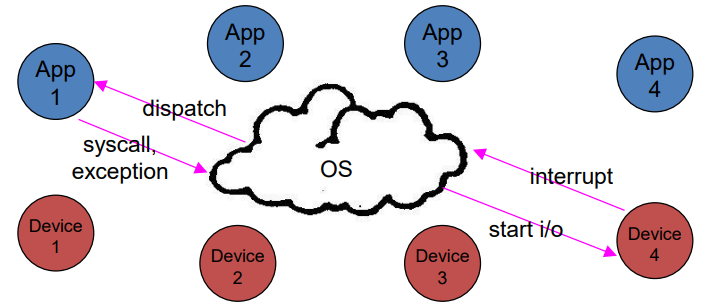
Operating Systems

Structure

OS Structure

The OS mediates access and abstracts away ugliness, sitting between applications and the hardware. Applications can request services explicitly (via syscalls) or Implicitly (via exceptions) and devices request attention via interrupts.

Operating System Design and Implementation

The design and implementation of the OS is not a ‘solvable’ problem but some approaches have proven successful. The internal structure of OSs can vary widely.

The design process starts by defining the goals and specifications:

* User goals: convenient to use, easy to learn, reliable, safe and fast
* System goals: easy to design, implement, and maintain as well as being flexible, reliable, error-free and efficient

These design choices may also be affected by the hardware and type of system being built.

An important principle in OS design is the principle of separation, specifically the separation of policy (what will be done? (algorithm) rules that are independent of whatever enforces them) and mechanism (how it will be done?).

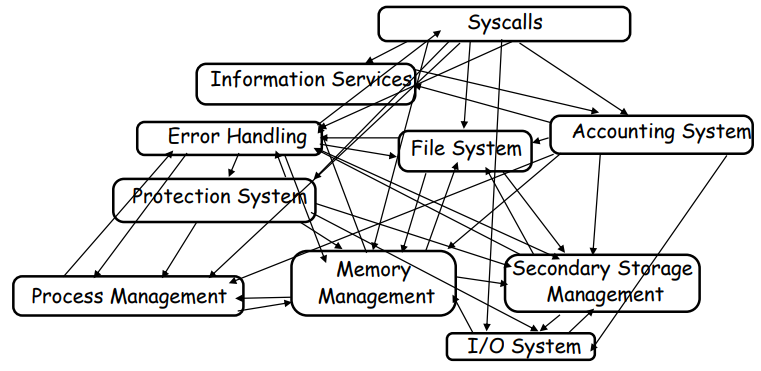
This separation allows as much flexibility as possible, policies are likely to change across places or over time and a general mechanism can support a wide range of policies.

Microkernel OSes are based on such a principle, a care kernel implements the mechanisms and policies are implemented outside the core kernel making them easily modifiable.

Major OS Services

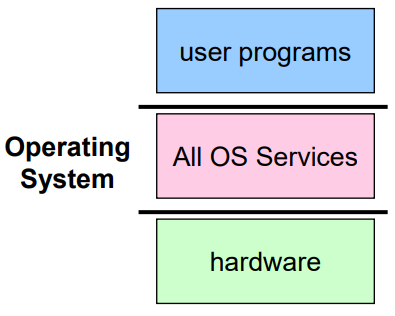
The major Services OSs tend to provide include: Processes management; Memory management; I/O management; Secondary storage management; File systems; Protection of resources; Networking; Shells (command interpreters); GUIs Etc…

For the most part, If a service doesn’t require privileged operations, OS designers have not included them in the kernel (for example, Shells and GUI wouldn’t be in the kernel).

OS Structure

It isn’t always clear how to stich the OS services together. How do we organise all of them, what are all of the code modules and where should they exist, how do they cooperate?

This is a massive software engineering and design problem, we need to design a large, complex program that performs well, is reliable, is extensible, is backwards compatible etc…

Monolithic OS Design

Monolithic design was likely the earliest form of OS organisation, UNIX was built as a monolithic OS (as is Linux). This is where all the OS services that can’t be user programs are implemented in the kernel itself. The mayjor advantage of monolithic design is that the cost os subsystem interactions (procedure call) is low, however it has many disadvantages too, including that it’s hard to understand, modify and maintain as well as being unreliable as there’s no isolation between system modules.

Layered OS Design

The traditional approach for OS design is layering. We implement the OS as a set of layers, each layer presenting an enhanced ‘virtual machine’ to the layer above (think of the layer model of the internet).

