COS 318: Operating Systems

I/O Device and Drivers

Kai Li
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)

Announcements

- Project 3
  - Precept will be tonight in CS 105 7:30-8:20pm
  - Design review Monday October 22 (6 days from now)
  - Project due November 4 at 11:59pm (20 days from now)
- Lab TAs for project 3
  - Michael Franklin, Fishbowl Sat 3-5pm, Sun 3-5pm
  - Ilias Glechaskiel, Fishbowl Sat 3-5pm, Sun 8-10pm

Topics

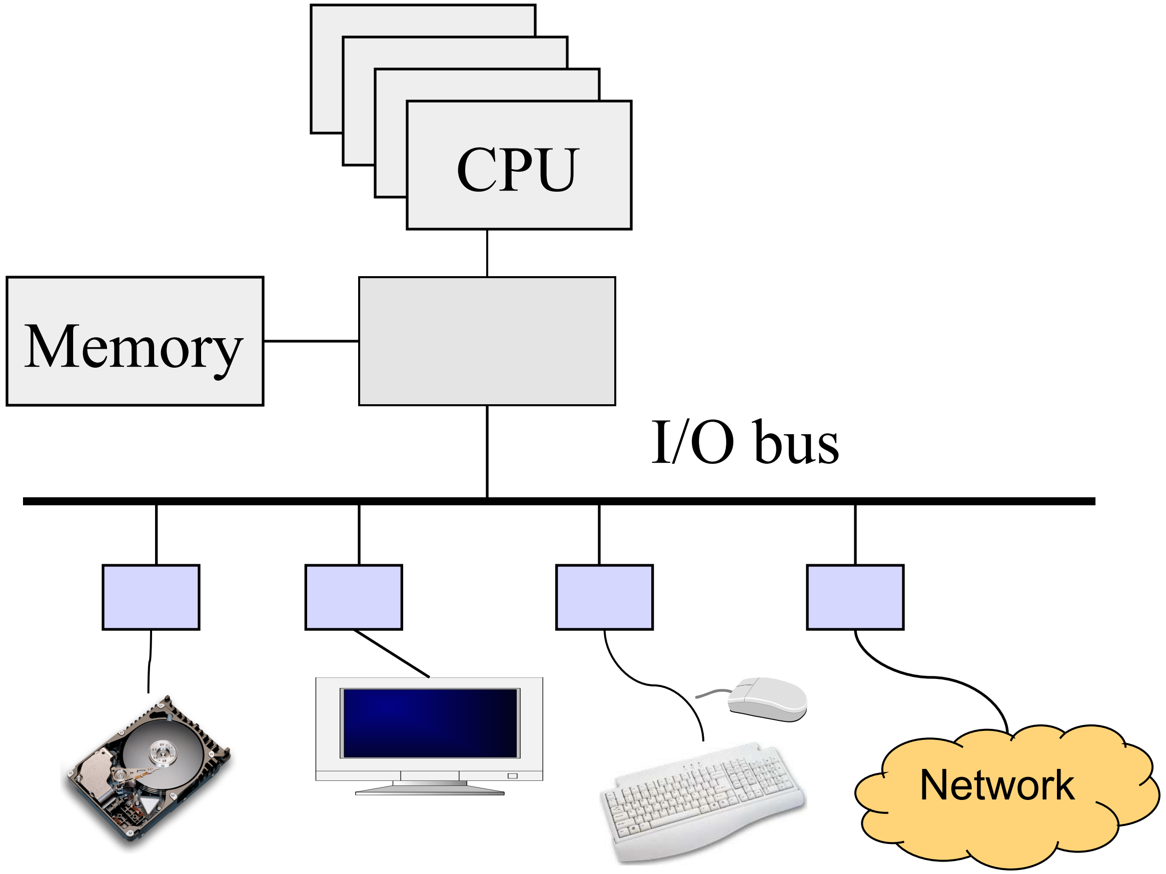
- I/O devices
- Device drivers
- Synchronous and asynchronous I/O

