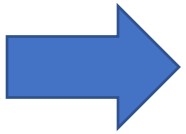
Explanation & Discussion for Project 1.C

Recap: Deadlock

Necessary Conditions:

- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait



Deadlock Prevention:

- Remove Explicit Mutual Exclusion
- Use one resource at a time; or, request all resources at a time
- Voluntarily give up assigned resource
- Order resources

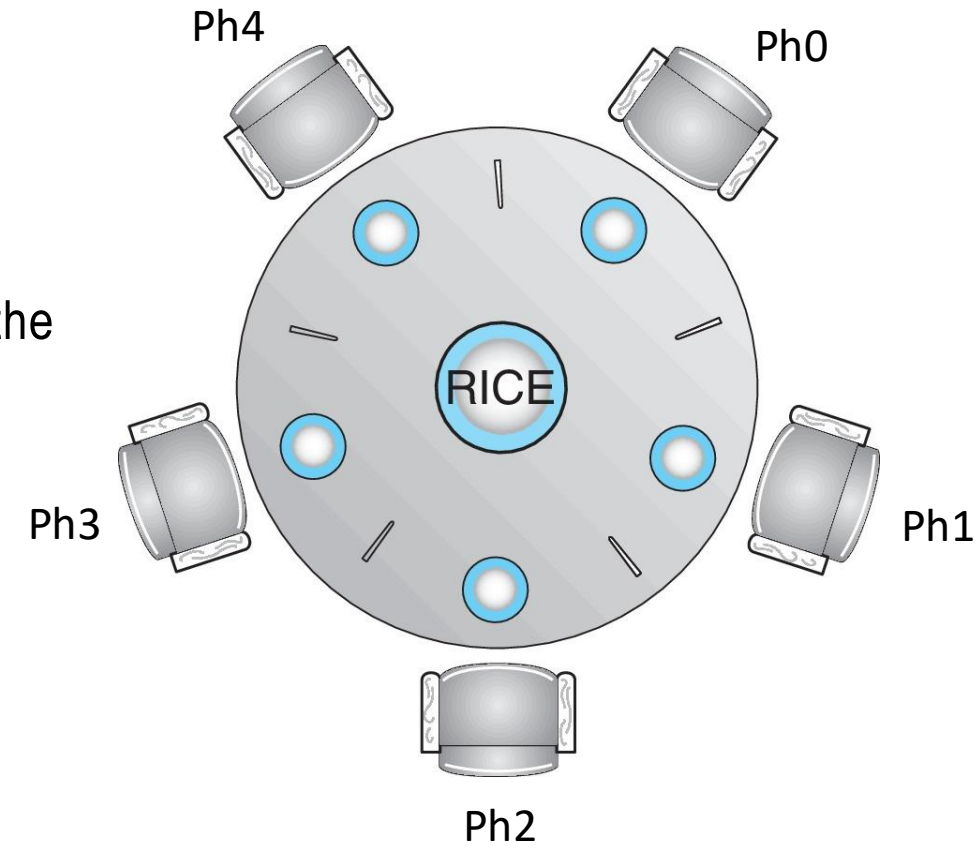
Classic Problem: Dinning Philosophers

5 philosophers alternate between thinking and eating

There are 5 chopsticks placed between the philosophers

To eat, a philosopher must pick up two chopsticks (the ones to the right and left of them), to think they put down both chopsticks

```
void * thread_start(void *arg) {
  // ph is the philosopher's id (0 to 4)
  int *ph = (int *)arg;
  while(1) {
    think();
    get_chopsticks(ph);
    eat();
    put_chopsticks(ph);
  }
}
```



A Broken Solution

How to implement get() and put() functions using semaphores?

Each chopstick is a type of resources, create a semaphore for each chopstick initialized to 1

A philosopher must sem_wait (decrement) a chopstick semaphore to pick it up and sem_post (increment) a chopstick semaphore to put it back

