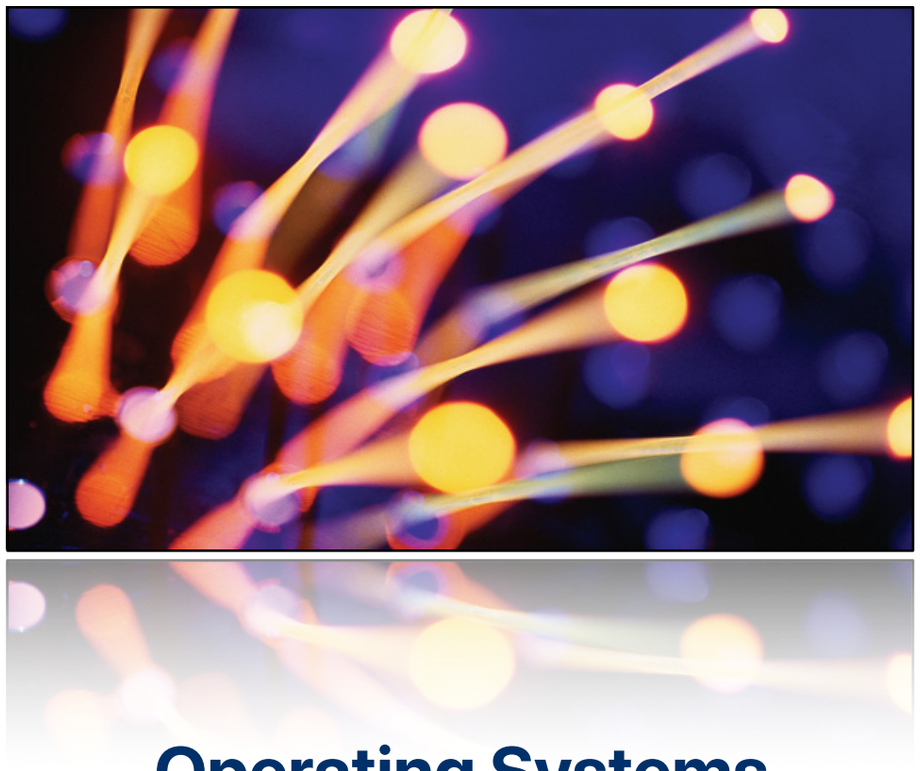
