Devices and Drivers

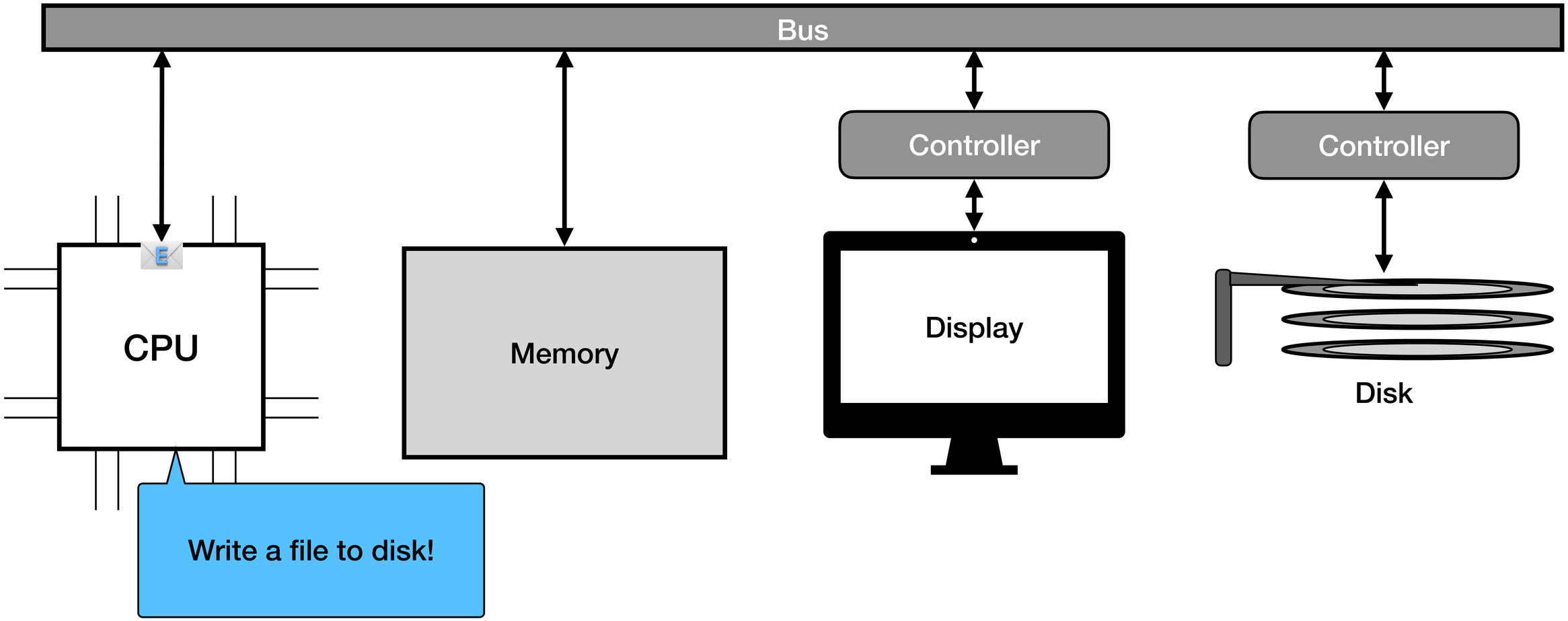
MM (programming) due Friday!

Outline

- Characterizing I/O devices
- Discovering new devices
- Designing a device driver

A Simple I/O Architecture

A *bus* is a collection of wires with *ports* for different controllers to attach to where messages can be sent



Chat with your neighbor(s)!

To send messages to a device, the OS tells the CPU (by executing an instruction) to notify the device of the operation to perform. How would the device indicate to the OS that the operation has completed?

The devices need to have some mechanism to communicate with the CPU! This means that the OS and the device need to share state

The OS device drivers need to know where the device will save the result of the operation!

Example: Programmed I/O (PIO) Devices

PIO devices have four 1-byte registers in OS memory

A typical interpretation of the bits in these registers may be as follows:

Write a byte to a device!

