Chapter 1. OS notes

# Operating System Definition

Operating systems need to play three different roles to run multiple applications:

1. **Referee**

An OS must manage the resources shared between different apps running on the same physical machine. Examples:

* Stop one program and start another.
* Isolate apps from each other, so bugs won’t interfere.
* Protect itself and apps from malicious viruses.
* Decide which apps get resources and when they get it.

1. **Illusionist**

Apps are written in a higher level of abstraction, therefore when we are making a program we do not want to think about the physical hardware limitations of the system. The OS provides an abstraction of the physical hardware to simplify our app design. This gives the illusion of nearly infinite memory. The OS can therefore change the amount of resources assigned to each app.

1. **Glue**

The OS “glues” together the experience between different apps. For example, cut and paste works uniformly across the system, or a file written by one app, can be read by another. Often OS gives a user interface, so apps can have the same “look and feel”. Most importantly the OS gives a layer separating apps from hardware i/o, therefore apps can be developed without thinking about the specific i/o hardware as keyboard and mouse.

These three roles explained in greater detail:

1. **Referee – resource sharing**

Sharing is an important and central part of most computers. Often an OS runs multiple apps at the same time; browsers, email, text editor and so on. The OS needs to keep all these separate and allocate the resources available to the apps that need them.

Doing multiple tasks at once is faster than just doing one task at the time. For instance, web browsers are more responsive if they can start loading the web page before all the files are received. The same thing applies to apps and the OS, which is a customer of its own abstractions.

The need for sharing raises several challenges for an OS:

**Resource allocation**

The OS must keep all simultaneous activities separate, allocating resources to each as appropriate. There is only a finite amount of hardware available in the system, therefore the OS must allocate the right amount of resource to the right app.

For example, if we run an infinite loop:

While (true) {

}

Executing this code directly on raw hardware would cause the computer to freeze, making it non-responsive to any user input. However, an OS would only give the app that is executing the code a certain amount of resources, therefore only the app would freeze not the entire computer.

**Isolation (fault isolation)**

If one app has an error, other apps and the OS should not be disrupted by it, this is called *fault isolation*. This requires restricting the behavior of the apps to less than the full power of the underlying hardware. Otherwise any script or app could take complete control over the machine. This would make it easy for apps to install spyware into OS or a bug in a program to corrupt the disk.

**Communication**

Despite the need for fault isolation, we also need communication between apps. For example, a web browser may be implemented by cooperating with a set of either apps. Since we need fault isolation and communication, we need to set up boundaries. An OS must allow the boundaries to be crossed in carefully controlled ways when the need arises.

**Summary**: One user should not be allowed to monopolize system resources or to access or corrupt other users’ files without permission. A buggy app should not be able to crash the OS or other unrelated apps, and yet apps must also work together. *Enforcing and balancing these concerns is a central role of the OS.*

1. **Illusionist – masking limitations**

To masking the limitations of the hardware available, *virtualization* is used. It provides the app with an illusion of resources that are not physically present. For example the OS can give the illusion that a single app has a dedicated processor, even though on a physical level there is a single processor shared by all the apps on the computer.

This can be taken one step further with *virtual machines*, often referred to as guest OS. A virtual machine is an application which virtualize the entire computer, giving the illusion that it thinks it is running on a real, physical machine. This illusion is provided by the OS itself.

The benefit of virtual machines (VM):

* + Older programs that only run on older versions of OS, can be run in a VM.
  + Aid debugging, developers can set breakpoints, stop the kernel, and single step their code just as they would when debugging an app.

1. **Glue – providing common services**

The third key role in an OS is providing a set of common, standard services to apps to simplify and standardize their design. The OS hides the specific of how the network and disk devices work for example, providing a simpler abstraction to the app. Therefore, the apps can focus on doing what they are supposed to do, rather than focusing on how the disk device store and handles all the data.

Another important reason for the common services is sharing. A web server must be able to read the file that the text editor wrote, therefore the file is stored in a standard format, with a standard system for managing file directories.

The graphical user interface library is also a standard service the OS provides. For example, Microsoft and apple provide a set of standard user interface widgets. This gives the different apps the same “look and feel”, such as pull-down menus and cut and paste commands.

