An introduction to operating systems

1 Introduction

There is no single definition of operating system. Operating systems exist because they are a reasonable way to solve the problems created by a computer system. The hardware itself is not easy to use; therefore it is necessary to help both the user and the programmer by abstracting the hardware complexity. The way to do this is by placing a layer of software above the hardware in order to present the user of the system and the applications a **virtual machine interface** that facilitates the understanding and use of the system. This software layer is called the operating system. The operating system includes a set of functions to control the hardware that are common to most applications, e.g., functions controlling the devices and interrupt service routines, hiding to the programmer the hardware details and offering an interface comfortable to use the system.

From another point of view, the operating system must ensure a proper and efficient system functioning. A computer system now consists of a large number of components that need to be managed. Throughout the history of computers there has been a significant development affecting the various components of the system. This evolution has been achieved both in the technological aspect (from valves and relays to VLSI circuits) and at the architectural level (different techniques to increase processor speed, memory hierarchies...) and in the field of programming languages (libraries, languages, interfaces...). This evolution was constrained by requirements of efficiency and ease of use of computers. However, it should be noted that the increased efficiency of each system component does not ensure an increase in the overall system efficiency. Indeed, the **tuned management of all resources** will be largely responsible for the success or failure. From this perspective, the operating system is responsible for providing an ordered and controlled allocation of different resources (processor, memory, Input/Output devices...) to each of the programs competing for them.

Metaphor of the driver and the mechanic

In any system it is important to distinguish between interface and implementation. The user of a system must know its interface, but how it is implemented is a matter of the designer or the maintenance staff. The user of a car only needs to know the interface so that the vehicle is helpful. So, he must learn to manage the steering wheel, turn signals, lights, accelerator and brake. To make things easier, manufacturers tend to standardize the interface: the accelerator is a pedal that is always located in the same place; the direction "right" is always represented in the controls as a turn in the clockwise direction... Since the system is not perfect, the user must perform some "management" tasks: if the car is not automatic, he must choose the right gear, when the tank is empty, he must fill it with a given type of fuel... However, these tasks tend to be increasingly limited. A century ago, the user of the car used to have a driver-mechanic, since the cars were very unreliable, and should be started manually by operating the starter motor from the outside. Today, one can be a good driver without having knowledge of mechanics, and many drivers ignore, for example, that the car has an electric motor for starting. The mechanics are in charge of maintenance, knowing perfectly the internal structure of the car, but they do not have to be good drivers: they might not even know how to drive.

We can say that the concept of operating system is linked to two different ideas. For a user/programmer, an operating system is the set of functions that allows him to use the resources of the machine obviating the characteristics of the hardware. This is the *functional vision* of the operating system, which allows one to view the system as a *virtual machine*. This is the vision that this course will deepen. For a system designer, however, an operating system is the software that, installed on the bare machine, allows controlling its resources efficiently. This view corresponds to the operating system *implementation*.

Both views largely refer to the same concepts and terms, but their approach —and their objectives— are different. In this introductory course to operating systems we will study the

functionalities offered by operating systems in general, as well as the basics of how the operating system supports them. The techniques and fundamental models of the design of operating systems, as well as the concepts and tasks of system and network administration, including security management, are studied in courses of the Computer Engineering specialization.

2 Functional vision of operating systems

Of the two approaches presented out above, this is the less clearly defined and developed in the literature. Perhaps this is due to the fact that historically it has been the interface programmer who designed the interface functionality, and he does not feel particularly inclined to discuss the specific services that the interface must provide. Hence, often services are later added or modified according to the needs in revisions. In the operating system, in addition, this interface is not unique, in the sense that, besides the set of **system calls** (primitives of the operating system) provided to applications, it can be considered —historically it has been— the **shell** as part of the operating system, and even, by evolution, the graphical user interface (GUI).

In what follows, we will consider the *system call interface* as the basic interface of the operating system, which defines the system as a *virtual machine*. The set of system calls of an operating system describes the interface between applications and the system and determines the compatibility between machines at the source code level.

The end user sees the computer system in terms of applications. Applications can be built with a programming language and are developed by application programmers. If we had to develop applications taking care at all times of the control of the hardware they use, application programming would be a daunting task and probably we could not enjoy sophisticated applications as the ones we have nowadays. In addition, applications also take advantage of a set of **tools** and services that facilitate even more the work of the programmer, as editors, compilers, debuggers… Here we also include **libraries** of functions that are available to applications (mathematical functions, graphical…). Typically, these services are not part of the operating system. Figure 1 presents a summary of this approach.