GoR	GoW = 1	IER	IEW					Control Register
RdyR	RdyW							Status Register
								Read Register
0	0	1	0	1	1	0	1	Write Register

Set the byte to be written in the write register

Set the “Go Write” (start write operation) bit to be true

Synchronous: spin on “RdyW” until it is set

Asynchronous: set IEW (interrupt enable write) to be true

Interfacing with Devices

- In order to interact with I/O devices (devices that are dynamically attached rather than integrated on the motherboard), the OS needs to have a well defined interface to be able know what messages to send along the system bus
- To communicate to a particular device, the OS and the device need to have shared space to notify the other that an operation needs to happen or has happened, respectively ➡️ the addresses of these shared memory locations need to be communicated on initialization
- The standard interfaces for the OS to do so are Programmable I/O devices (PIO) in which the OS maintains PIO registers for each device or Direct-Memory Access (DMA) devices in which the OS uses an agreed upon location in the “Zone DMA” region of OS memory ➡️ the code that implements this interface is called the “device driver”
- PIO is a simpler interface that is more CPU intensive whereas DMA devices have their own controller that writes to the OS kernel memory ➡️ examples of PIO devices are typically peripherals (e.g., keyboard, mouse, etc…) whereas DMA requires higher throughput (e.g., disks, network cards, etc…)

Chat with your neighbor(s)!

Think about the potential configurations of various I/O devices that are attached in an architecture. Why might the task of implementing drivers for MacOS be more difficult than Ubuntu or Windows OS?

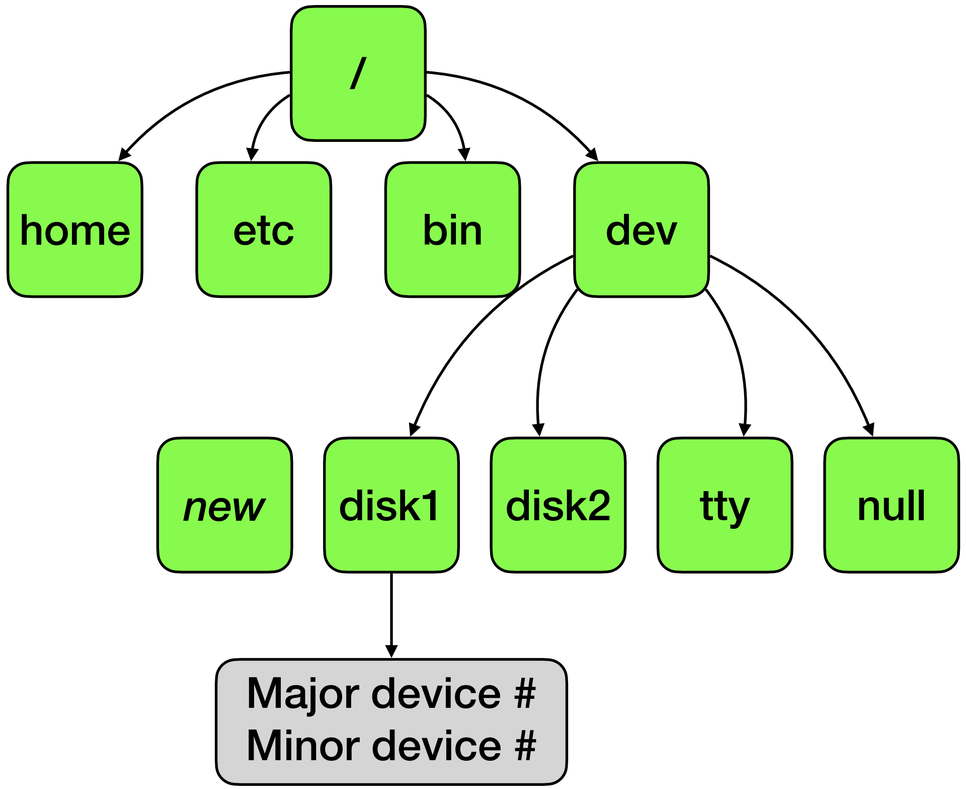
Attached I/O devices are heterogenous and dynamic!