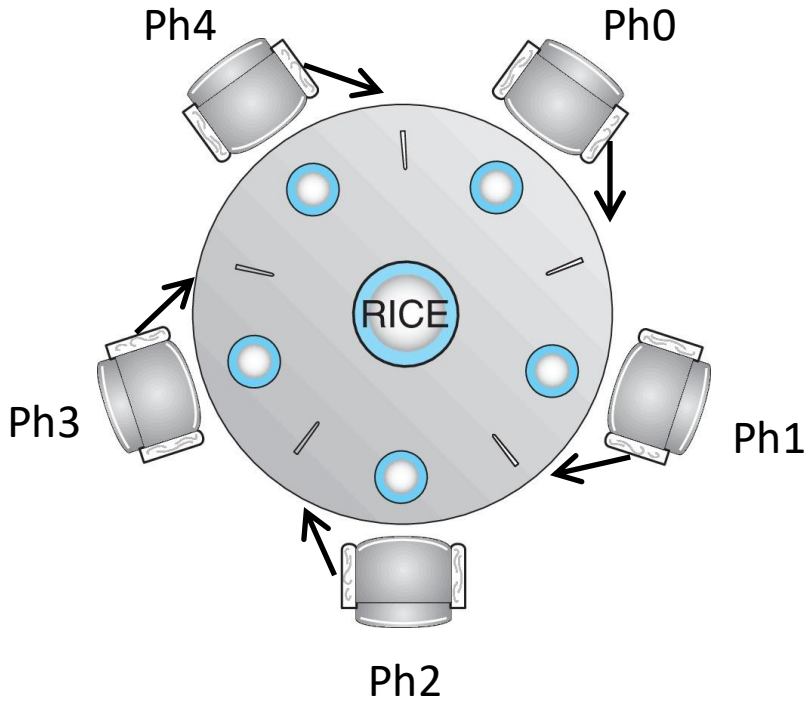
```
sem_t chopsticks[5];

void init_chopsticks() {
  for (int i=0; i<5; i++) {
    sem_init(&chopsticks[i], pshared, 1);
  }
}

void get_chopsticks(int ph) {
  sem_wait(&chopsticks[left(ph)]);
  sem_wait(&chopsticks[right(ph)]);
}

void put_chopsticks(int ph) {
  sem_post(&chopsticks[left(ph)]);
  sem_post(&chopsticks[right(ph)]);
}
```

The Problem - Deadlock

What is wrong with the solution? Consider the following execution:

Ph0 takes left chopstick
Context switch to Ph1
Ph1 takes left chopstick
Context switch to Ph2
Ph2 takes left chopstick
Context switch to Ph3
Ph3 takes left chopstick
Context switch to Ph4
Ph4 takes left chopstick
Now what? All threads are waiting for right chopstick, the system is stuck, forever!

When a group of threads are each waiting on another forming a cycle a **deadlock** has formed



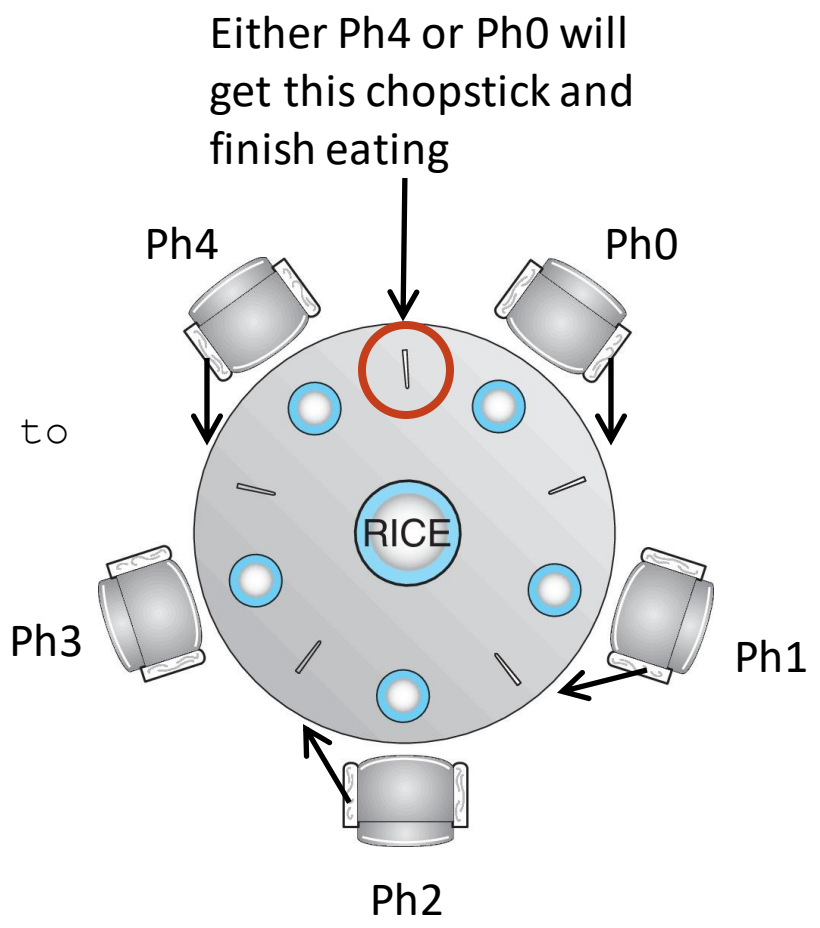

```
// assume chopsticks[] is an array of 5 semaphores initialized to 1

void get_chopsticks(int ph) {
  sem_wait(&chopsticks[left(ph)]);
  sem_wait(&chopsticks[right(ph)]);
}

void put_chopsticks(int ph) {
  sem_post(&chopsticks[left(ph)]);
  sem_post(&chopsticks[right(ph)]);
}
```