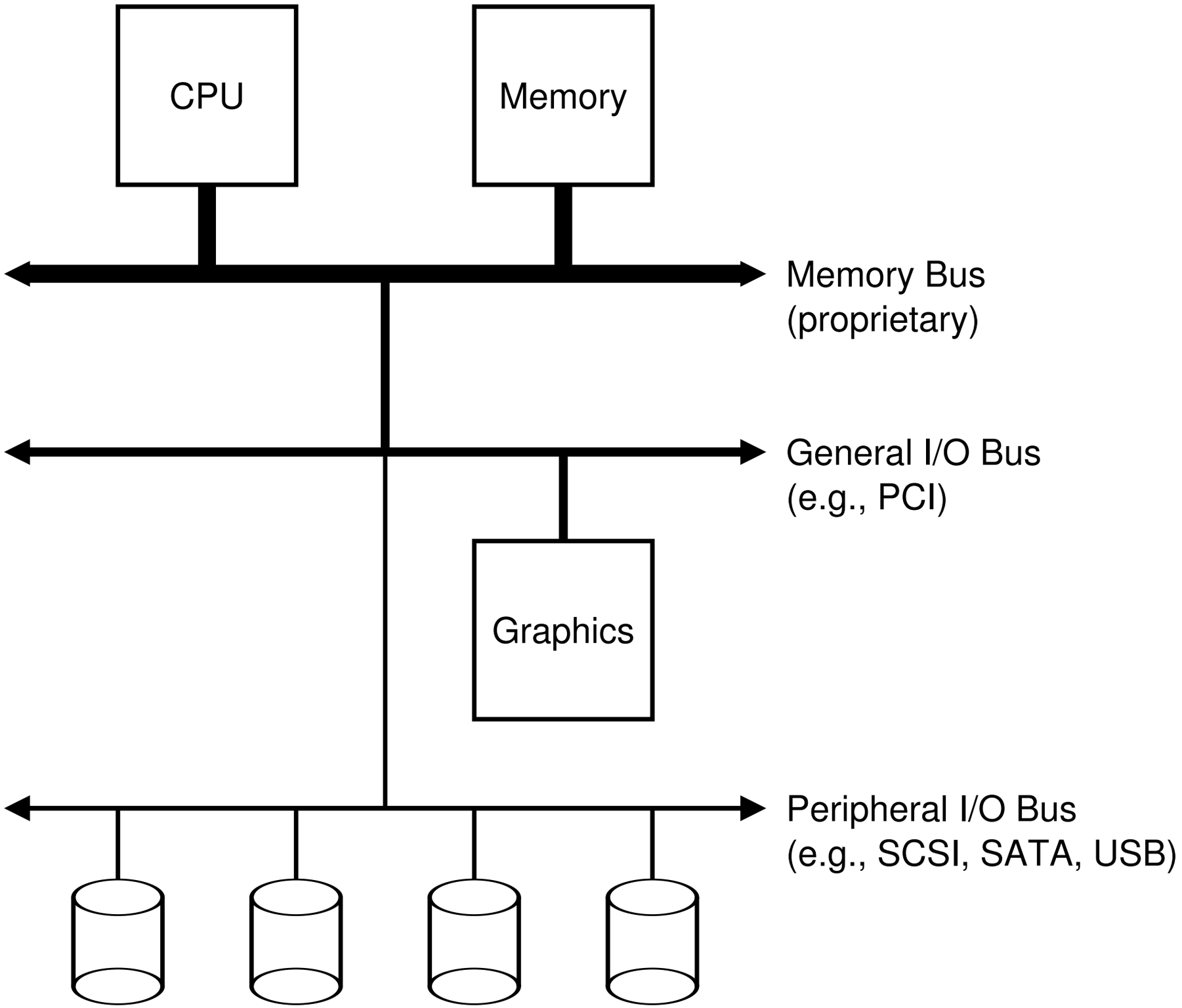
Input/Output Devices