Input and Output

- A computer's job is to process data
  - Computation (CPU, cache, and memory)
  - **Move data into and out of a system** (between I/O devices and memory)
- Challenges with I/O devices
  - Different categories: storage, networking, displays, etc.
  - Large number of device drivers to support
  - Device drivers run in kernel mode and can crash systems
- Goals of the OS
  - Provide a generic, consistent, convenient and reliable way to access I/O devices
  - Achieve potential I/O performance in a system

Revisit Hardware

- Compute hardware
  - CPU and caches
  - Chipset
  - Memory
- I/O Hardware
  - I/O bus or interconnect
  - I/O controller or adaptor
  - I/O device
- Two types of I/O devices
  - Programmed I/O (PIO)
  - Direct Memory Access (DMA)



Definitions and General Method

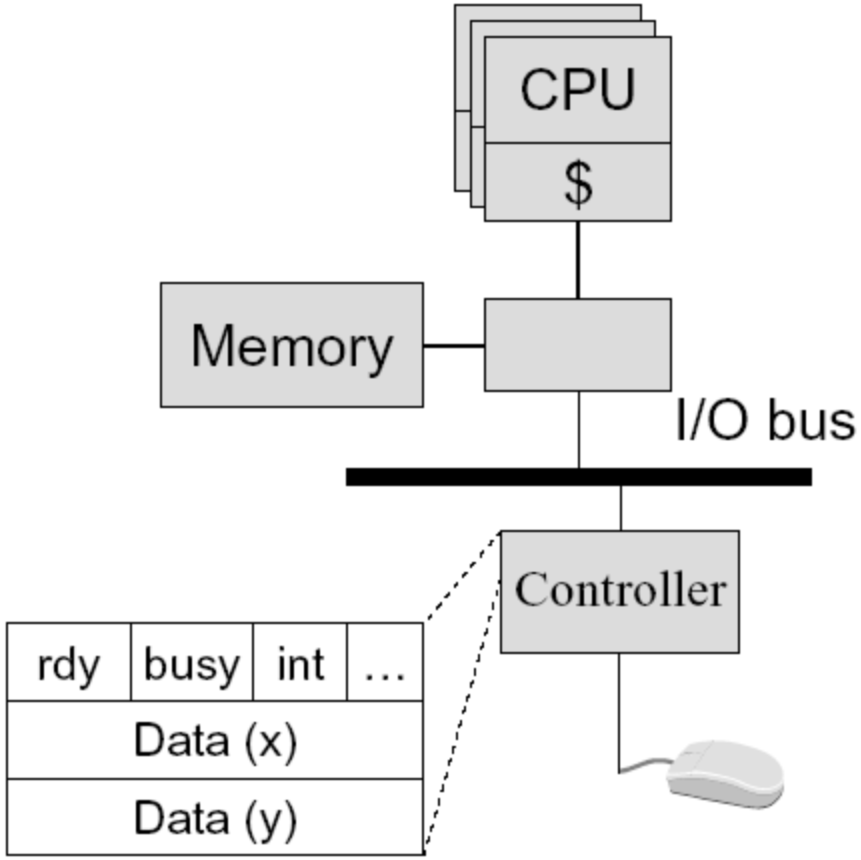
- Overhead
  - CPU time to initiate an operation
- Latency
  - Time to transfer one byte
  - Overhead + 1 byte reaches destination
- Bandwidth
  - Rate of I/O transfer, once initiated
  - Mbytes/sec
- General method
  - Abstraction of byte transfers
  - Batch transfers into block I/O for efficiency to prorate overhead and latency over a large unit