One Solution

Consider a simple rule: Ph4 must pick up the right chopstick before the left

No longer possible for cycle to form, therefore deadlock is not possible

```
// assume chopsticks[] is an array of 5 semaphores initialized to

void get_chopsticks(int ph) {
  if (ph == 4) {
    sem_wait(&chopsticks[right(ph)]);
    sem_wait(&chopsticks[left(ph)]);
  } else {
    sem_wait(&chopsticks[left(ph)]);
    sem_wait(&chopsticks[right(ph)]);
  }
}
```

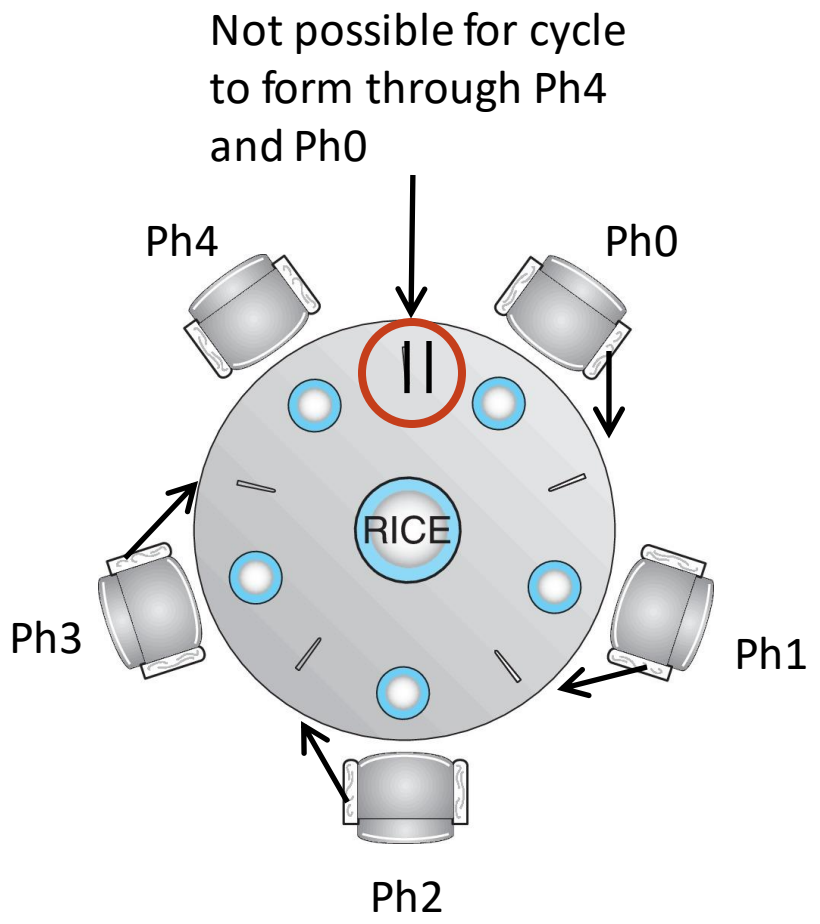


Slightly Different Problem

Consider a second chopstick placed between Ph4 and Ph0

Is deadlock possible?

```
sem_t chopsticks[5];

void init_chopsticks() {
  sem_init(&chopsticks[0], pshared, 2);
  for (int i=1; i<5; i++) {
    sem_init(&chopsticks[i], pshared, 1);
  }
}
```



I/O Devices

(based on Ch. 36)

Slides revised from
Matthew Tancreti
Iowa State University

I/O Devices

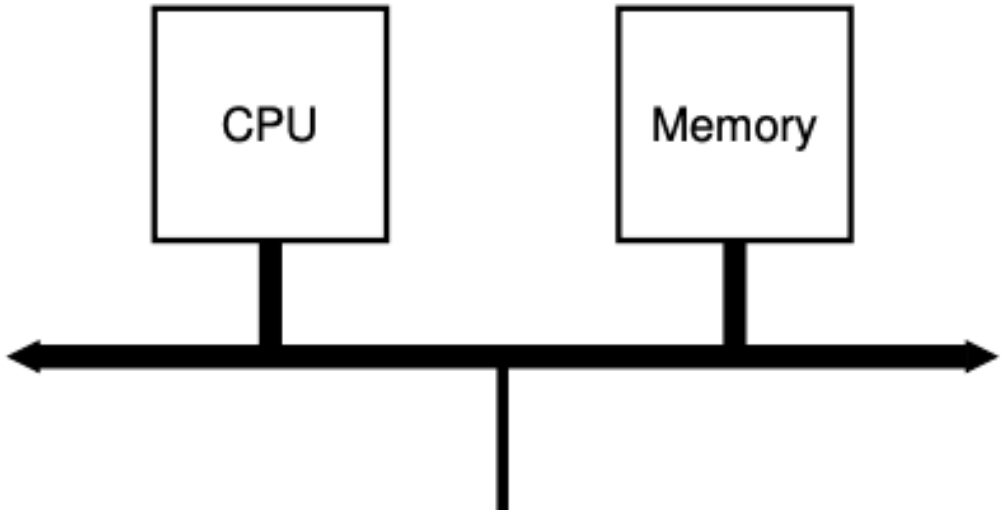
Computer are not much use without Input and Output