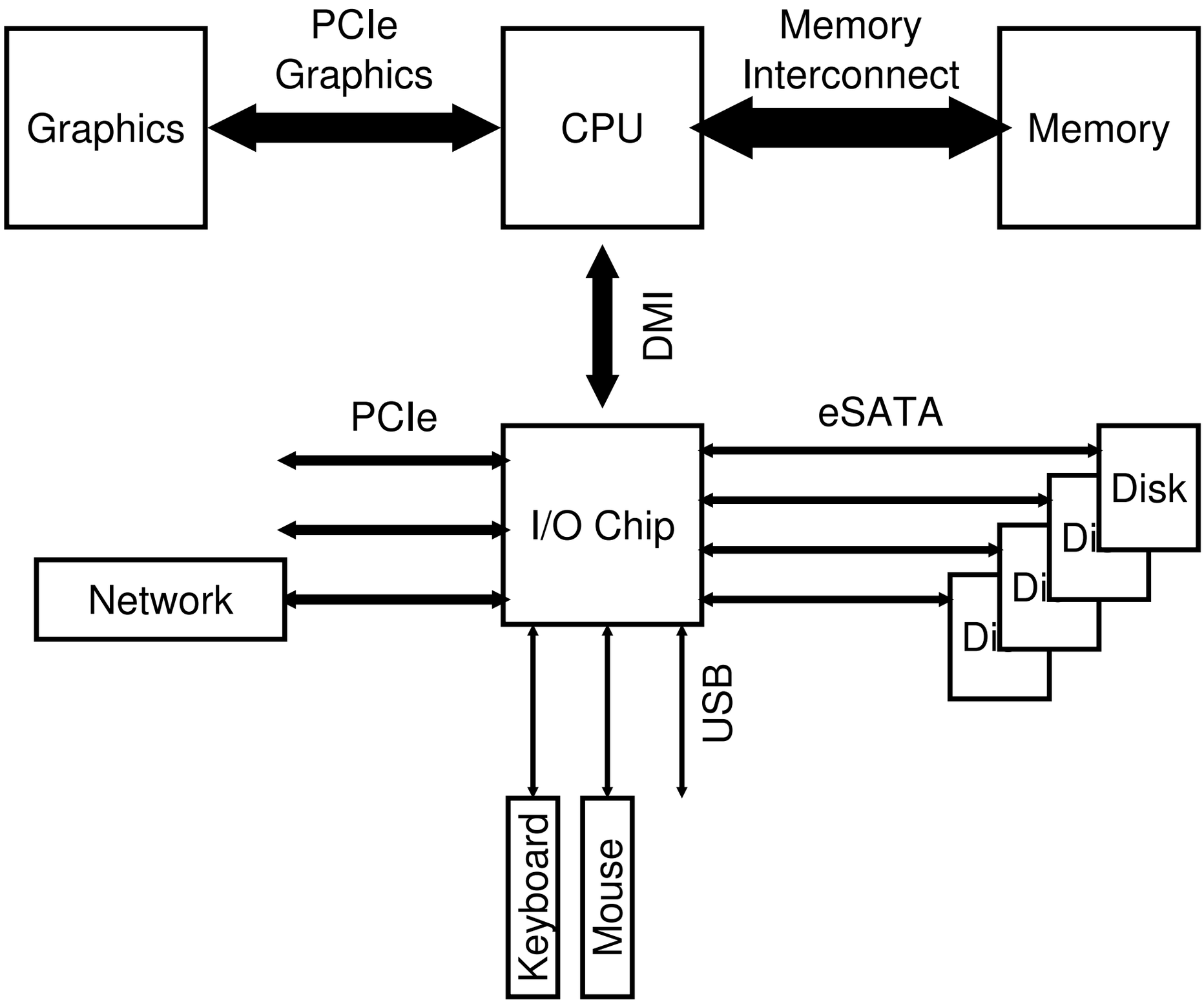
Operating Systems

Baochun Li
University of Toronto

Traditional I/O: hierarchical architecture



Modern I/O system architecture



A canonical I/O device

Device — actual hardware

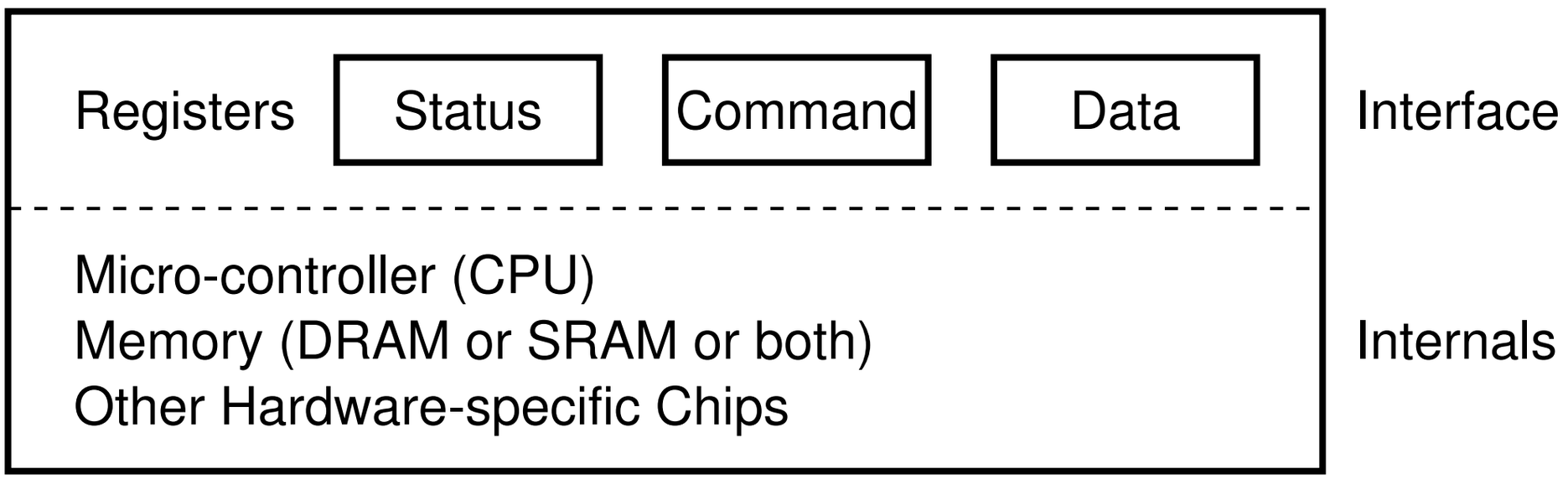
includes a hardware interface

