Overview

Let’s explore how to integrate I/O into a system

This section should help us answer the following:

- **What is the best way to integrate I/O into a system?**
- **What are the basic mechanisms at work?**
- **What can we do to make them more efficient?**

Introduction

Being able to receive **input** and produce an **output** is an important part of a computer system. **Input/output (I/O)** allows our system to receive information, do something with it, and send us a response in return.

When we say I/O **devices**, we generally mean objects that are the CPU handles that are *not* memory, and are connected to a computer system. There are many types of devices, including but not limited to:

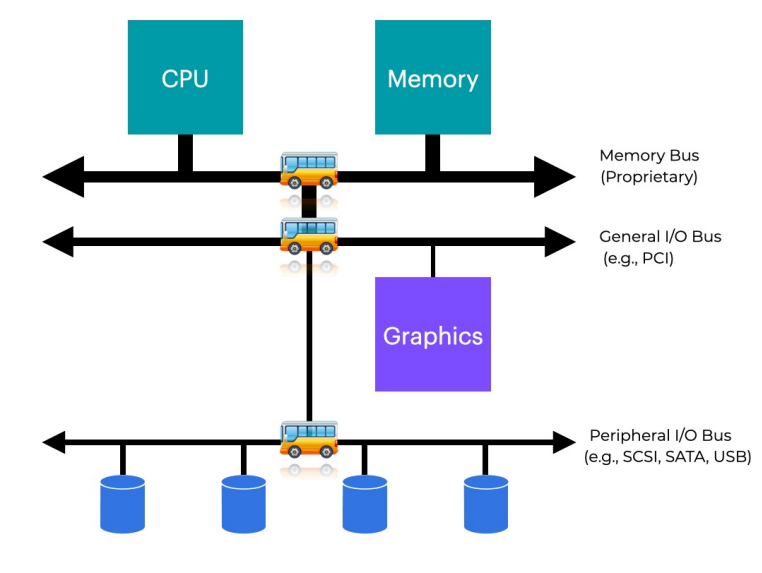
- SSD
- Hard Drives
- Graphics
- Networking
- Mouse/ Keyboards
- Audio

System Architecture: The Bus

The CPU is connected to the system's main memory through a memory bus or connection.

A **bus** is a system for transferring data between components inside a computer or between computers

- Some devices, like graphics and high performance I/O devices are connected to the system using a **general I/O** bus like a PCI.
- **Peripheral buses** like SCSI, SATA, and USB are used to connect slower devices like disks, mouse, and keyboards.



.guides/img/buses

Why is there a bus hierarchy for I/O?

Because of both **physics** and **cost**. Faster buses have to be shorter, so a high performance memory bus doesn't have extra space for devices and peripherals.

Designing multiple high-performance buses are **expensive**, so it's not something we want to do more of than we have to. Systems designers put high performance parts closer to the CPU and lower performance parts further away. This give the CPU the ability to accommodate several devices.

▼ **What is a peripheral?**

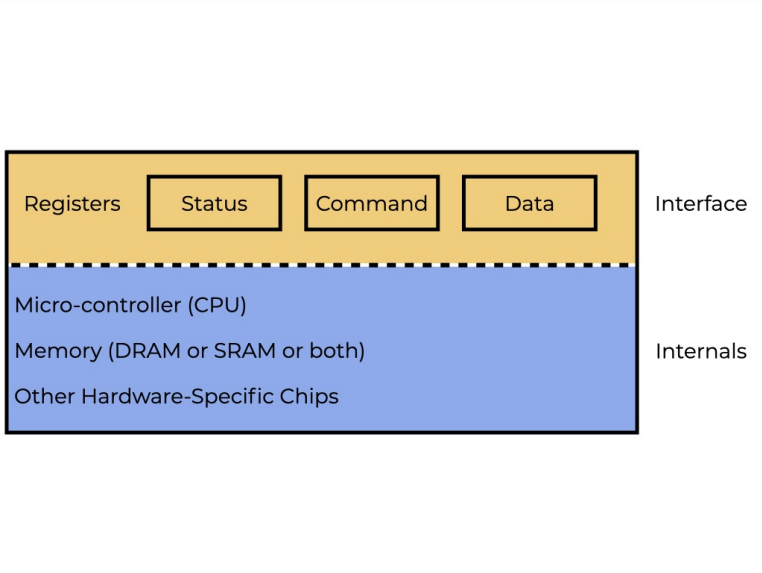
A peripheral is an internal or external device that connects directly to a computer or other digital device but doesn't contribute to the computer's primary function. It helps end users access and use the functionalities of a computer.