I/O is tremendously slower that the CPU and our goal is to keep the CPU busy

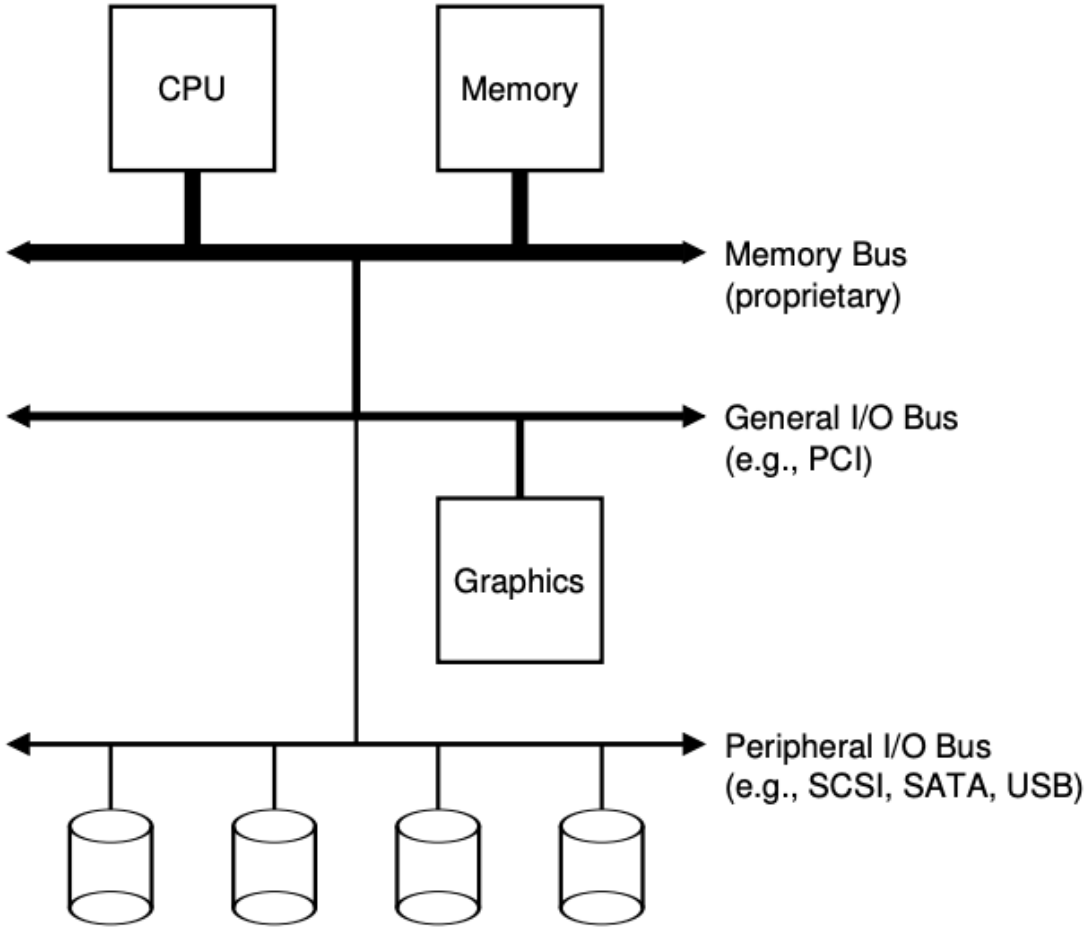
We have seen multiprogramming as an example of how to achieve this

How should computers and OSes be organized to perform I/O efficiently?