Initiate

Data transfer

Programmed Input Device

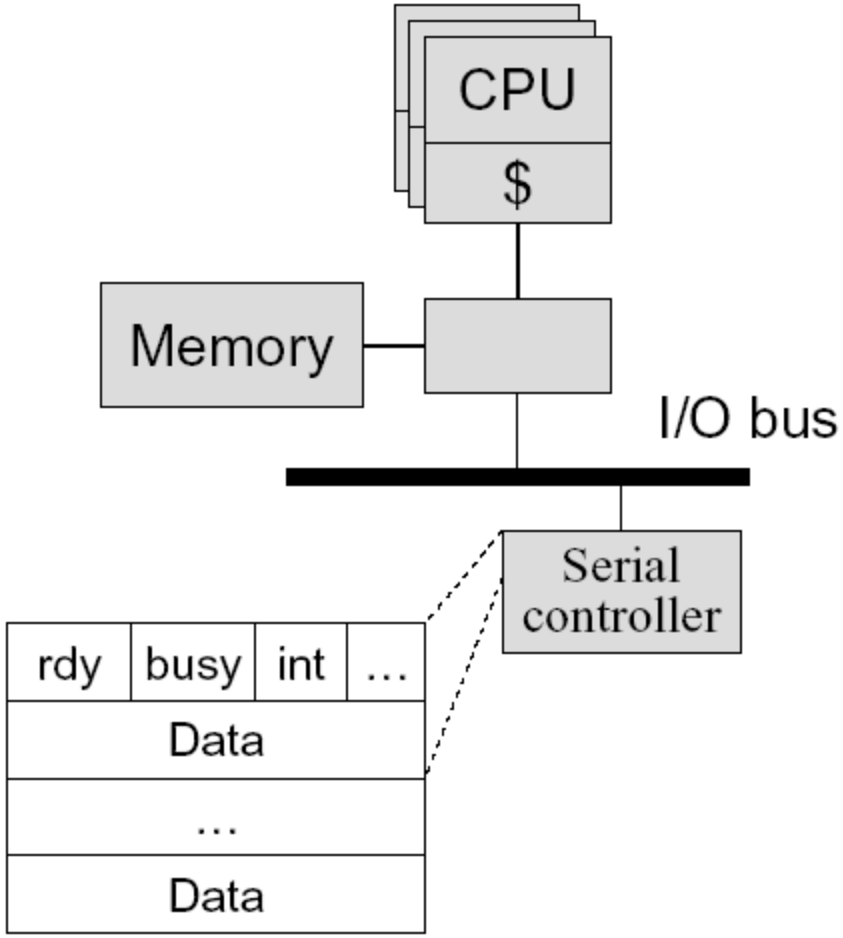
- Device controller
  - Status register
    ready: if the host is done
    busy: if the controller is done
    int: interrupt
    …
  - Data registers
- A simple mouse design
  - Put (X, Y) in data registers on a move
  - Interrupt
- Input on an interrupt
  - Read values in X, Y registers
  - Set ready bit
  - Wake up a process/thread or execute a piece of code



rdy	busy	int	…
Data (x)			
Data (y)			