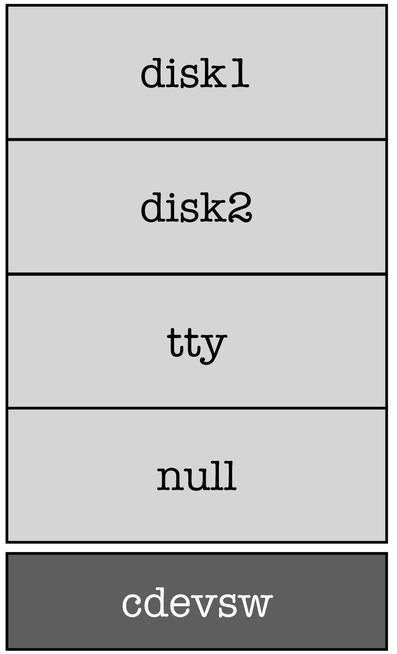
If you have full knowledge of exactly what the attached devices are, then you can program the OS to statically implement these handlers

Modern OSes need to implement *dynamic device discovery* and maintenance!

Statically Maintaining Devices

Goal: have a dynamic mechanism to track which devices and what kind of devices are attached





How to handle new device?

Add driver to OS code (new entry in static table)

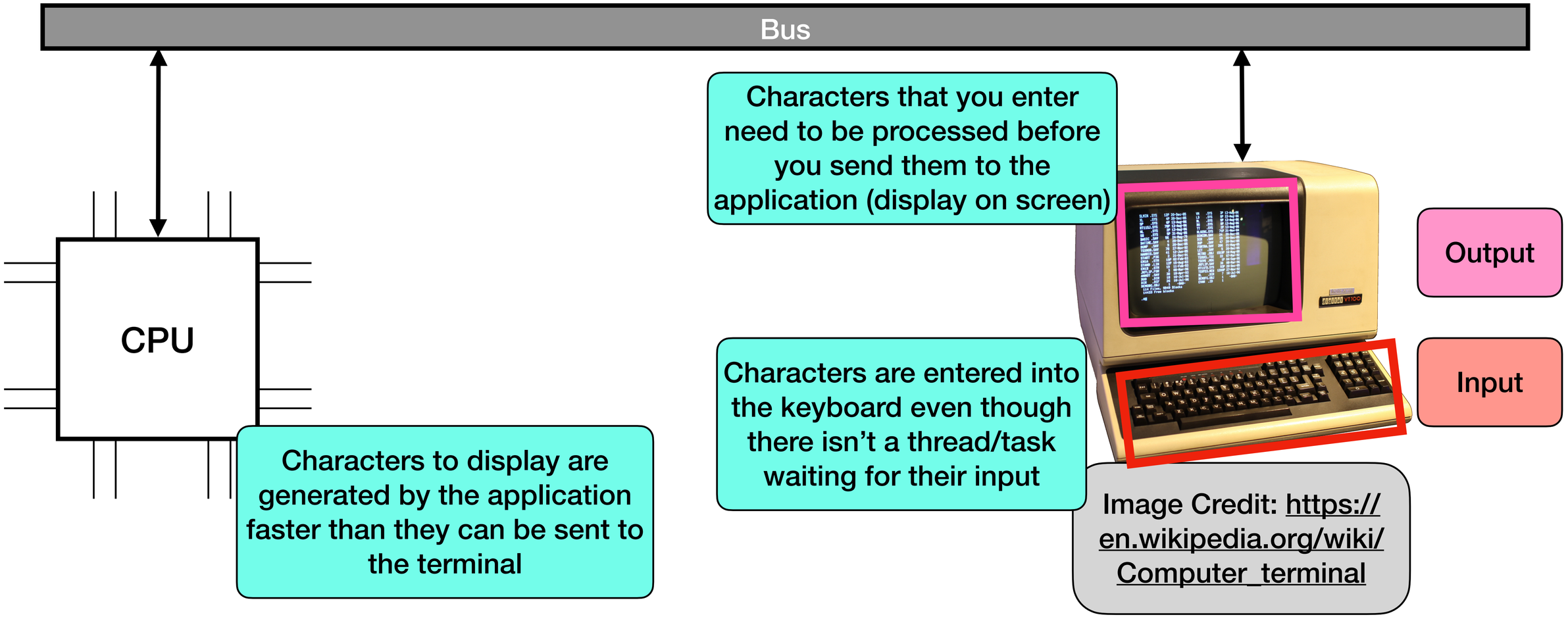
Recompile OS and reboot with this version of the OS