Interfaces and interfaces

Perhaps the most frequent professional vice of computer engineers is to not properly differentiate the various system interfaces. One can just take a look at any application or operating system to realize it. For example, one can find in "Accessories", together with a calculator or a player and a sound recorder, tools like "disk defragmenter". Surprisingly, in "Control Panel" one can find an application to read text aloud. It's like if a car manufacturer had placed a wrench on the dashboard, next to the hole for sunglasses, and the radio under the hood, near the engine. Not surprisingly, many home users hate computers. "Computing is very complicated", they say. Well, so is mechanics. However, no driver feels unsafe while driving by not knowing how to use a wrench. In this sense, one could say that current operating systems are like cars of a century ago.

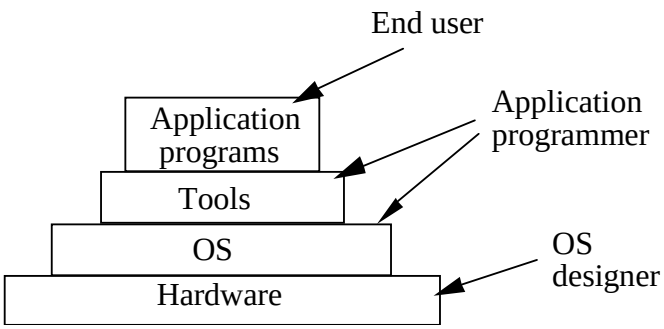

***Figure 1**. Layered structure of a computer system.*

3 Functions of an operating system

In general, and regardless of the type of interface, operating systems typically provide a set of functions that can be summarized as follows:

- **Program execution**. Running a program requires a number of tasks. Instructions and data must be loaded into main memory, files and I/O devices must be initialized… The operating system performs all these tasks.

- **Control of I/O devices**. Each device requires its own set of instructions and control signals to operate. The operating system takes care of all these details so that the programmer can see the access to the devices as simple reads and writes.

- **Access to files**. Historically we have used the concept of a file as the permanent representation of a set of information with a global name in the system. Files reside in nonvolatile memory such as disks and flash drives. Besides the nature of the device, the operating system has to manage the file format and the way of storing.

- **System access control**. For multi-user systems, the operating system has mechanisms to control the access to system resources based on the rights defined for each user.

- **Detecting and responding to errors**. When a computer system is in operation it may fail. These errors can be hardware (memory or device access error), or software (arithmetic overflow, attempt to access a forbidden memory position...). In many of these cases the system has hardware components to detect these errors and to communicate to the operating system, which should give a response that eliminates the error condition with the least possible impact on the applications that are running. The answer may go from the ending of the program that caused the error, to retrying the operation or simply reporting the error to the application.

- **Accounting**. It is common for an operating system to provide tools for tracking operations and accesses, and for collecting data regarding resource usage. This information may be useful to anticipate the need for future improvements and to adjust the system so as to improve its performance. It can also be used for billing purposes. Finally, upon a security issue, this information can be used to discover the attacker.

4 Operating system interfaces

In a system structured in layers, a layer L_k provides an **interface** to the upper layer L_{k+1}, represented by a set of functions which determine how layer L_k is accessed from layer L_{k+1}. The implementation of layer L_k is independent of the interface and is said to be *transparent* to the layer L_{k+1}, in the sense that when designing the layer L_{k+1} there is no need to worry about how layer L_k is implemented. An interface must specify precisely the functions offered and how they are used (arguments, return values…).

Generally, an operating system offers three different interfaces:

User interface. When there were no graphics terminals like the ones we have nowadays, the user had to communicate with the system by typing commands that allowed running programs, consulting directories… To do so, the operating system offered a specific utility, the command interpreter (*shell* in Unix terminology), whose interface was presented as a set of commands whose

usage form was (or should be) well specified in a manual (for example the Unix *man*, Section 1). Nowadays graphical user interfaces greatly facilitate user interaction by means of intuitive concepts and objects (icons, pointers, mouse clicks, drag and drop...). If in the case of shells each system offered its own shell (the user had to learn to use it, usually attending a course), the graphical user interfaces are common and intuitive enough so that their use is available to everyone.