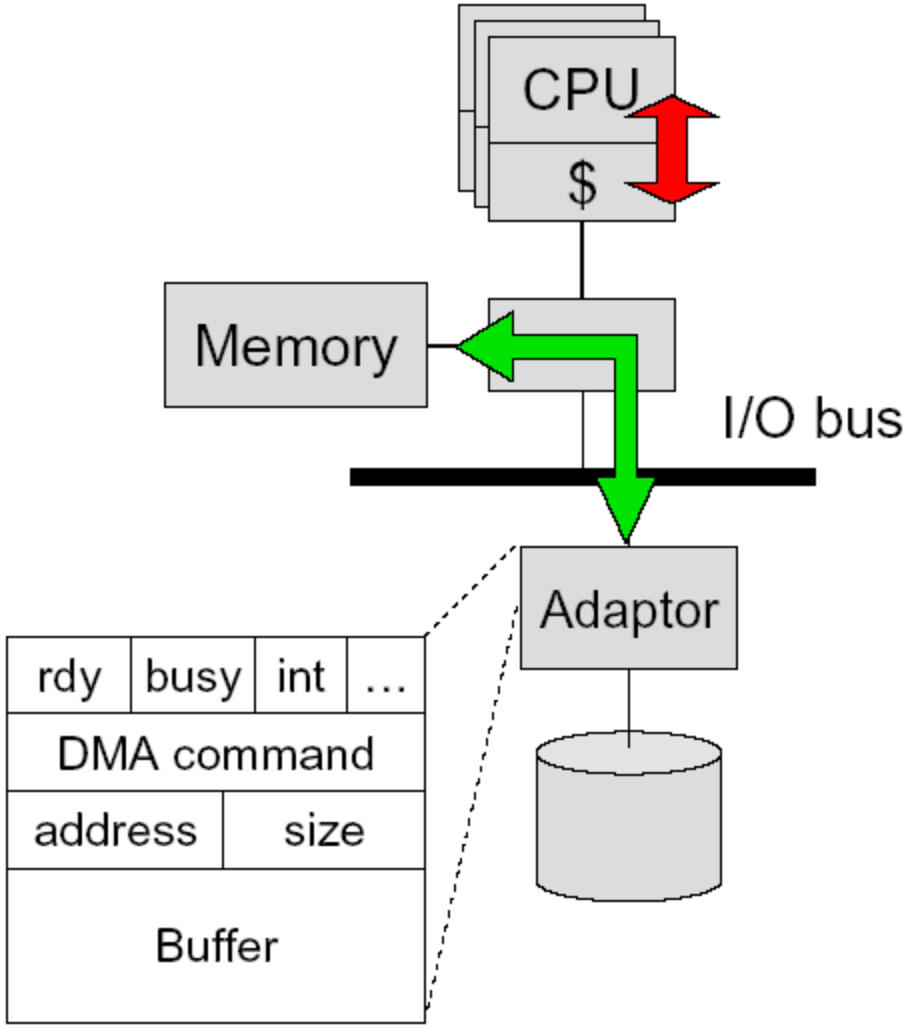
Programmed Output Device

- Device
  - Status registers (ready, busy, …)
  - Data registers
- Example
  - A serial output device
- Perform an output
  - Wait until ready bit is clear
  - Poll the busy bit
  - Writes the data to register(s)
  - Set ready bit
  - Controller sets busy bit and transfers data
  - Controller clears the ready bit and busy bit