Common service code uses the abstractions provided by the other two operating system roles (referee and illusionist). Common service takes up most of the code in an OS, but the complexity of an OS is due to the resource sharing and masking of hardware limits, therefore those are the main focus in understanding.

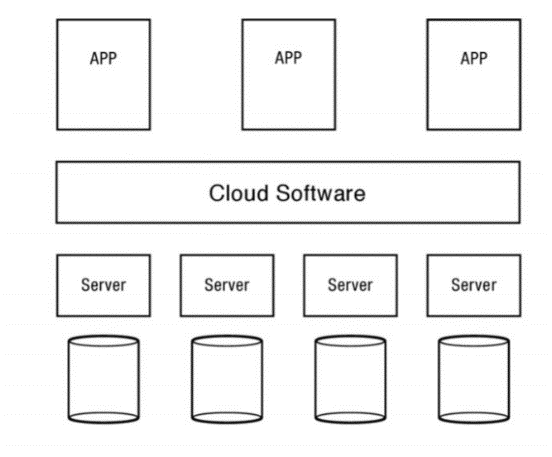
**OS System design Patterns**

These challenges that OS address applies to many different computer domains. Not only are the challenges the same, the solutions are as well. Further on some of the systems with design challenges like those in OS are explained:

**Cloud computing**

Cloud computing is where apps run on shared computing and storage infrastructure in large scale data centers instead of on the user’s own computer. This system faces many of the same challenges in OS.

1. **Referee**: How are the resources allocated between the different apps running in the cloud? How does one prevent malicious and buggy apps from interfering with other apps?
2. **Illusionist**: The hardware resources in a data center is always changing and evolving, how can you provide an abstraction layer to isolate apps developers from these changes in hardware?



1. **Glue**: Since cloud services are in data centers, work is distributed between different

machines, what kind of abstractions are provided to help services coordinate and share data between their various tasks?

Web Browsers

Many of the challenges in an OS can also be found in a web browser. Browsers have to load and display web pages, often with embedded scripting programs that the browser must execute. Therefore, the browser needs to isolate the user, itself and other web sites from malicious scripts. Also, most browsers have a plugin architecture for extensions, which must be isolated from causing harm.

1. **Referee**

Often a user has multiple tabs open in a browser, how can the browser ensure responsiveness when each of the tabs are running its own script? In other words, how does the browser share the resources available. Also how can we limit web scripts and plugins to prevent bugs and malicious script accessing sensitive user data?

1. **Illusionist**

Web services are often distributed on several geographical locations depending on the user, to improve the user experience. If a server crash or its network connection has problems, the browser can connect to a different server. How does the browser make this transition transparent to the user?

1. **Glue**

How does the browser achieve a portable execution environment for scripts that works consistently across OS and hardware platforms?

**Internet**

The internet is used by a huge number of people every day, but at the physical layer, those users share the same underlying resources. There are several malicious behaviors such as ddos (denial-of-service attacks) on the internet, that prevents legitimate users to communicate with the server. There are various attempts underway to design solutions that will let the internet continue to function despite such attacks.

1. **Referee**

Should all users on the internet be treated equally (net neutrality), or should ISPs be able to favor some uses over others? Is it possible to redesign the internet to prevent ddos, spam, phishing and other malicious behaviours?

1. **Illusionist**

The internet provides its users an illusion of a single worldwide network that can deliver a packet from any machine on the internet to any other machine. The reality is that networks are composed of many discrete elements with: 1. The ability to transmit limited size packets over a limited distance.2. Some chance that packets will be lost or ruined in the process.

The internet provides its users a facility to reliably transmit data of arbitrary length, anywhere in the world.

1. **Glue**

The internet protocol suite was explicitly designed to act as a interoperability layer that lets network apps evolve independently of changes in network hardware and vice versa.

# Operating System Evaluation

Desirable criteria for OS:

* **Reliability and availability**: do the OS do what you want?
* **Security**: Can the OS be corrupted by an attacker?
* **Portability**: Is the OS easy to move to new hardware platforms?
* **Performance**: Is the UI responsive, or does the OS impose too much overhead?
* **Adoption**: How many other users are there for this OS?

1. **Reliability and availability**

*Reliability*: a system does exactly what it is designed to do.