Administration interface. The administrator of a computer system is the person in charge of installing the system, maintain it and manage its use. In a system composed of several computers, this work includes managing user accounts and network resources, with special attention to the care of user privacy and information security. The system administrator is a professional who knows the specific tools and functions that the system offers for it and that can only be used by him, as they require special privileges. Overall, he relies for it on an extension of the shell (for example, in Unix, specified in Section 8 of *man*), although the use of these tools does not exclude the use of the graphical user interface. Instead, a personal system should not require, ideally, management effort by the user, since he is not supposed to be an expert for it, like the driver of a car is not required to have mechanical expertise. The reality is that, like a car driver should know how to change a wheel, a computer user has to solve nowadays some management problems arising from the immaturity and imperfection of operating systems.

Programming interface. To develop applications on an operating system, the programmer uses, regardless of the programming language used, a set of functions to access operating system services, the system call interface. These functions do not differ in appearance from other library functions provided by the language. However, calls to the operating system are specific to that system and therefore probably incompatible with those of another operating system, since they refer to objects and concepts specific to that system. Actually, it is common that the programmer does not directly use operating system calls, but specific library functions of the language for that purpose. For example, if the C programming language is used, the programmer uses the *printf* function to output data, regardless of the operating system he is using. However, *printf* is a function implemented in terms of calls to the operating system (in the case of Unix, the *write* system call), so that the code generated is specific to that system. This, in general, is not taken into account by the application programmer, but it is by the library developer, a systems programmer, who is the user of the operating system call interface and will therefore rely on the corresponding specification (in Unix, Section 2 of *man*).

APIs

*Nowadays programmers use to talk about API (*Application Programming Interface*) to refer to the set of functions available in a platform for application development. An API can be the set of system calls extended with other library functions, though system calls themselves are usually hidden by library functions that facilitate programming. There may also be specific APIs tailored to specific applications. Ultimately, an API depends on the programming language and operating system for which this API is implemented.*

*In the Java world, since it is an interpreted language, the APIs are independent of the operating system: it is the virtual machine (*JVM*) which interprets the library functions for the underlying operating system.*

5 Evolution of operating systems

From the perspective offered by the already relatively long history of operating systems, and considering its application fields, now we can talk about different models of computation, which determine the functionality of an operating system, and sometimes its structure:

Batch systems. The earliest operating systems (1950s) were called **monitors**. The users gave their program with the input data in a stack of punch cards (a lot) to the computer operator who sequentially ordered lots and placed in a card reader. Each batch included control cards with orders

for the monitor. The last card was a return order to the monitor that allowed it to start automatically loading the next program.

Multiprogrammed systems. The price of a CPU at that time was exorbitantly high, so it was intended to work 100% of the time, which is unattainable with batch systems, since the processor, when executing an I/O instruction, should wait for the device, very slow compared to the processor speed, to complete the operation. This led to the engineers of the time to devise strategies for a more efficient use of the CPU. By loading multiple programs in memory, when a program needed to wait for an I/O the processor could execute another program. This technique, known as **multiprogramming** or **multitasking**, was developed in the mid 1960s and is the basis of modern operating systems.

Time-sharing systems. At that time new applications appeared requiring an operating mode in which the user, sitting at a terminal, interacted directly with the computer. This operating mode, **interactive**, is essential, for example, in the processing of transactions or queries. The interactive processing requires, of course, multiprogramming, but must also provide a **response time** (time elapsed from the ordering of a transaction until the answer is obtained) reasonably short. That is, the user that interacts from a terminal can not be waiting for long because some program, **aimed at calculation**, does not leave the CPU for not executing any I/O for a while. For this reason, in **time-sharing** systems, introduced in the second half of the 1960s, the operating system runs the programs in short bursts of computation time (*quantum*), in an interleaved way. Thus, if there are n programs loaded in memory, each program will have in the worst case (when no program required I/O) $1/n$ of the processor time. Given a *quantum* small enough and a not too big n, the user does not observe a significantly long response time for his program and has the feeling of being using a dedicated processor with a speed $1/n$ of the actual processor. This idea is known as **shared processor**, and reflects the ideal behavior of a time-sharing system, minimizing the response time.