example: status, command and data registers

serves as an interface between CPU and device

Device driver — software in the OS kernel

device-specific code for controlling each I/O device



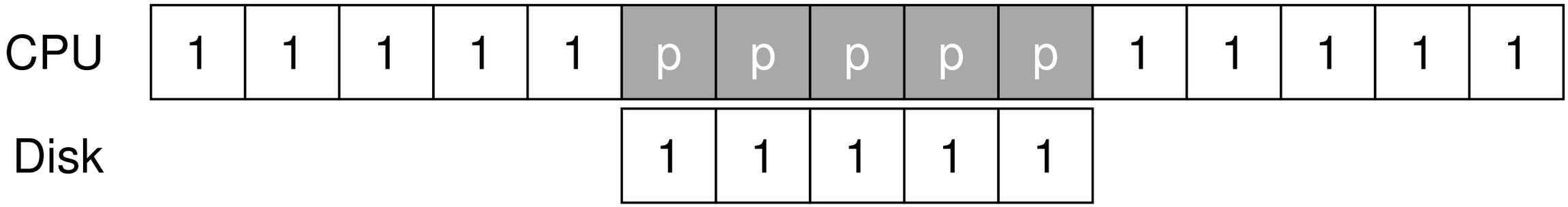
The Canonical Protocol: Polling

CPU is doing the work by polling the device: "Programmed I/O"