The OS is the lowest level of software running on a system, therefore a OS error can have devastating and hidden effects. For example if an OS breaks or run into an error, you may not be able to get work done, or previous work is lost if the OS corrupts files on the disk. However, in contrast, apps are more forgiving when it comes to errors and crashes. The OS provide fault isolation and a rapid and clean restart.

*Availability*: the percentage of time that the system is usable.

Availability is affected by two factors: MTTF and MTTR. Availability can be improved by increasing MTTF, or reducing MTTR.

* MTTF: the frequency of failures measured in (mean time to failure).
* MTTR: The time to restore a system to a working state after failure (e.g. reboot). Measured in (mean time to repair).

A buggy system that crashes frequently, losing the users work, is both unreliable and unavailable. A system that crashes frequently, but never loses the users work and cannot be attacked by malicious malware, is reliable but unavailable. And last a system that has been attacked by malicious malware, but appear to run normally is unreliable but available.

Thus, both reliability and availability are desirable.

1. **Security**

*Security*: the computers operation cannot be compromised by a malicious attacker.

*Privacy*: an aspect of security: data stored on the computer is only accessible to authorized users.

No useful computer is perfectly secure!

* Any complex software has bugs, which can be exploited by an attacker.
* Computer hardware can be tampered with, to provide access.
* The computers administrator might be untrustworthy.
* OS developer might insert a backdoor to gain access to the system.

An OS should always be designed to minimize it vulnerability to attack. Even though an OS might have strong fault isolation, the system will not be secure if the apps are not designed for security. A good example of this is the internet email standard. It provides no strong assurance of the sender’s identity since it is possible to include any email address in the “from” field.

If the fault isolation would be by itself sufficient, apps will not have the ability to share and interact with each other. Therefore, we do not only want to isolate programs from one another, but to easily share data between programs and between users. Thus, an OS need both an enforcement mechanism and a security policy.

*Enforcement*: how the OS ensures that only permitted actions are allowed.

*Security**policy*: defined what is permitted. Who can access what data, and who can perform what operations.

A fault in either of these will be a vulnerability to the system. If there is an error in the enforcement, it could allow an attacker to evade the policy. And if there is an error in the policy, it could allow an attacker to access when he should have been prohibited.

1. **Portability**

*Portable*: abstraction that does not change as hardware changes.

A program written for win8 should run correctly regardless of hardware changes, e.g. different graphics cards, different types of memory, type of network. This portability also applies to the OS itself. Since OS are among the most complex software system ever invented, it is impractical to rewrite them for hardware changes, or new apps. Instead OS are derived from old ones, as IOS was derived from MacOS X code base.

OS lifetime are therefore often measured in decades. OS achieve this portability by using a simple, standard way for apps to interact with the OS, called the *abstract virtual machine (AVM)*. This is the interface provided by the OS to the apps:

1. API (application programming interface): the list of function calls the OS provides to apps.
2. The memory access model
3. Which instructions can be legally executed. For example, instruction to change whether the hardware is executing trusted or untrusted code must only be accessed by the OS

The portable hardware abstraction is so powerful that the OS uses it as well internally. The OS can be largely implemented independently of the hardware specifics. The interface that makes this possible is called the *hardware abstraction layer (HAL).* Even though it might seem that the AVM and HAL are identical the AVM must do more. Apps execute in a restricted, virtualized context, while the OS itself uses a procedural abstraction much closer to the actual hardware.

1. **Performance**

The design of the OS can greatly affect the apps perceived performance. The OS decides when an app can run, how much memory it can use, whether its files are cached in memory or clustered efficiently on disk.

Performance can be measured in several different ways:

*Overhead*: the added resource cost of implementing an abstraction presented to apps.

*Efficiency*: the lack of overhead in an abstraction.

Another issue is *fairness*. Should the resources be divided equally between users or apps, or should some get preferential treatment? If so, how does the OS decide what tasks get priority.

*Response time / delay*: how long it takes for a single task to run, from start to finish.

* An OS with a good response time can be an OS where the time from the user moves the hardware mouse until the pointer on the screen moves is very low. An OS with poor response time can be unusable.

*Throughput*: the rate at which the system completes tasks. A measure of efficiency for a group of tasks rather than a single task. Improving response time does not also imply that the throughput is improved, as discussed in chapter 7.