Teleprocessing systems. In the first time-sharing systems terminals were connected to the processor by means of specific wiring that was installed in the building. When large companies and institutions (e.g., banks) began buying computers, they found the need to transmit information between their branches and the computer at the headquarters. Fortunately, there already existed the telephone wiring, which was used to transmit digital information using a modulator-demodulator (*modem*) at each end, connected to the conventional telephone line. Unlike the transmission using special wiring, telephone communication is very prone to errors, so it was necessary to develop more sophisticated communication protocols. These protocols were, initially, proprietary (owned by the computer manufacturer, which was also the one who supplied the terminals, modems and software).

Personal systems. Cheaper hardware and the advent of the microprocessor in the late 1970s made it possible to provide a dedicated system for a single user at a reduced cost, a key feature of a personal system. The operating system for personal computers is, at first, **single-user** (no protection mechanisms) and **single-tasking**, that is, not very different from the primitive monitor-based systems except for the fact that it is used

A question of price

It is necessary to look at the evolution of the cost factor regarding technology to understand the path followed by the system management models. Before the development of integrated circuit technology, a computer costing millions of dollars, was composed of tens or hundreds of thousands of individual electronic components (transistors and, previously, valves), weighed several tons and occupied a large and heated room. However, their performance in terms of processing power and storage were comparable to those of the chip in a smart card of today. It can be understood then that in the 1960s, engineers started the development of operating systems with multiprogramming and virtual memory, able to take full advantage of these machines (IBM/360's basic configurations, the most popular mainframe of that era, came with 8 Kbytes of memory and executed a few thousand instructions per second, yet the CPU was very fast compared to the punch card reader). Today, operating systems still include virtual memory, but most personal computers do not need to use it.

interactively through a terminal. Today the available hardware allows multitasking personal systems (Mac OS, Windows, Linux) supporting sophisticated **graphical user interfaces**.

Networked systems. With the advent of the personal computer the terminals of teleprocessing systems are replaced by PCs that can take certain computing tasks, downloading the central time-sharing system. In particular, PCs can execute any communication protocol. With the adoption of standard protocols (e.g., TCP/IP), personal computers can communicate with each other: there is no *one* central computer, but a set of computers that are connected together. If a computer in the network provides access to a particular resource, then it is the **server** of that resource. The remaining computers, **clients**, access the remote resource using a **client-server** protocol. Managing access to networks has complicated the operating system and has led to the emergence of services that are deployed on it (known as *middleware*), resulting in **distributed systems** that are deployed today in the field of Internet and have generated concepts and schemes very sophisticated, such as *Web services*, *peer-to-peer* and *cloud computing*. Although this course is restricted to the study of centralized systems, we must not forget that the reality is more complex.

Mobile systems. The evolution of hardware does not end with personal computers. These are becoming smaller, which, together with the use of a battery and a wireless network, provides autonomy and makes them mobile systems. In principle, this change does not significantly affect the operating system. However, with the new century and by means of the evolution of mobile telephony new devices with increasing computing capabilities have appeared. These devices, now called *smartphones*, are capable of supporting smaller versions of operating systems designed for personal computers (Mac OS, Windows, Linux), although there are also specific operating systems (as Symbian, or Google Android) with great performance, including new forms of interaction (touch screens, cameras, positioning information...) and new applications (such as navigation). This field is undoubtedly the hottest area for the development of current and future technology of operating systems and extends to very different types of devices (e.g., cameras, smartcards, or control devices *embedded* in appliances or cars...), which are capable to network and interact spontaneously with each other even without human intervention.

Groundhog Day (1993)

The long history of operating systems has followed a cyclical path. It is surprising to learn that sophisticated concepts and complex techniques to implement such as multiprogramming and virtual memory have almost half a century and were part of the first time-sharing systems. When, fifteen years later, personal computers appeared, the first operating systems developed for them dispensed with these mechanisms because their limited hardware could not support them. In fact, apart from the interactive operation mode, they were not very different from the primitive monitors. However, as the hardware of personal computers gained in performance, their operating systems were integrating these techniques. So, while at the time they distinguished between mainframes, workstations and personal computers, any computer today is capable of supporting a complex operating system. More recently, miniaturization has led to the emergence of small devices (mobile phones, smartphones, are the most notable example) with increasing computing and storage capacity. Again, history is repeating itself: if the first operating systems for mobile phones were extremely simple, there are already smaller versions of general purpose operating systems aimed at mobile phones, which are integrating features like multitasking.