Main System Bus

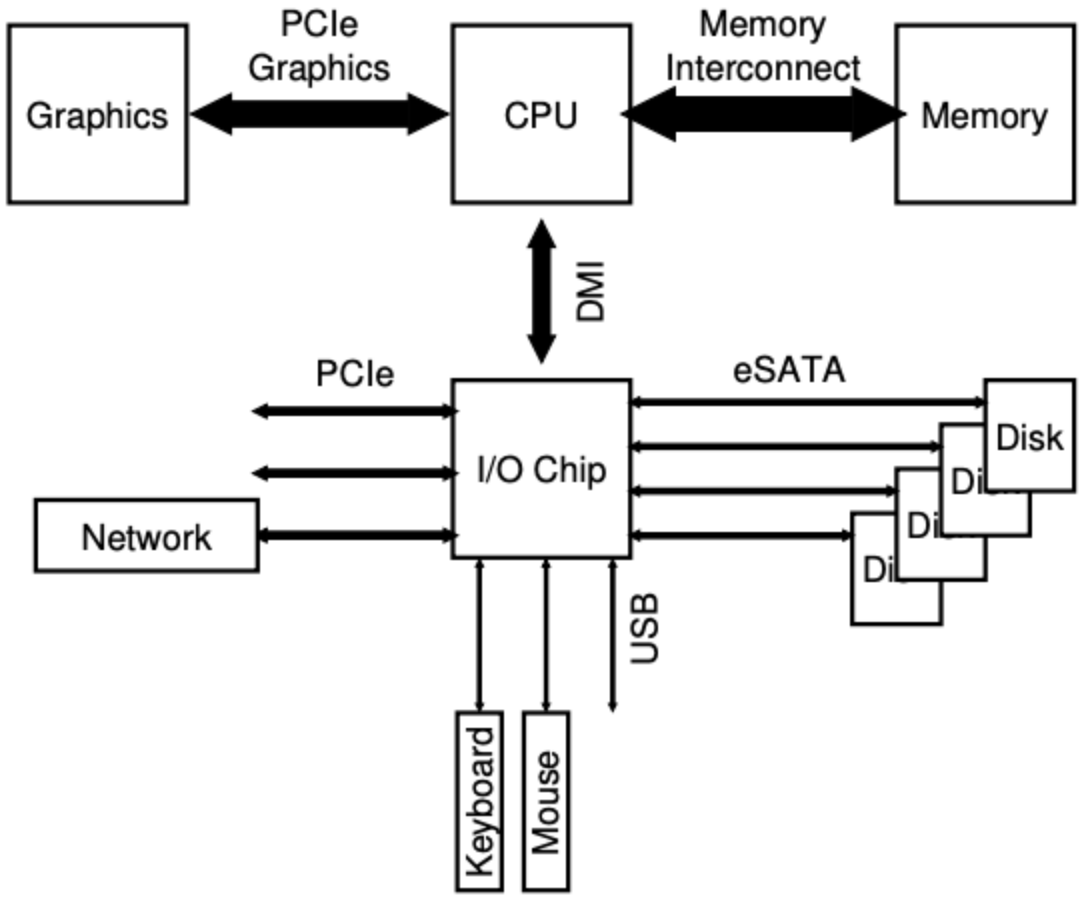


Bus Hierarchy: Prototypical System Architecture



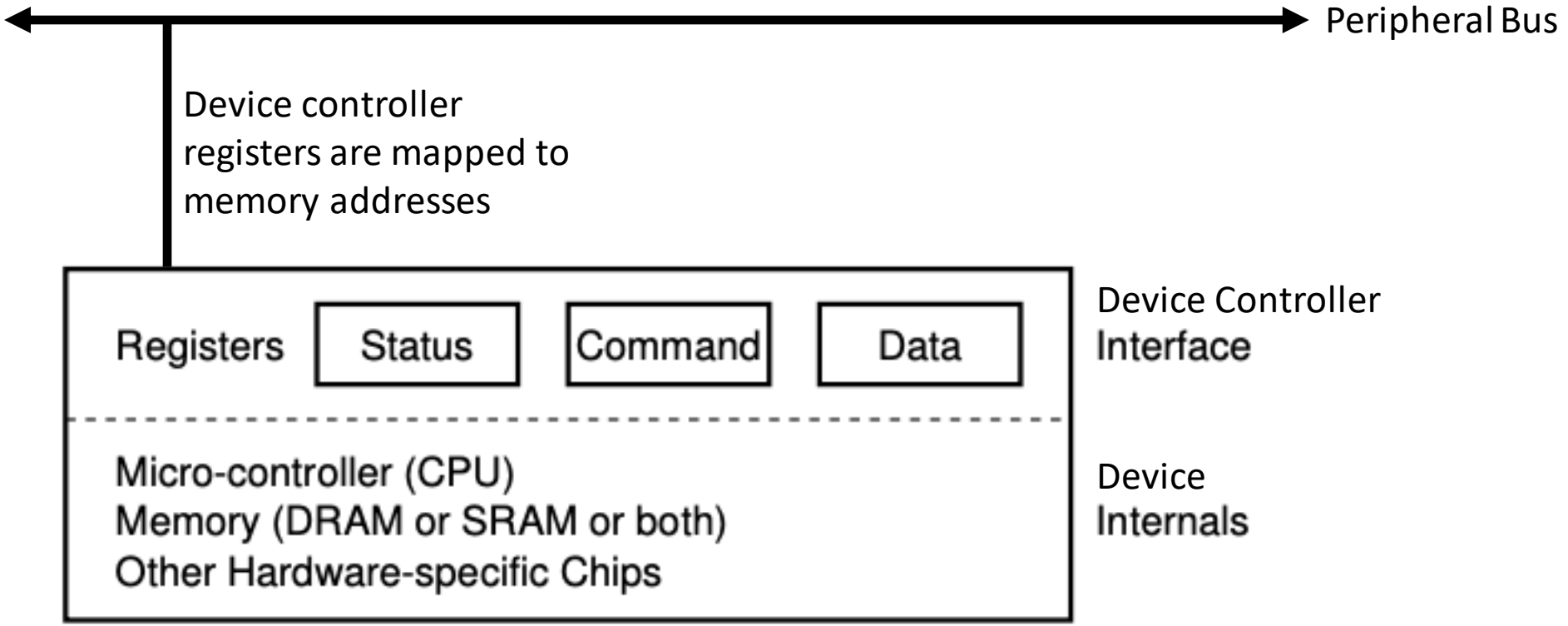
PCI: Peripheral Component Interconnect
SCSI: Small Computer System Interface
SATA: Serial Advanced Technology Attachment
USB: Universal Serial Bus