Checkpoint

Which of the following is NOT true of I/O bus hierarchy?

A Canonical Device

Let's look at an imaginary **canonical device** to help us understand some of the machinery we need for efficient device interaction.



.guides/img/can_device.001

- A device includes two essential parts:

1. The hardware **interface** it provides to the rest of the system.
    - Hardware, like software, has to use some kind of **interface** that lets the system software control its operation.
    - All devices have a specific interface and protocol for normal interactions.
2. The **internal structure** of the device.

- This element of the device is implementation-specific, and it's in charge of putting the device's abstraction to work for the system. Like having a mini-computer dedicated to one task.

The most simple devices depend on one or two hardware chips to do their jobs. More complex devices depend on a CPU, some general-purpose memory, and other various device-specific chips.

Checkpoint

There are two essential parts of a canonical device. Which of the following belong within the interface of the canonical device?

The Canonical Protocol

This simple interface has three registers:

1. **Status Register** - Is read to check the devices current status
2. **Command Register** - Used to tell the device to do a particular job
3. **Data Register** - Used to send/receive data to/from the device

The OS can change a devices behavior by reading and writing to these registers. The protocol below shows how the OS and device collaborate to finish a task:

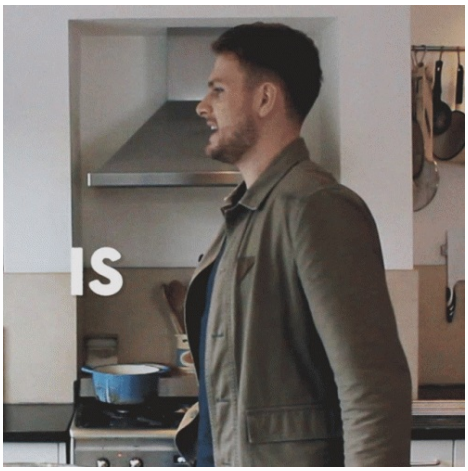
```
 While (deviceStatus == busy); // wait until device is free
   // Write some data to DATA register
   // Write some command to COMMAND register

       (start device and execute command)
   While (deviceStatus == busy); // wait until device is
finished
```

There are 4 parts to how the OS interacts with a device:
1. **Polling** the device to check its status regularly until it is ready for a command
1. **Sending data** to the data register
1. **Writing a command to the command register** to tell the device that the data and command are there to be processed.
1. **Polling** the devices repeatedly to see if its either finished or encounters an error

This works, but has a big problem…

It's inefficient. It wastes a lot of CPU time waiting for a device that could be slow to finish it's job instead of switching to another task.

How can the OS check a device's status without having to ask it repeatedly, and reduce the time the CPU needs to manage it?



Checkpoint

Match the Canonical Device Register with its description.

Lowering CPU Overhead With Interrupts

Instead of constantly polling the device, The OS can use an **interrupt** to:
* Send a request
* Put the calling process to sleep, and
* Switch the context to another job.

When the I/O request is finished, it sends a **hardware interrupt** and makes the CPU go into the OS to a specific **interrupt handler**. This **handler** is OS code that will complete the request, wake up the process that's waiting for I/O to finish, and lets it continue about its merry way.

Interrupts let computation and I/O **overlap**, which is **important for an efficient system**.

.guides/img/no_overlap

In the graphic above:

- An I/O request is sent to disk, causing **Process 1** to be blocked.
- Then the system continues without an interrupt, constantly requesting the device's status until the I/O is finished.
- When the disk finishes the request **Process 1** can continue about its merry way.

If we use interrupts and allow overlap, the OS can do something else instead of just waiting for the disk.

.guides/img/overlap

- The OS runs **Process 2** on the CPU while the disk is handling **Process 1's** request.
- When the request is finished, in interrupt is issued and the OS wakes up **Process 1** so it can continue.

The disk and CPU are being effectively used the entire time, which is a good thing.

Interrupts aren't always the best choice, though.

If a device is fast to finish, taking the time to switch processes, handle the interrupt, and switch back would slow down the system too much to be worth it.

- If the device does it's job fast, **polling** may be the better option.
- If it is slow enough to allow for overlapping, **interrupts** might be your best bet.

If the device speed is unknown or unreliable, using a little of both in your approach could give you the best of outcome.

Checkpoint

When would it be more efficient to use an interrupt?