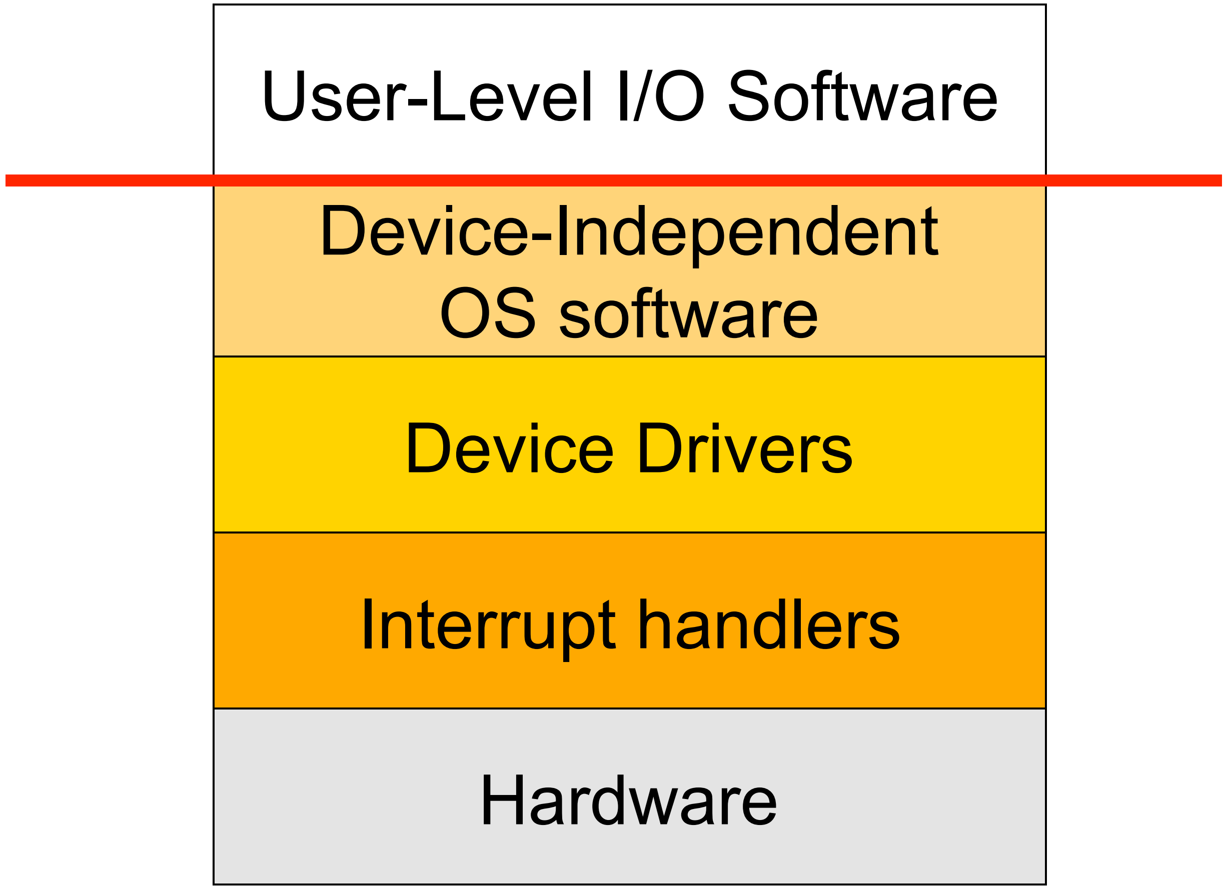


Direct Memory Access (DMA)

- DMA controller or adaptor
  - Status register
    (ready, busy, interrupt, …)
  - DMA command register
  - DMA register (address, size)
  - DMA buffer
- Host CPU initiates DMA
  - Device driver call (kernel mode)
  - Wait until DMA device is free
  - Initiate a DMA transaction
    (command, memory address, size)
  - Block
- Controller performs DMA
  - DMA data to device
    (size--; address++)
  - Interrupt on completion
    (size == 0)
- Interrupt handler (on completion)
  - Wakeup the blocked process