Modern System Architecture



DMI: Direct Media Interface (Intel)

Hardware Device



How OS interact with device: Polling (programmed I/O)

While (Status == Busy); //wait until device is not busy

Write data to Data register

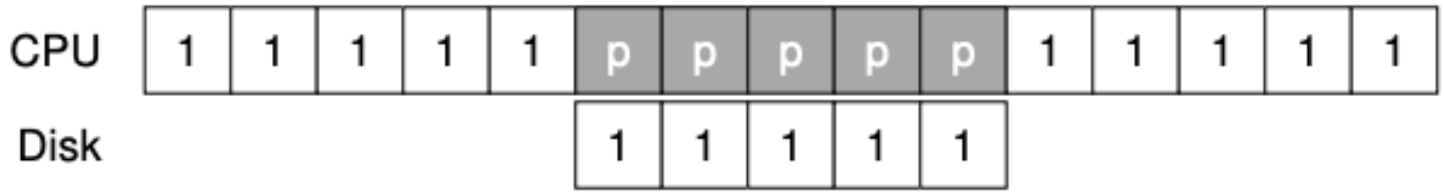
Write command to Command register

//start the device and execute command

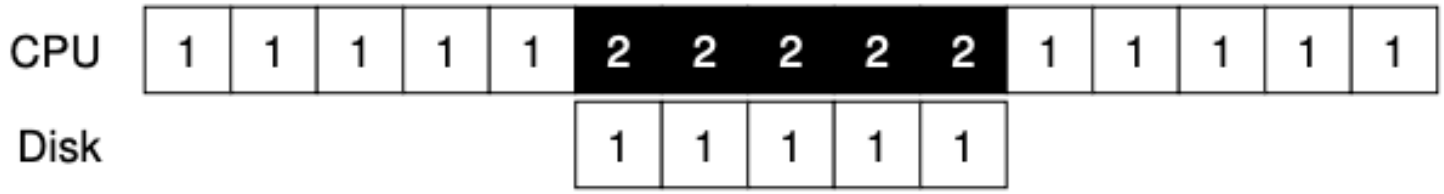
While (Status == Busy); //wait until device is done with the request

