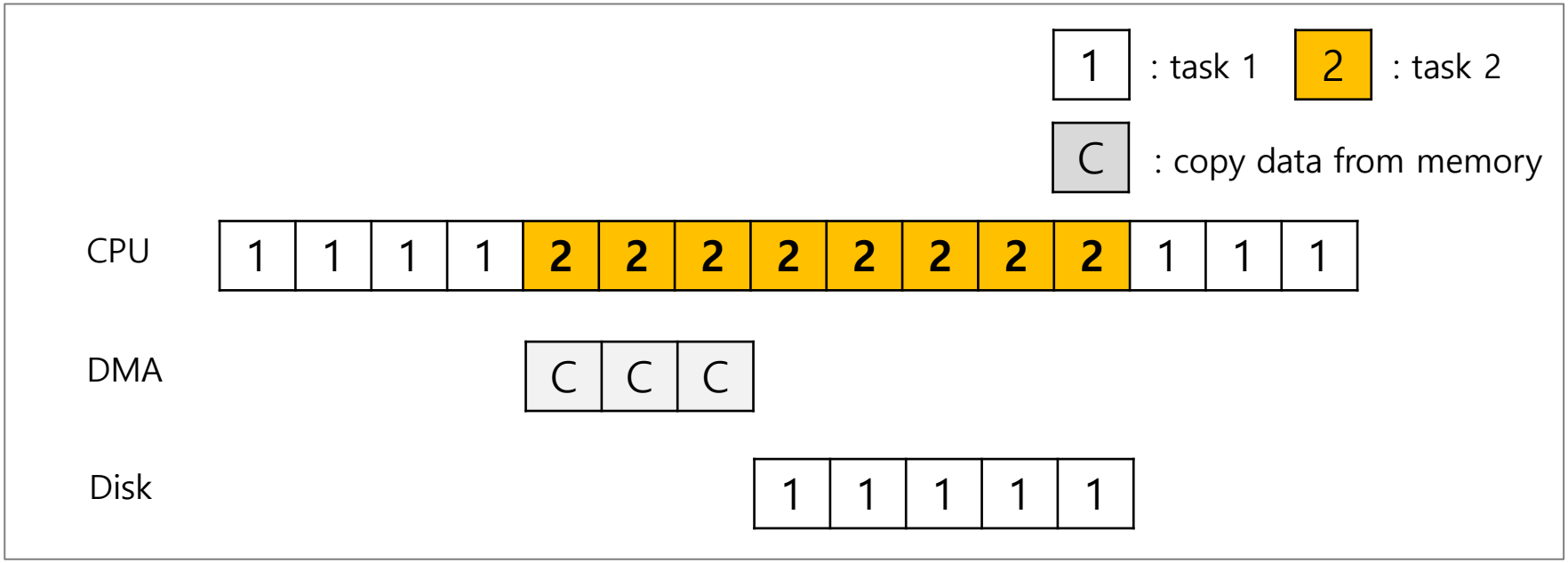


Diagram of CPU utilization by DMA

Device interaction

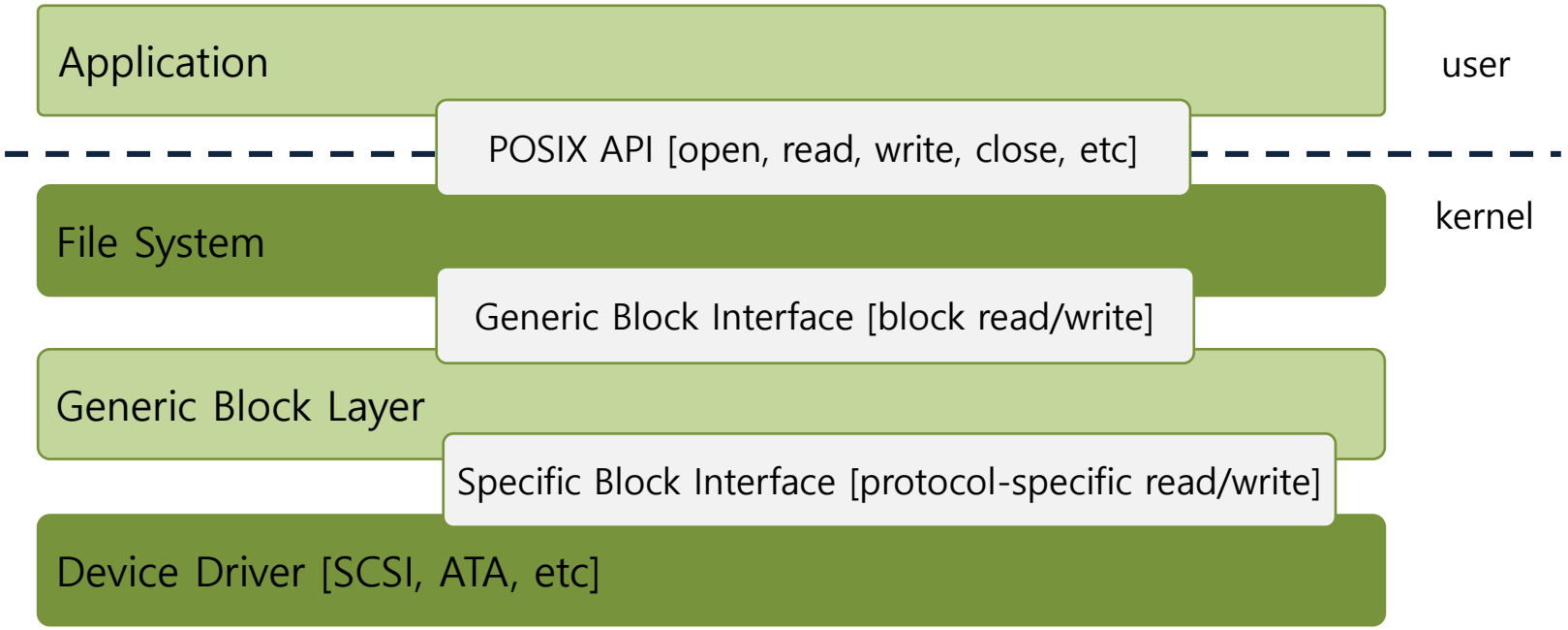
- Problem
  - How can the OS communicate with a **device**?
- Solutions
  - I/O instructions: a way for the OS to send data to specific device registers.
    - Ex) `in` and `out` instructions on x86
  - memory-mapped I/O
    - Device registers available as if they were memory locations.
    - The OS `load` (to read) or `store` (to write) to the device instead of main memory.

Device interaction (Cont.)

- How does the OS interact with **different specific interfaces**?
  - Ex) We'd like to build a file system that worked on top of SCSI disks, IDE disks, USB keychain drivers, and so on.
- Solutions: Abstraction
  - Abstraction encapsulate any specifics of device interaction.

File system Abstraction

- File System unaware of which disk class it is using.
  - Ex) It issues **block read** and **write** request to the generic block layer.



The File System Stack

Problems of File system Abstraction

- If a device has special capabilities, these capabilities will go unused in the generic interface layer.
  - Ex. SCSI disk have rich error reporting but IDE disks do not. Program to lowest common denominator so OS cannot see rich SCSI errors.
- BTW, Over 70% of OS code is found in device drivers.
  - Device drivers are specialty programs that are written to communicate with specific (or general) devices. They are needed because different devices have different protocols (ex. SCSI verses IDE). Drivers are middleware that handle communication between the OS (via Generic Block Layer) and any device you plug into your system.
  - Sometimes written by non-experts. So they are a primary contributor to **kernel crashes**.