I/O Software Stack



User-Level I/O Software

Device-Independent OS software

Device Drivers

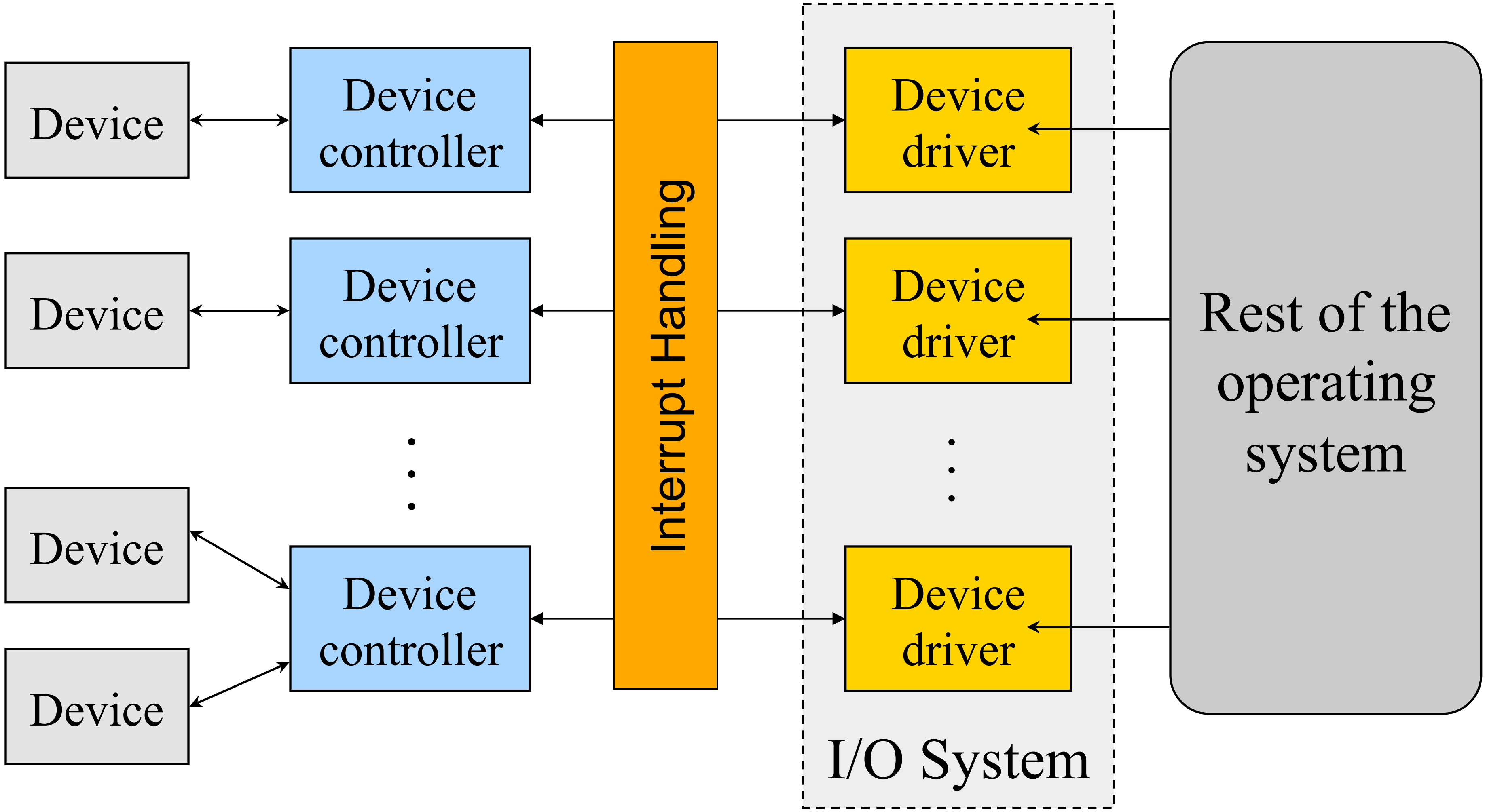
Interrupt handlers

Hardware

Recall Interrupt Handling

- Save context
- Mask interrupts if needed
- Set up a context for interrupt service
- Set up a stack for interrupt service
- Acknowledge the interrupt controller, enable it if needed
- Save entire context to PCB
- **Run the interrupt service**
- Unmask interrupts if needed
- Possibly change the priority of the process
- Run the scheduler

Device Drivers



Device

Device

Device

Device

Device controller

Device controller

Device controller

Interrupt Handling

Device driver

Device driver

Device driver

I/O System

Rest of the operating system

A Typical Device Driver Design

- Operating system and driver communication
  - Commands and data between OS and device drivers
- Driver and hardware communication
  - Commands and data between driver and hardware
- Driver operations
  - Initialize devices
  - Interpreting commands from OS
  - Schedule multiple outstanding requests
  - Manage data transfers
  - Accept and process interrupts
  - Maintain the integrity of driver and kernel data structures

Device Driver Interface

- Open(deviceNumber)
  - Initialization and allocate resources (buffers)
- Close(deviceNumber)
  - Cleanup, deallocate, and possibly turnoff
- Device driver types
  - Block: fixed sized block data transfer
  - Character: variable sized data transfer
  - Terminal: character driver with terminal control
  - Network: streams for networking

Character and Block Device Interfaces

- Character device interface
  - read(deviceNumber, bufferAddr, size)
    - Reads “size” bytes from a byte stream device to “bufferAddr”
  - write(deviceNumber, bufferAddr, size)
    - Write “size” bytes from “bufferAddr” to a byte stream device
- Block device interface
  - read(deviceNumber, deviceAddr, bufferAddr)
    - Transfer a block of data from “deviceAddr” to “bufferAddr”
  - write(deviceNumber, deviceAddr, bufferAddr)
    - Transfer a block of data from “bufferAddr” to “deviceAddr”
  - seek(deviceNumber, deviceAddress)
    - Move the head to the correct position
    - Usually not necessary