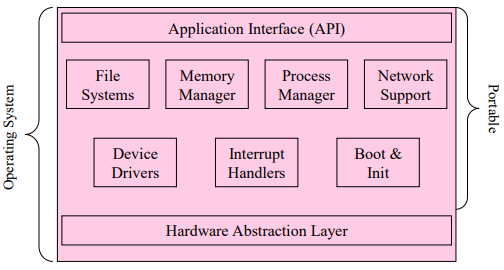
The first description of this approach was Kijkstra’s THE system:

* Layer 0: Hardware
* Layer 1: Kernel. Implements a virtual processor for each process
* Layer 2: Page Manager. Implements virtual memories for each process
* Layer 3: Console Manager. Implements virtual consoles
* Layer 4: Device Managers. Handles devices and provides buffering
* Layer 5: Job Managers. Execute users’ programs

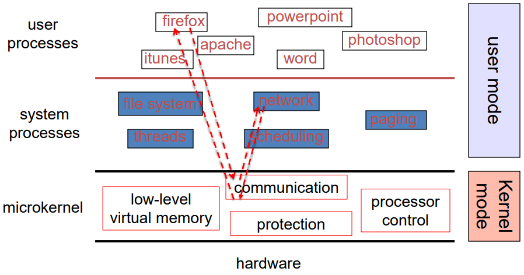
A large benefit of this is that each layer can be tested and verified independently but still has many problems, it imposes hierarchical structure but real systems are more complex than that (a file system requires virtual memory services and virtual memory would like to use files for its backing store, so which is above the other), strict layering isn’t flexible enough.

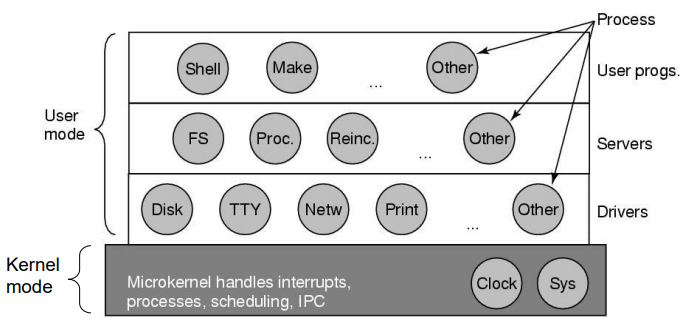
The layered model also has poor performance as each layer crossing has overhead associated with it.

This creates a disjunction between the model and reality where the system is modelled as layers but not really built that way.

Hardware Abstraction Layer

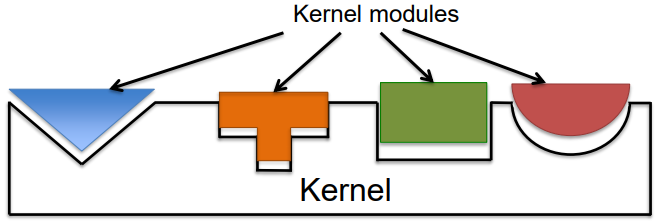
An example of layering in modern operating systems is the hardware abstraction layer. The goal of this is to separate hardware-specific routines from the core kernel of the OS providing portability and improving readability.

Microkernel OS Design

Microkernel OSs were popular in the late 80s/early 90s and have had a recent resurgence in popularity. The goal of microkernels is to minimise what goes in a kernel and organise the rest of the OS as user-level processes (services). This results in better reliability (isolation between components), making extension and customisation of the system easier but it also results in poor performance (user/kernel boundary crossings).

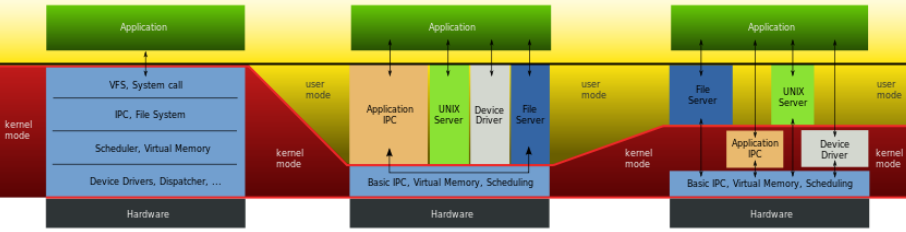
Loadable Kernel Modules

The kernel always consists of a set of core services, but we can dynamically load others as we like.

This is common in modern implementations.