How OS interacts with device: Interrupts

Without Interrupts – P means CPU is in polling (spinning) loop

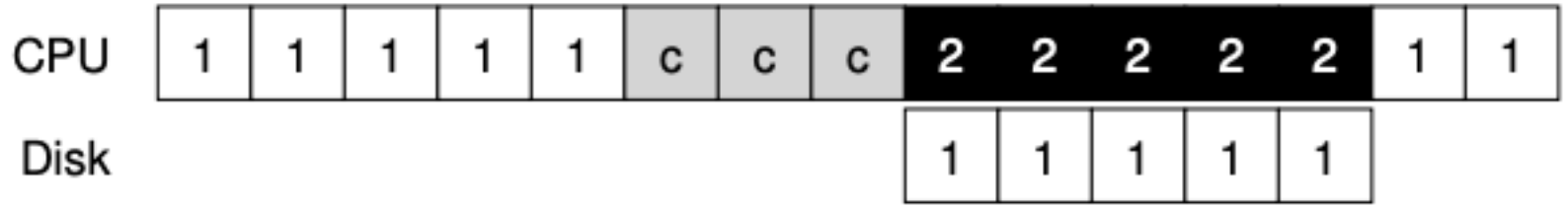


With Interrupts – CPU can be switched to running process 2 during I/O

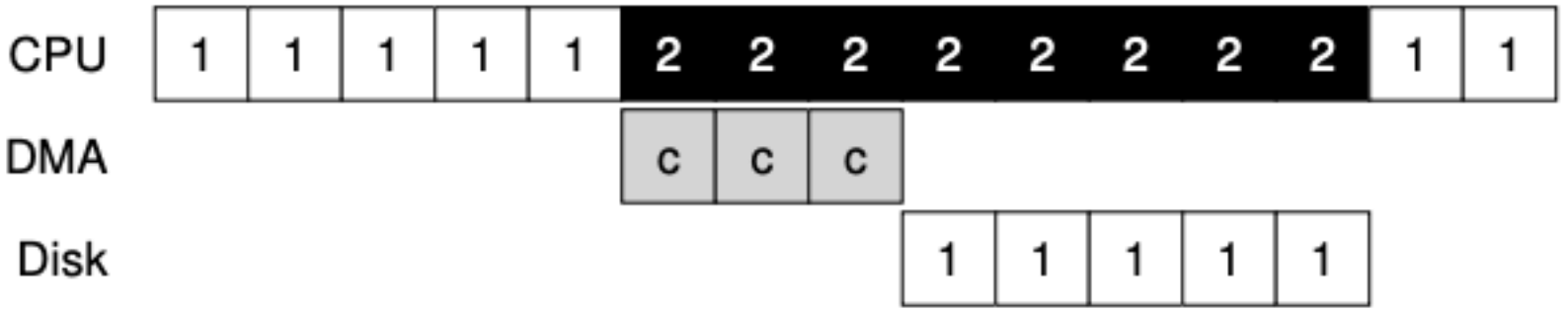


How OS interacts with device: Direct Memory Access (DMA)

Without DMA – C means CPU is copying



With DMA



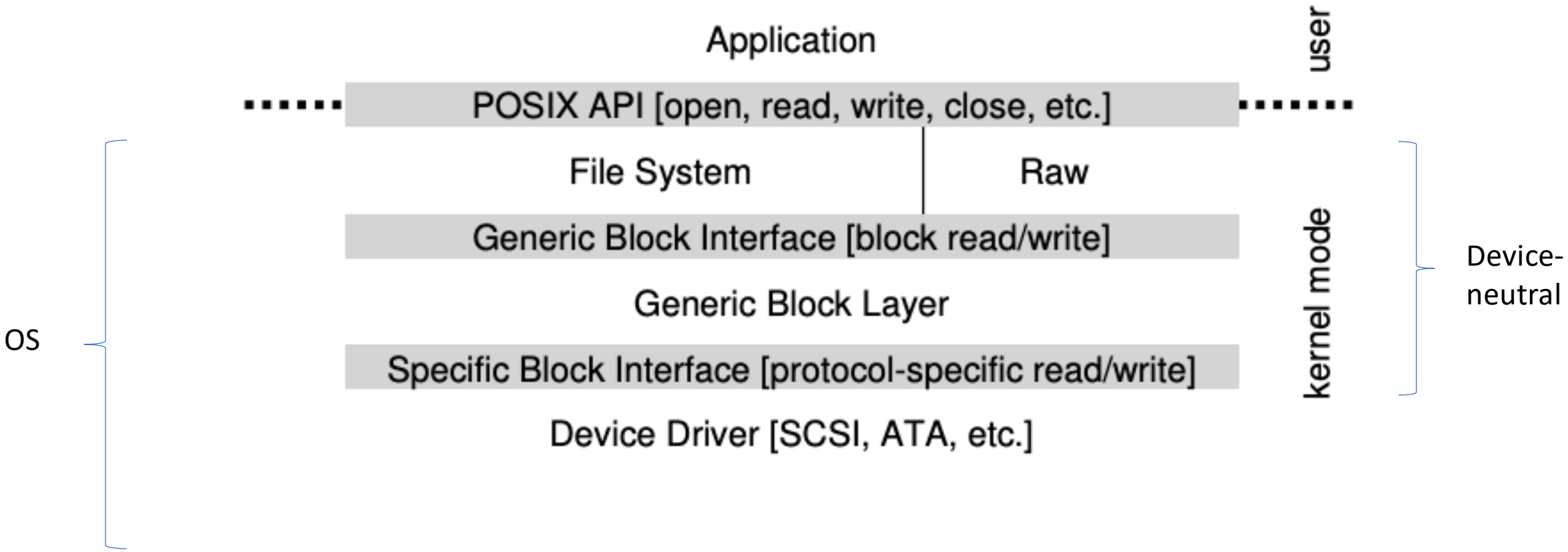
Methods of Device Interaction

Two methods:

- Explicit privileged I/O instructions, e.g., in and out (x86)

- Read/Load instructions to specific addresses (representing device registers), known as memory-mapped I/O.