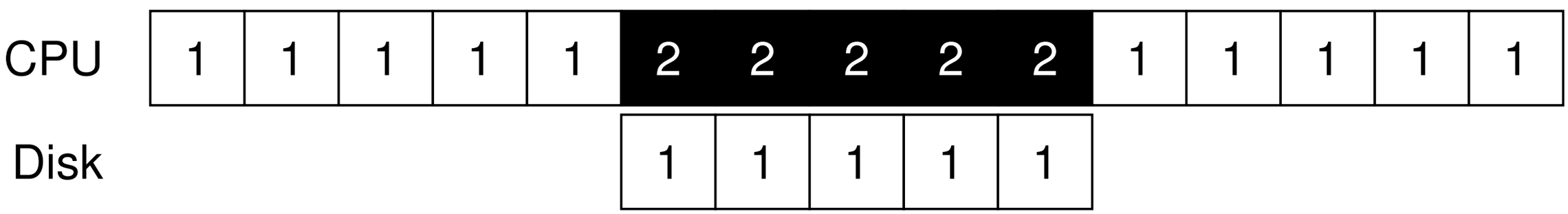
```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (Doing so starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

Disadvantages with Polling

Polling is inefficient: wasting CPU time waiting for a slow device to complete its work



Solution: Interrupts



Interrupts revisited

Benefits

Allows for overlap of computation and I/O

Improves CPU utilization

Problem 1: Livelock

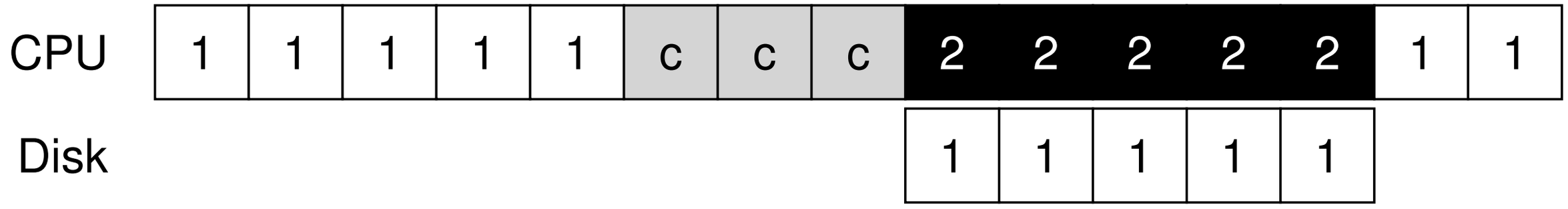
Example: when a huge number of packets arrive at a web server, each generating an interrupt, the OS may be servicing interrupts at all times, never allowing a user-level process to run

Problem 2: slows down the system if a device is very fast

Perhaps try a hybrid approach: poll for a while, and if not done, use interrupts

More efficient data movement with DMA

Even with interrupts, the CPU still needs to copy the data from memory to the device explicitly, one word at a time

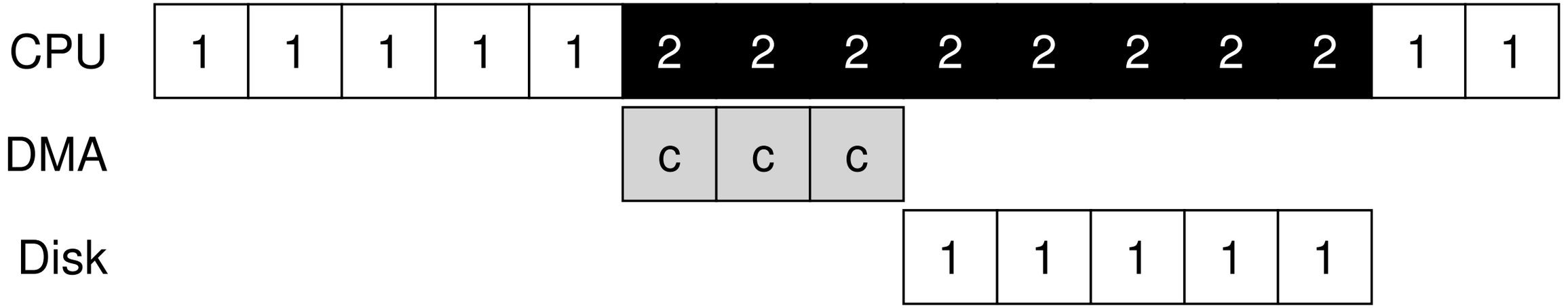


How do we offload the work and allow the CPU to be more effectively utilized?

Direct Memory Access (DMA)

The OS programs the DMA, a special device, telling it where the data lives, how many bytes to copy, and which device to send it to

When the DMA is complete, the DMA controller raises an interrupt



Direct Memory Access (DMA): Recap

DMA helps data to be transferred from device straight to/from memory

CPU is not involved

With the DMA controller —

DMA does the work of moving the data

CPU sets up the DMA controller (“programs it”)

CPU continues

The DMA controller moves the bytes

How does the OS communicate with the device?