This is very convenient as there’s no need to reboot after adding new modules, efficient as there’s no need for message passing (unlike microkernel) and flexible as any module can call any other module (unlike the layered model).

Hybrid OS Design

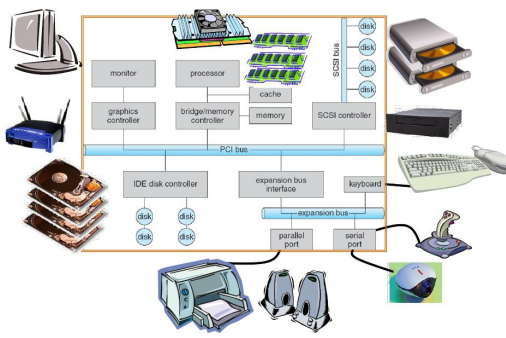
Hybrid OS design (as the name suggests) is where many different approaches are used, this allows us to exploit the benefits of monolithic and microkernel designs at once. This is how many modern OSs are designed (including Windows).

This also adds extensibility via kernel modules.

IO Subsystems

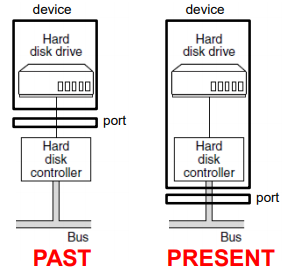
Devices

There are many types of devices that we use with computers:

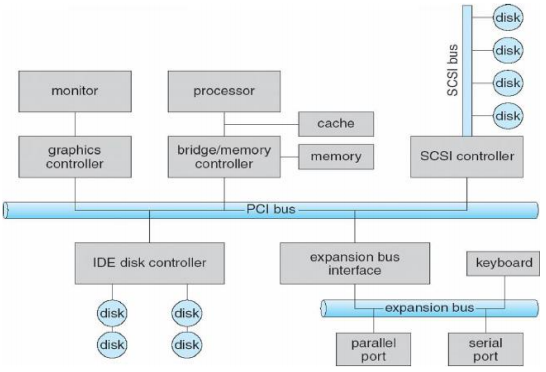
* Storage devices
  + Disk
  + Tape
* Transmission devices
  + Network connections
  + Bluetooth
* Human-interface devices
  + Screen
  + Keyboard
  + Mouse
  + Audio in
  + Audio out
* Specialised devices
  + E.g. to control a machine or specialised equipment (aircraft)

I/O Hardware

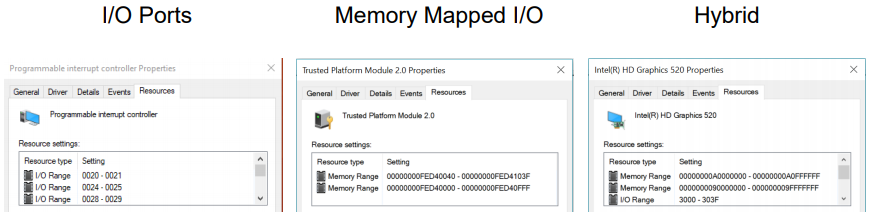
There’s a variety of I/O devices but they all have similar concepts

* Ports
  + The connection point for the device (USB, serial, Ethernet etc…)
* Buss
  + Peripheral busses (PCI/PCIe)
  + Expansion bus, that connects relatively slow devices
* Device
* Controller (host adapter)
  + This is what lets the device talk to the bus
  + Electronics that operate the port, bus and device, sometimes this is integrated or sometimes a separate circuit board (host adapter)
  + This contains a processor, microcode, private memory, a bus controller etc..

The busses (blue) handle the traffic between I/O devices and the processor, the examples here are:

* PCI/PCIe which connects with high speed graphics, networking etc… and to low speed busses
* SCSI which used to be for fast devices with large bandwidth (disks, scanners etc.)

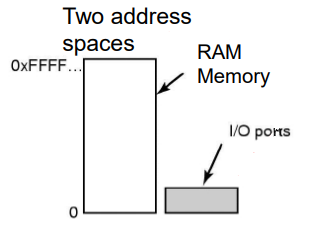
CPU to Device Communication

Controllers have registers for data and control and buffers (memory like areas) mostly for data. The CPU communicates with devices by reading and writing in these. There are three main methods to perform this communication:

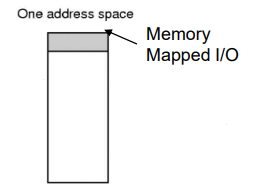
* I/O ports
* Memory-mapped I/O
* Hybrid