🙏

Maintaining Devices

- There are thousands of types of devices that could all be implemented… how does the OS kernel have the implementation of the drivers for each of them?
- If done statically, then the OS needs to have *all possible* device drivers in the OS kernel or the OS needs to be recompiled every time a device is attached…
- To work around this, modern I/O architectures implement a handler for a particular *bus-type* rather than device ➡️ the OS monitors ports (USB, PCIe, etc…) and an interrupt is raised when the device is attached
- When a device is attached, the USB protocol notifies the OS kernel of the device type being attached so that it knows which driver is being used ➡️ the OS then needs to do a lookup in its device driver library (`/dev`) for the right interface for that device

Case Study: Terminal



Chat with your neighbor(s)!

Characters to display are generated by the application faster than they can be sent to the terminal

Characters are entered into the keyboard even though there isn't a thread/task waiting for their input

Characters that you enter need to be processed before you send them to the application (display on screen)

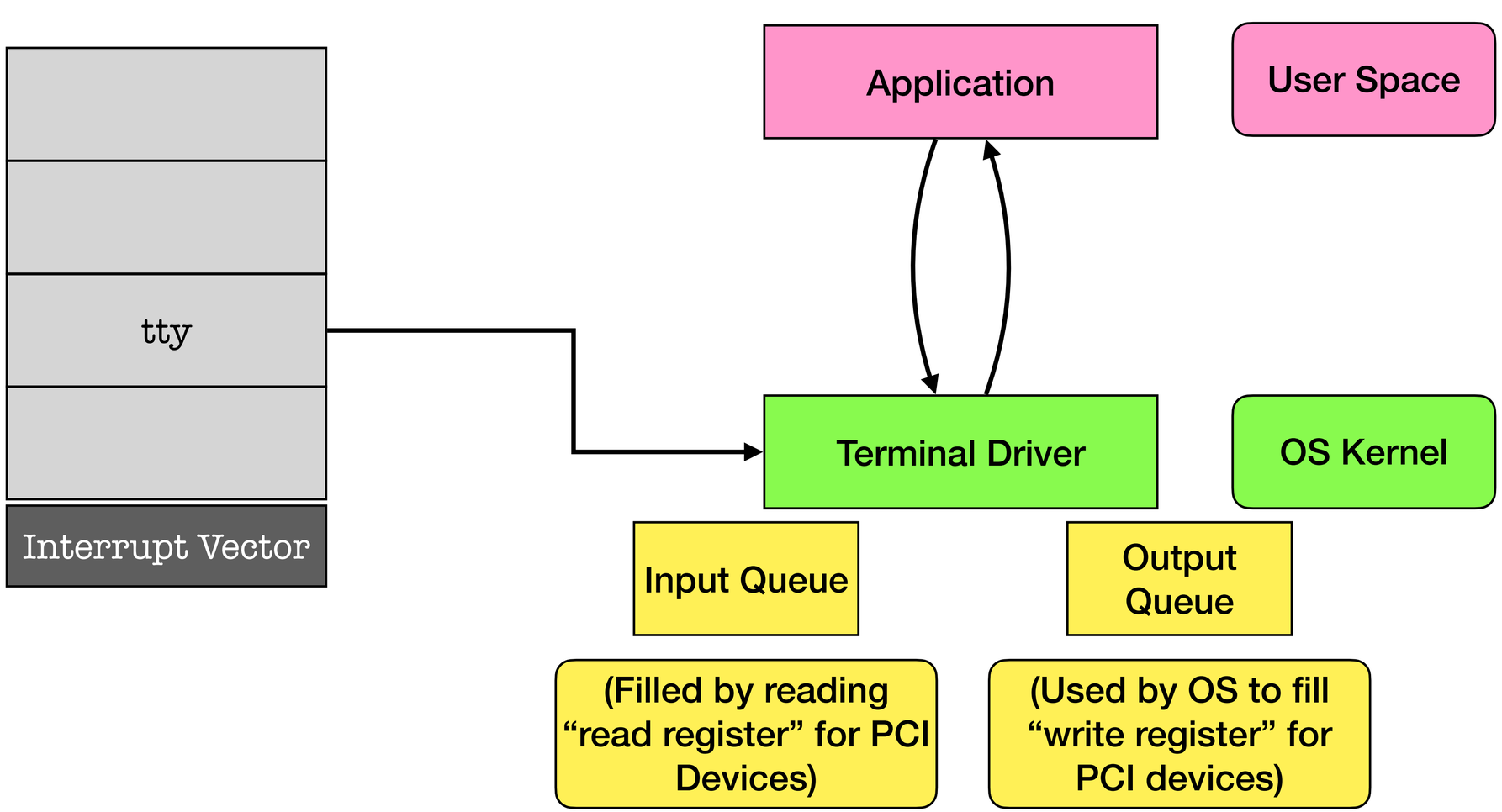
Thinking about the previous concerns related to maintaining the terminal device, what does the operating system need to implement to allow the application on the device to interface with the terminal?