6 A classification of operating systems

When classifying current operating systems one can take into account different criteria, derived from the concepts introduced above. One possible classification is based on the following criteria, which can be combined: (1) if the system can run at the same time one or more than one program, (2) if it supports the connection from a single terminal or from more than one, and (3) whether it supports a single user or can manage more than one user.

(1) Monoprogrammed/multiprogrammed. They are also known as **single-tasking/multitasking**, terms that we will consider synonymous. In the primitive operating systems, both monitors as the first systems for personal computers, for example MS-DOS, the execution of a program had to finish for the start of the next program. These systems are called monoprogrammed (single-tasking). From 1965 there appeared the first multiprogrammed systems (OS/360, Multics). Today, virtually all operating systems are multiprogrammed (multitasking). In multiprogrammed systems, several programs run **concurrently**, i.e., interleaving their executions over time, which are perceived as simultaneous. They use the concept of **process** (or task) to designate a running program. As stated above, multiprogramming was motivated by the need to optimize processor usage, and therefore running processes in a multiprogrammed system usually represent independent applications. Later multiprogramming has been used to express concurrency in the same application, where a set of tasks cooperate in a coordinated manner. For example, in a word processor we can find a task in charge of reading and processing keyboard input, another task in charge of checking the spelling, a third task responsible for periodically saving changes… A particular class of multiprogrammed operating systems is the **multithreaded** systems, which allow expressing the concurrency in an application more efficiently. The difference between a process and a *thread* (also called *subprocess*) is, for our purposes, very small, and we will not address it at this time. Thus, multiprogramming means multiplexing the processor among processes, as explained above. Obviously, a multiprocessor system (a computer with multiple processors) enhances further the multiprogramming by allowing the concurrent execution of programs to be also **parallel**. This is known as **multiprocessing**, and operating systems that control these systems are called **multiprocessor operating systems**. Although there are significant differences in the implementation of a multiprocessor operating system with respect to a single-processor operating system, with respect to the functional vision of applications and users they hardly transcend.

(2) Single-terminal/multiterminal. An operating system ready to be connected simultaneously from different terminals is said to be multiterminal, otherwise it is said to be single-terminal. Time-sharing operating systems, such as Unix, are multiterminal. An operating system designed for personal computers —MS-DOS, Windows 95/98— is, naturally, single-terminal. It is noteworthy the case of Linux, a Unix system for personal computers, which maintains the multiterminal Unix philosophy by means of a set of virtual terminals. Mac OS X, also derived from Unix, is another multiterminal example. It is clear that a multiterminal system must be somehow multiprogrammed: as we shall see, it is common that each terminal (real or virtual) has an associated process that manages the connection.

(3) Single-user/multiuser. A multiuser system is able to provide user authentication and includes policies for managing user accounts and access protection, providing privacy and integrity to users. In the primitive monitor-based operating systems, shared by several users, this function was carried out manually by the system operator. The first operating systems for personal computers, such as MS-DOS, were single-user. The general purpose operating systems of today are multiuser. Note that some personal systems, such as mobile phones, include some verification mechanism (usually a password), but lack of policies to protect accesses to system resources and user management; they simply authenticate *the* user, but are in all aspects single-user.

7 The operating system market

From a closer perspective to the business world, we must refer to two groups of operating systems. First, those operating systems that have been designed by a manufacturer for a specific architecture in order to protect their products (both software and hardware) for potential competitors, which are called **proprietary operating systems**. The manufacturer designs the operating system specifically

for the architecture, and provides the necessary updates. Even sometimes the specification of its system call interface is not made public or is constantly changing, making difficult the development of applications by other manufacturers. This creates a closed world that encompasses the architecture, the proprietary operating system and the applications, enabling the control by the manufacturer of the market for their products and establishing big dependencies for customers. Some examples of proprietary operating systems, largely deployed, are (or have been) IBM systems, Digital VAX VMS, Apple Mac systems, and Windows systems of Microsoft for the PC platform.[1]

With the advent of Unix (circa 1970) a new philosophy arises: since it is written almost entirely in a high level programming language (C), the operating system is **portable** to other architectures and therefore so are the applications at the source code level. Furthermore, in the case of Unix, its source code was freely distributed. This had contradictory effects: on the one hand it contributed to the wide dissemination of the system; on the other hand, each manufacturer introduced their own modifications not only in the source code but also in the system call interface, so that you have to refer to different Unix systems, not fully compatible with each other (System V, BSD, AIX, Ultrix, Solaris, Linux...). We can say that the family tree of Unix is really complex.