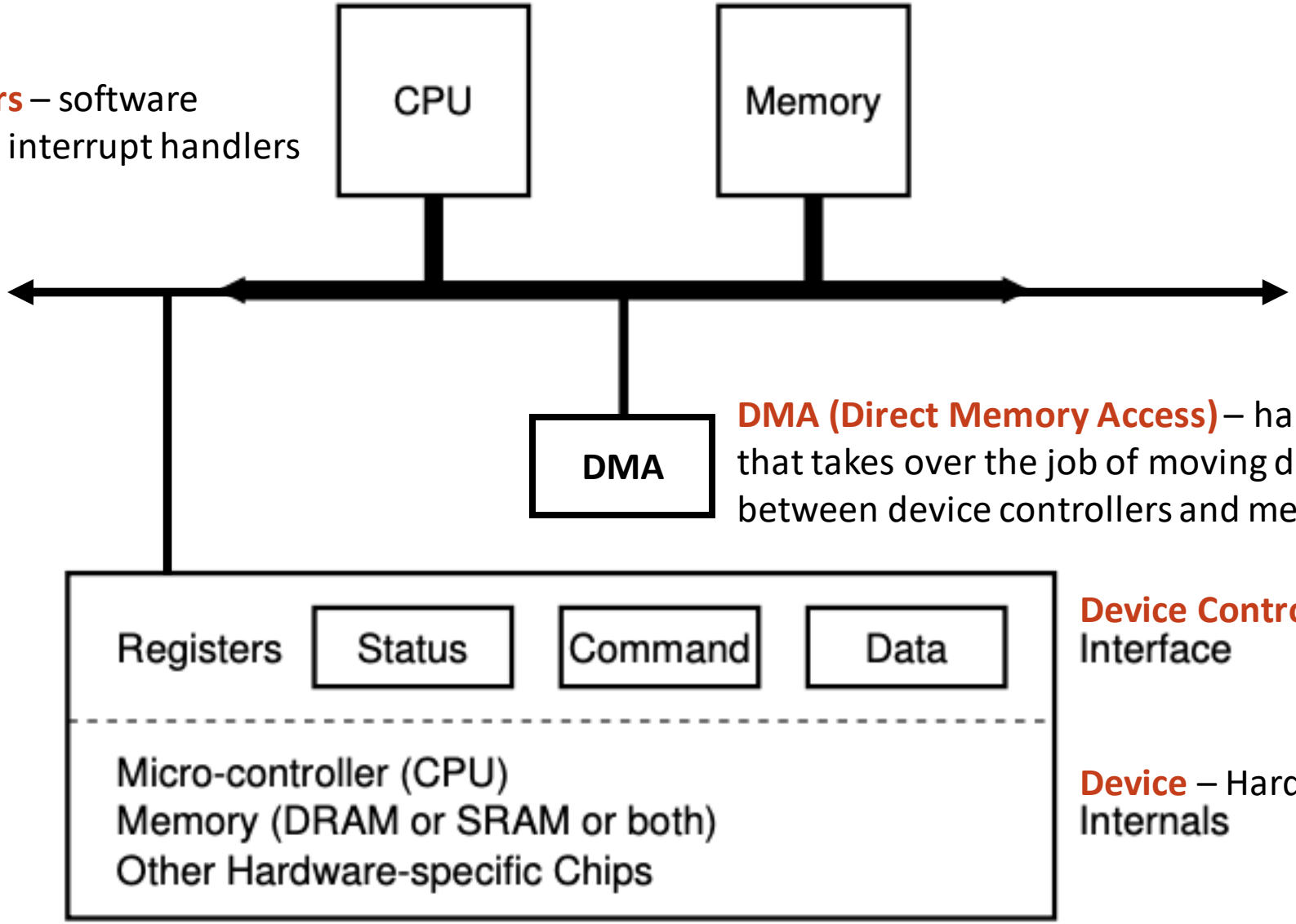
Fitting Into OS: Device Drivers & Abstraction Levels



Putting it Together

Application

Device Drivers – software connected to interrupt handlers in the OS



Bus(es) – set of wires that transfer addresses and data between any two connected hardware

DMA (Direct Memory Access) – hardware that takes over the job of moving data between device controllers and memory

Device Controller – Hardware/Firmware Interface

Device – Hardware/Firmware Internals