I/O Instructions

in and **out** instructions on x86: privileged instructions

use a **port** to name the device

Memory-mapped I/O

Hardware makes device registers available as memory locations

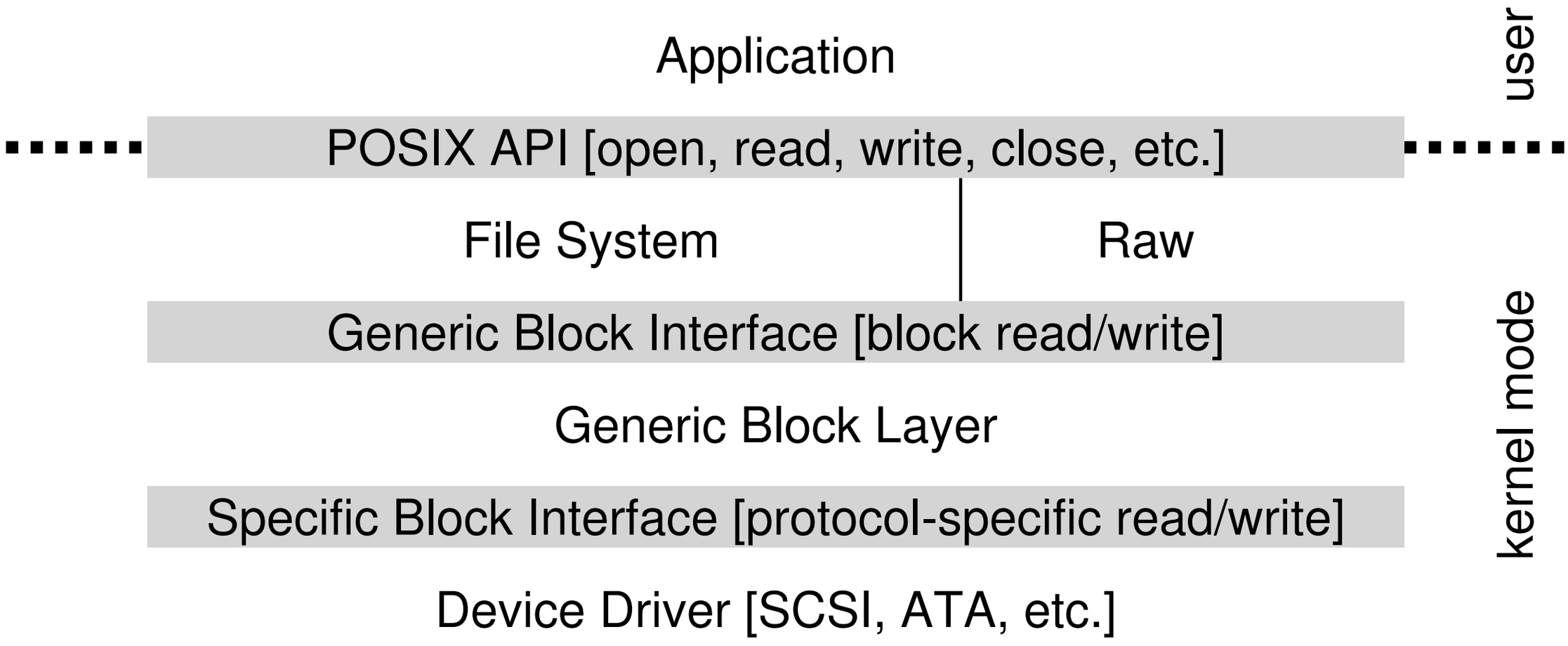
Just use the **load** and **store** instructions to read and write the address

No new instructions needed

How to keep the OS device-neutral?

— Use abstraction!

The file system stack in Linux



70% of the code in Linux is found in device drivers — the main source of kernel crashes

Disadvantages of using abstraction

Devices with specific capabilities still need to present a generic interface, and specific capabilities get unused

- Richer error reporting in SCSI than ATA/IDE
- Only a generic IO error will be presented to the file system

Block Devices

Block devices

- Stores information in fixed-size blocks, each with its own address
- Possible to read or write each block independently
- Use read(), write(), lseek() system calls to access

Raw I/O access can also be allowed

- raw I/O may be needed for file system checkers, disk defragmenters, and database systems

Character Devices

Delivers or accepts a stream of characters — like pipes

Unbuffered, direct access

Examples: serial ports, parallel ports, audio devices

Not addressable and does not have any seek operation

Both block and character devices can be accessed by using the device file in Linux and macOS, in **/dev**

Note: device files in **/dev** do not have to correspond to actual physical devices (e.g., /dev/null, /dev/random)

What we've covered so far

Three Easy Pieces: Chapter 36.1 – 36.7 (I/O Devices)