Unix Device Driver Interface Entry Points

- `init()`
  - Initialize hardware
- `start()`
  - Boot time initialization (require system services)
- `open(dev, flag, id)` and `close(dev, flag, id)`
  - Initialization resources for read or write, and release afterwards
- `halt()`
  - Call before the system is shutdown
- `intr(vector)`
  - Called by the kernel on a hardware interrupt
- `read(…)` and `write()` calls
  - Data transfer
- `poll(pri)`
  - Called by the kernel 25 to 100 times a second
- `ioctl(dev, cmd, arg, mode)`
  - special request processing

Synchronous vs. Asynchronous I/O

- Synchronous I/O
  - read() or write() will block a user process until its completion
  - OS overlaps synchronous I/O with another process
- Asynchronous I/O
  - read() or write() will not block a user process
  - the user process can do other things before I/O completion
  - I/O completion will notify the user process

Detailed Steps of Blocked Read

- A process issues a read call which executes a system call
- System call code checks for correctness and buffer cache
- If it needs to perform I/O, it will issues a device driver call
- Device driver allocates a buffer for read and schedules I/O
- Controller performs DMA data transfer
- Block the current process and schedule a ready process
- Device generates an interrupt on completion
- Interrupt handler stores any data and notifies completion
- Move data from kernel buffer to user buffer
- Wakeup blocked process (make it ready)
- User process continues when it is scheduled to run

Asynchronous I/O

- API
  - Non-blocking read() and write()
  - Status checking call
  - Notification call
  - Different form the synchronous I/O API
- Implementation
  - On a write
    - Copy to a **system buffer**, initiate the write and return
    - Interrupt on completion or check status
  - On a read
    - Copy data from a **system buffer** if the data is there
    - Otherwise, return with a special status

Why Buffering?

- Speed mismatch between the producer and consumer
    - Character device and block device, for example
    - Adapt different data transfer sizes (packets vs. streams)
- Deal with address translation
    - I/O devices see physical memory
    - User programs use virtual memory
- Spooling
    - Avoid deadlock problems
- Caching
    - Avoid I/O operations