The ideal consisting of a world of **open systems**, with public specifications, accepted and standardized, allowing full portability of applications (and users[2]), is a goal rarely achieved. In this regard, there have been efforts to define standard specifications. For example, the POSIX specification is a reference in the Unix world. A developer that follows in the system calls of its program the POSIX specification knows that he can compile and run it on any Unix system that follows the POSIX standard.

In this sense, it would be useful that operating systems were designed with the ability to support different system call interfaces. This was the philosophy of microkernels in the 1980s, which implemented the system call interfaces as services *outside* the operating system itself. However, the development of microkernel-based operating systems has had a limited commercial impact. The best known is the Mach 3.0 microkernel, on which the Mac OS X operating system from Apple relies. However, the most common approach today is to support applications of heterogeneous systems through *emulation* (virtualization) of other operating systems on a host operating system. There are numerous virtualization programs, e.g., VMware, Virtual PC, or Win4Lin.

It should be noted a phenomenon that revolutionized the market of software in general and operating systems in particular: the spontaneous emergence of a community of programmers who develop **free software**[3]. Internet is the necessary way for sharing and exchanging code and ideas rapidly in the community. As a result, and this has been amply demonstrated, the software adapts very dynamically to particular problems, the development of new products is very fast, and errors are corrected and versions refined with great agility. Organizations like GNU[4] grant a license to copy, modify and redistribute free software with the condition that the new distribution includes the source code.[5] Linux is today a settled example of this philosophy.

[1] Still and all, there are important differences between proprietary systems. For example, Microsoft had the success in the 1980s to *open* its software platform (the interface of MS-DOS) to other developers.

[2] This means that the user does not “miss” the interface when changing the system.

[3] Not to be confused with *freeware*.

[4] http://www.gnu.org

[5] This license is called *Copyleft*.

On another level it should be noted that as computer technology was occupying new application areas, niche markets for new types of operating systems have been developed. A remarkable example is the market of **real-time operating systems**, for long time common in industry (control systems), and more recently in other areas (e.g., video decompression in a multimedia system). In real-time systems response times are limited by a deadline. After the deadline, the response is invalid and can even be catastrophic (think about the stability control of a car). Often these types of systems are **embedded** in more complex systems (for example, the control of the stability in a vehicle). Although general purpose operating systems (such as Windows, Linux or Mac OS) allow running certain noncritical real-time applications (like a video cassette recorder, VCR), there are specific real-time operating systems (e.g., QNX, FreeRTOS and many others). Many general-purpose operating systems also support real-time tasks, but are only suitable when missing the deadline is not critical (e.g., multimedia applications).

Winners and losers

In the early days (50s and 60s of the twentieth century), the operating system was developed in machine language by the manufacturer of the architecture, which distributed the system as an indivisible package. The operating system and the architecture were absolutely interdependent. Later, after the experience of Unix and the C programming language, software and hardware manufacturers specialized, allowing, in principle, the operating system to be easily transported to different platforms (the core of Unix contained 1000 lines of machine code, dependent of the architecture). As a consequence, the architecture could support different operating systems. However, the introduction of personal computers made evident the need of some form of standardization of operating systems, from the interface for applications to the user interface. Standardization came by way of the facts from two factors: the strategic alliance between IBM and Microsoft, and the opening of the hardware (PC) and software (MS-DOS interface) platforms to other manufacturers. This was at the expense of Apple, the major competitor of Microsoft, which started in the 80s with an undoubted technological advantage, but closed its platform to their own products. As the PC architecture was conquering markets, Microsoft systems conquered the OS market. The emergence of Linux and the philosophy of free software in the 90s occurred too late to respond to the monopolizing inertia of Windows systems. The early history of the personal computer can be found in the book Fire in the Valley: The Making of a Personal Computer*, by Paul Freiberger and Michael Swaine, brought to the screen by Martyn Burke with the title* The Pirates of Silicon Valley*.*

At present operating systems, beyond its original orientation, have had to be adapted to a multitude of devices, such as mobile phones and other consumer devices. To this we must add the **embedded systems**, increasingly present in our environment (appliances, cars, industrial plants, robots…). Typically embedded systems are subject to physical constraints and have real time requirements, sometimes critical, leading to specific solutions, as already mentioned. With respect to the world of mobile devices (smartphones or tablets), in some cases conventional operating systems have been adapted to the constraints of the devices (size and power), such as Microsoft Windows Mobile, Apple iPhone OS or Palm OS. In other cases specific systems have been developed, such as Symbian OS or Google Android.