*Performance predictability*: whether the systems response time or other metric is consistent over time.

* Can be more important than average performance.
* For example:
  + System 1: keystroke is instantaneous, but 1% of the time takes 10 seconds.
  + System 2: keystroke takes exactly 0,1 seconds 100% of the time.
  + System 2 is more user friendly despite the slower average performance!

1. **Adaption**

Two factors outside the OS immediate control are important for success:

* The wide availability of apps ported to that OS.
* The wide availability of hardware that the OS support.
  + For example an iPhone runs IOS, but with the pre-installed apps and the contents of the app store, the iPhone would be just another cellphone.

The network effect: the value of some technology depends on the number of people who have adopted it.

* Apps and hardware designer spend their efforts on the OS platform with the most users.
* Users favor the OS with the best apps and cheapest hardware.
* More users imply more apps and cheaper hardware, which implies more users, in a virtuous circle.

When designing an OS, it is important to take the network effect in consideration, to take advantage of it or avoid being crushed by it.

*Proprietary system*: A system under control of a single company, it can be changed at any time to meet the needs of its customers.

*Open system*: A system with open and public source code, which gives anyone the ability to inspect and change the code.

* Windows 8 and MacOS are proprietary OS.
* Linux is an open OS.

**Design tradeoffs**

Most practical OS designs strike a balance between the goals of reliability, security, portability, performance and adoption.

# Operating Systems: Past, Present and Future

**Early Operating Systems**

The first OS were runtime libraries intended to simplify the programming of early computer systems. These computers often took up an entire floor of a warehouse and cost millions of dollars. Despite this they were only capable of running one single program at a time, which had to be loaded into the system bit by bit.

The first and foremost need for OS was to reduce programmer errors. For example, an OS would provide a standard I/O routine, which made it more likely that a user’s program would produce a useful output.

**Multi-User Operating Systems**

The next step with OS was sharing, which poses many of the same challenges that we see in todays OS. Processor time is valuable; therefore, the processor should not be idling when loading programs as in the early days.

The new OS at the time were called *batch operating systems*. A batch OS works on a queue of tasks, it runs a simple loop: load, run and unload each job in turn. While a job/task was running, the OS would set up the I/O devices using a process called direct memory access (DMA).

These batch OS were extended to run multiple apps at once, called multitasking or multiprogramming. Multiple programs were loading into memory and was ready to use the processor if any previous task needed to pause to for example read I/O. This increases processor efficiency to nearly 100%.

**Time-Sharing Operating Systems**

After the cumulative effect of Morre’s Law mean that cost of computing dropped to where systems could be optimized for users rather than for efficient use of the processor. UNIX was developed in the early 70s, which became the basis for MacOS, Linux, VMware and android.

Time-sharing means that for example when the user types an input on a keyboard or mouse, it will cause an interrupt to the processor signaling the event. It is then queued, processed and fetched by for example a word processor. Hundreds or even thousands of such events can be processed per second.

**Modern Operation Systems**

Today we have diversity of computing devices, with many different OS running on them.

* Desktop, laptop and netbook OS
* Smartphone OS
* Server OS
* Virtual Machines
  + A VM monitor is an OS that can run another OS.
* Embedded systems
  + Computers integrated into consumer devices, such as: microwave ovens, automobile, airplanes, LEGO robot and medical devices.
* Server clusters

**Future Operating Systems**

If we rely on computers for critical systems such as: power grid, the telephone network, medical devices and automobile/traffic control, we need better security and reliability.

The future of OS is also the future of hardware:

* **Very large-scale data centers**: OS will need to coordinate the hundreds of thousands or even millions of computers in data centers to support essential online services.
* **Very large-scale multicore systems**: Computer architecture already contain several processors per chip; this trend will continue, yielding systems with hundreds or possibly even thousands of processors per machine.
* **Ubiquitous portable computing devices**: The OS on smartphones, tablets and e-book readers are untethered from the keyboard and the screen, therefore we will se an increase in the OS responding to voice, gestures and perhaps even brain waves.
* **Very large-scale storage**: All data that can be stored, will be. The OS will need to store enormous amounts of data reliably so that it can be retrieved at any point, even decades later.