We need a data structure to manage both the input and the output!

When characters are entered into the terminal, the OS needs a queue to handle these characters as input

When the application wishes to communicate the result of the operation back to the terminal, it maintains a copy of what it writing until the terminal notifies that the write has completed

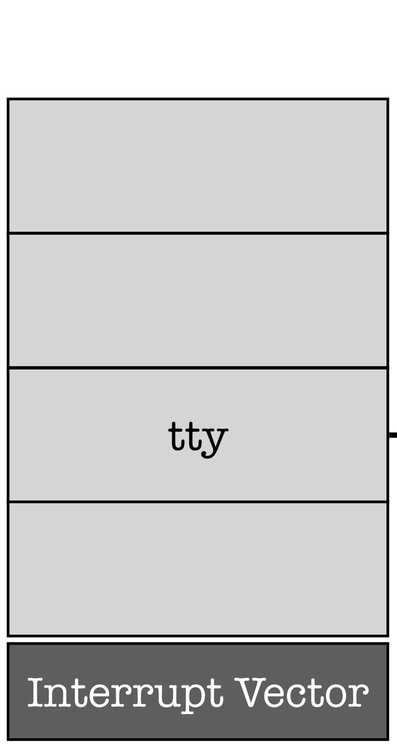
Case Study: Terminal



Case Study: Terminal

- The terminal device interacts with the operating system by writing the byte associated with a character into the “read register” of the PIO device and raising an interrupt
- In handling the interrupt, the OS reads the data from the “read register” into a buffer of all of the characters that it will send to an application or that it will process itself (start a process, etc…)
- When the application runs, it will ultimately produce some output that the OS will need to communicate back to the terminal to display ➡️ this output is maintained in a queue, whose contents are sent to the device by writing to the “write register”

Case Study: Terminal



Application

User Space

Characters to display are generated by the application faster than they can be sent to the terminal

Characters are entered into the keyboard even though there isn't a thread/task waiting for their input

Terminal Driver

OS Kernel

Input Queue

Output Queue

Characters that you enter need to be processed before you send them to the application (display on screen)

(Filled by reading "read register" for PCI Devices)

(Used by OS to fill "write register" for PCI devices)

Chat with your neighbor(s)!

Suppose you were to implement a device driver for a network card. How would you structure your driver?

Summary

- A “device driver” generally describes the OS code associated with the management and maintenance of a device
- To communicate with a device, the device needs to be implemented knowing how it will interact with the OS (either as a PIO or DMA device) ➡️ the OS then needs to communicate to the device where the associated shared data will be
- In order to implement the device driver, the driver must: (1) manage the shared state, (2) have a mechanism to receive large quantities of shared state (if applicable), (3) notify the device of the response to influence its next state (if applicable)