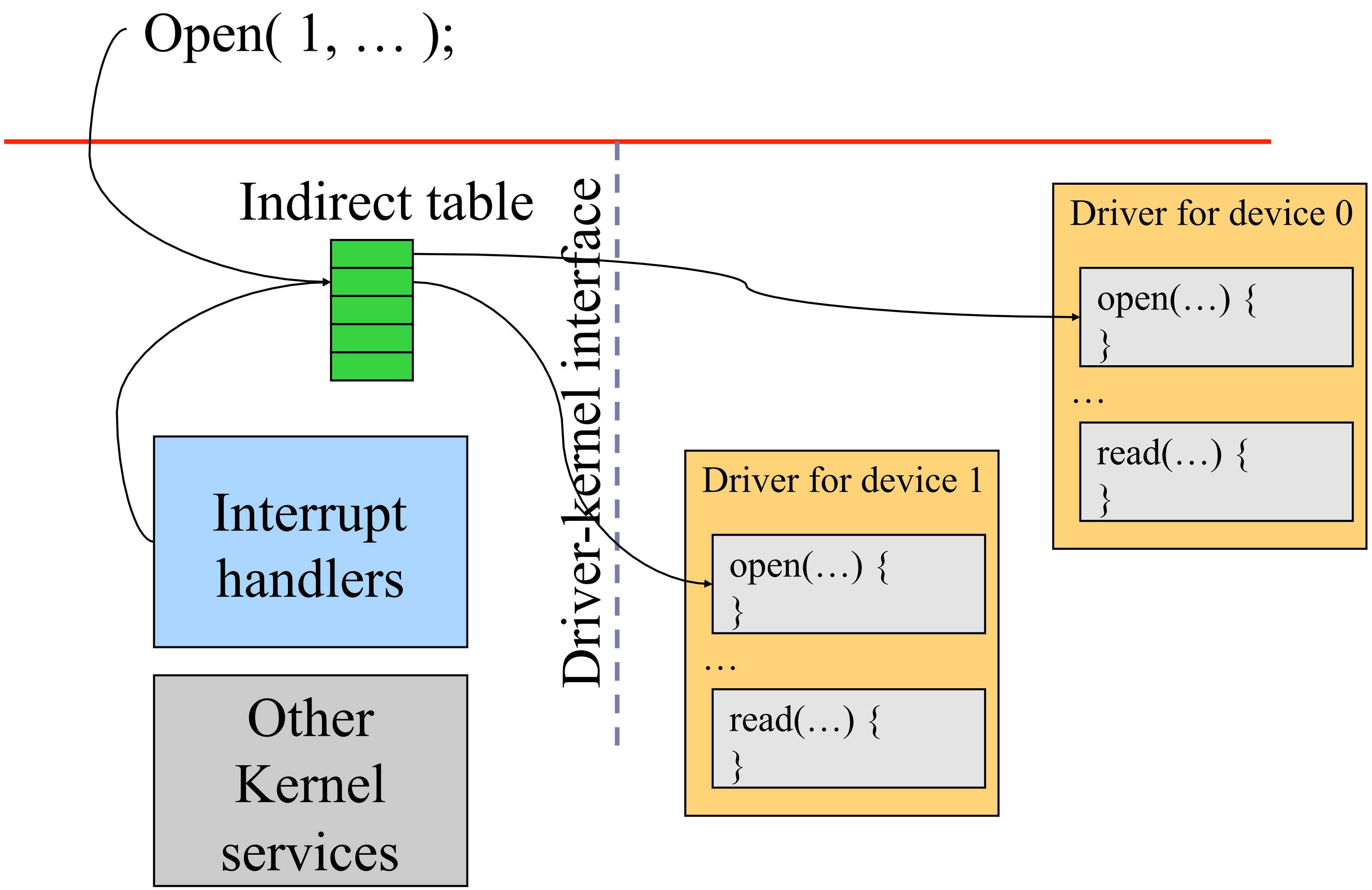
Think About Performance

- A terminal connects to computer via a serial line
  - Type character and get characters back to display
  - RS-232 is bit serial: start bit, character code, stop bit (9600 baud)
- Do we have any cycles left?
  - 10 users or 10 modems
  - 900 interrupts/sec per user
  - What should the overhead of an interrupt be
- Technique to minimize interrupt overhead
  - Interrupt coalescing

Other Design Issues

- Build device drivers
  - Statically
    - A new device driver requires reboot OS
  - Dynamically
    - Download a device driver without rebooting OS
    - Almost every modern OS has this capability
- How to down load device driver dynamically?
  - Load drivers into kernel memory
  - Install entry points and maintain related data structures
  - Initialize the device drivers

Dynamic Binding: Indirection



Issues with Device Drivers

- Flexible for users, ISVs and IHVs
  - Users can download and install device drivers
  - Vendors can work with open hardware platforms
- Dangerous methods
  - Device drivers run in kernel mode
  - Bad device drivers can cause kernel crashes and introduce security holes

- Progress on making device driver more secure
  - Checking device driver codes
  - Build state machines for device drivers

Summary

- IO Devices
  - Programmed I/O is simple but inefficient
  - DMA is efficient (asynchronous) and complex
- Device drivers
  - Dominate the code size of OS
  - Dynamic binding is desirable for desktops or laptops
  - Device drivers can introduce security holes
  - Progress on secure code for device drivers but completely removing device driver security is still an open problem
- Synchronous and Asynchronous I/O
  - Synchronous I/O is simple
  - Asynchronous I/O are efficient but difficult to use