I/O Ports

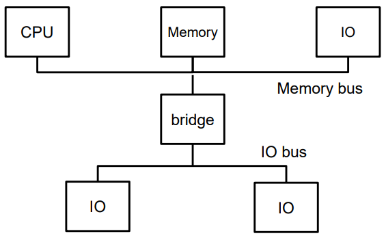
Each control registers an I/O port number, the CPU can then use Special (privileged, so kernel only) instructions to access the I/O port space:

* CPU read in from device I/O port to CPU register
  + IN REG, PORT
* Cpu writes to device I/O port from CPU register
  + OUT PORT, REG

This separates the I/O port space and the memory space so ‘IN R0, 4’ refers to a different place in memory to ‘MOV 4, R0’.

Memory-Mapped I/O

All the control registers and buffers are mapped into the memory space. Each controller register is assigned a unique memory address (there’s no RAM for this address), this address may be at the top of the physical address space.

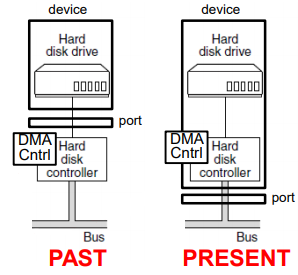
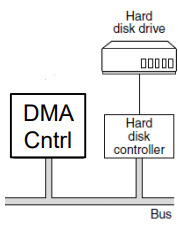
Hybrid

The hybrid approach mixes these two approaches, allowing different devices to use either of the two or a mix, such as memory mapped for the data buffers and ports for the control registers.

Offloaded Communication

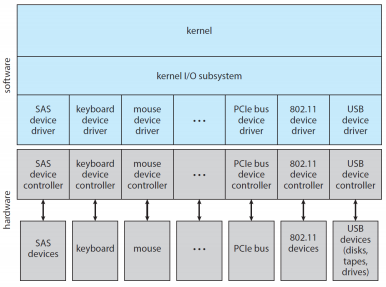
The CPU can request data from an I/O controller one byte at a time, this is called programmed I/O (PIO). This wastes a lot of the CPU’s time for large data transfers (but it’s ok for small ones).

We’d prefer to offload the data transfers, this is done using DMA (Direct Memory Access) controller which transfer the data for the CPU both from/to I/O devices and between I/O devices.

Direct Memory Access

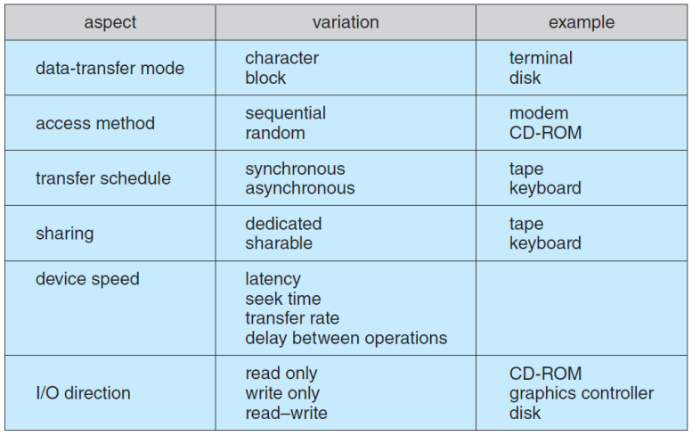
DMA requires a DMA controller either on the device host controller or on the motherboard.

The DMA controller contains register to be read/written by the software including:

* Memory address register
* Byte count register
* Control register to
  + Direction of the transfer
  + Transfer unit
  + Byte burst size
  + Etc…

OS Device Drivers

There is a large variety of devices each with different vendors/models and specs. The OS deals with I/O devices in a standard and uniform way using abstraction, encapsulation and software layering. This is done using a specific interface (file) encapsulating the differences in devices using device dreivers’ classes (each OS has its own standard). This allows an application to open a file without knowing what kind of disk it is.

Characterising I/O Devices

There are many ways for devices to differ, here is a table of important ways in which they do.

I/O Services Provided by The OS

The kernel has an I/O Subsystem service available to applications and other parts of the OS. This is responsible for:

* Management of the name space for files and devices
* Access control to files and devices
* Operation control (e.g. a modem cannot seek())
* File-system space allocation
* Device allocation
* Buffering, caching and spooling
* I/O scheduling
* Device-status monitoring, error handling and failure recovery
* Device-driver configuration and initialisation
* Power management of I/O devices

The Life Cycle of an I/O Request