Moving Data With DMA

The CPU wastes time manually transporting data between devices using programmed I/O (PIO).

How can we offload this job to make better use of the CPU?

.guides/img/dma

In the graphic above:

- Process 1 is running and wants to write some data to the disk. It starts the I/O, which has to copy the data from memory to the device, one word at a time.
- When the copy is finished, I/O on the disk begins, letting the CPU to be used for other things.

Direct Memory Access (DMA) is the our answer to this problem. A **DMA engine** is a system component that handles transfers between devices and main memory without relying so much on the CPU.

- The OS tells the DMA engine where and how much data to copy and what device to send data to.
- Now, the OS is finished with the transfer and can go on to other tasks.
- The DMA engine raises an interrupt when its finished, letting the OS know that the transfer is done.

Now, the DMA engine is in charge of copying data.

.guides/img/copydma

Since the CPU is free, the OS can do something else (like run **Process 2**). **Process 2** gets to use some CPU before **Process 1** starts up again.

Which of the following allows the CPU use of a dedicated memory controller?

Methods Of Device Interaction

How should a device's hardware communicate with the OS?

Device communication is generally done in two ways:

1. Having **explicit I/O instructions** that defines how the OS delivers data to specific device registers.
    - Normally, these instructions are privileged. Only the OS is allowed to interface directly with devices because it controls them.
2. **Memory-mapped I/O** is the second method of interacting with devices.
    - This makes device registers available like they were memory addresses.
    - To access a specific register, the OS issues a load (to read) or store (to write) address.
    - The hardware then delivers the load/store to the device instead of main memory.

There isn't much difference between the two. Both are still in use today, although the memory-mapped approach has the benefit of not needing new instructions.

Fill in the blanks to complete the statements below.

The Device Driver

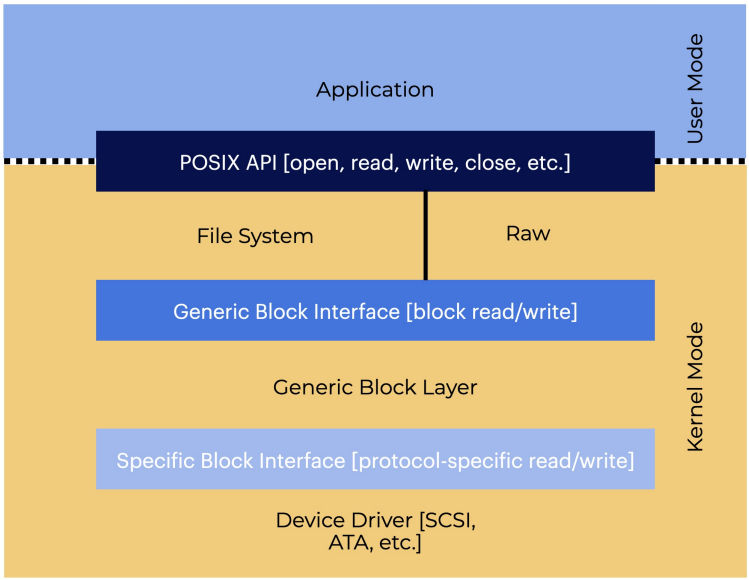
Let's investigate how to integrate devices with special interfaces into an OS that we want to make as general as possible.

How can we ensure that our operating system is neutral enough for all types of devices while hiding device interactions from key OS subsystems?

We can handle this with **abstraction**.

The OS has to understand how a device works in detail. Device interface functionalities are wrapped within this piece of software called a **device driver**.

Let's explore the Linux file system software stack.



.guides/img/deviceDriver

Here, a file system (or application) doesn't care about disk class. It just sends read and write requests to the generic block layer, which routes them to the appropriate driver. This shows how operating systems can hide information with abstraction.

Certain programs can read and write blocks directly without needing the file abstraction. Most systems support low-level storage management applications using this interface.

This encapsulation does have some drawbacks. If a device has a lot of unique features but needs to offer a generic interface to the rest of the kernel, those unique features will be wasted.
#
Checkpoint
Which of the following is a piece of software that encapsulates the functionalities of a device's interface?

Summary

You should now have a good idea of how an operating system interacts with a device.

- Several strategies, including interrupts and DMA, have been developed for enhancing device efficiency.
- Specific I/O instructions and memory-mapped I/O have also been proposed.
- Finally, the concept of a device driver has been introduced, showing how the OS can capture low-level features, making it easier to develop the rest of the OS in a device-agnostic manner.