8 Examples of operating systems

We will discuss here in more detail the history and main characteristics of the more relevant operating systems, in line with the concepts introduced in the previous sections. We will focus on those families of operating systems that have made history in computing and whose innovations, directly or indirectly, remain today.

IBM mainframe operating systems

IBM was for many years the dominant computer company in the market for hardware, operating systems and applications. His first major operating system, OS/360, whose development ended in 1964, was a complex batch multiprogramming system that stored tasks in partitions (of fixed or variable size, depending on the version). One version, TSS/360 (*Time Shared System*, 1967),

offered time-sharing and multiprocessing (with two CPUs), although its enormous complexity (all systems at that time were developed in assembler) caused that it never worked too well and that its spread was low.

MVS (*Multiple Virtual Storage*, 1974) provided virtual memory. It introduced the concept of virtual machine, which allowed running multiple copies of the operating system into independent logical partitions, providing a high degree of safety. The MVS architecture has survived and today is part of the z/OS system.

VMS from Digital

By 1970 the introduction of integrated circuits had cheapened significantly the cost of computers and expanded its area of use. It appeared the concept of minicomputer to designate a range of affordable computers (on the order of tens of thousands of Euros) with a small size (like a small closet). At that time, Digital Equipment Corporation triumphed with its PDP minicomputer family. The PDP-11, of 16-bit, was the culmination of the saga. It worked with the RSX-11 operating system, designed to support real-time applications.

The inherent limitation of the 16-bit architecture led Digital to introduce in 1977 the VAX-11 architecture (*Virtual Address eXtension*), of 32 bits, and the VMS operating system VMS (*Virtual Memory System*). One of the features of VMS is its adaptability to the diverse hardware support level of the different implementations of the VAX architecture, especially regarding virtual memory. Another feature is that the file system manages file versions, identified by a suffix denoting the version which is part of the file name. It has a sophisticated process scheduling policy based on dynamic priorities. Many of the ideas present in VMS were adopted in the development of Microsoft Windows NT. In 1991 VMS was renamed OpenVMS for the Alpha architecture, the successor of VAX.

The Unix family

In 1970 at Bell Laboratories of AT&T they started to develop a Unix system, which would have a great impact and subsequent development. Their ancestors were the CTSS and Multics systems. The latter, although not commercially successful, set the way for future operating systems. Unix, whose first version was developed in assembly language on a PDP-7, was entirely rewritten in 1972 in C (language developed at Bell Labs specifically for the Unix project), being the first operating system written in a high level language. In 1974 there was already a public description of it.

AT&T distributed Unix freely; so many universities and companies used it for their computers and developments. Since the source code was made public, Unix has a lot of ramifications (Digital's **Ultrix**, Microsoft's **Xenix**, IBM's **AIX**, HP's **HP-UX**...), but basically there are two families: ***System V*** from AT&T and ***BSD*** from the University of Berkeley, whose most popular version was marketed by Sun Microsystems. While the latter is more powerful with regard to network support, the two families were unified in the *System V Release 4 (SVR4)*, which in Sun's version was called **Solaris**.

Unix versions are also available for PCs, being the most popular **SCO** or **Santa Cruz** among the commercial versions, and **Linux** and **FreeBSD** among the freely distributed. Linux is a project initiated by Linus Torvalds in the University of Helsinki in the early 1990s, which proposed free operating system software in the line of GNU (General Public License) and *Free Software Foundation* in the field of applications. Linux is having a huge success not only in small servers, but also in large machines. Its introduction into the market of personal computers is increasing,

thanks to major advances in three areas: ease of installation, friendly graphical environments, and a growing number of quality office applications.

Figure 2 shows, in a simplified form, the Unix family tree.

Unix is multiprogrammed, multiuser and multi-terminal, and supports various interfaces both alphanumeric (shell, C-shell, K-shell...) and graphical (Openwin, Motif, KDE, Gnome...). The latest versions support even multiprocessing.

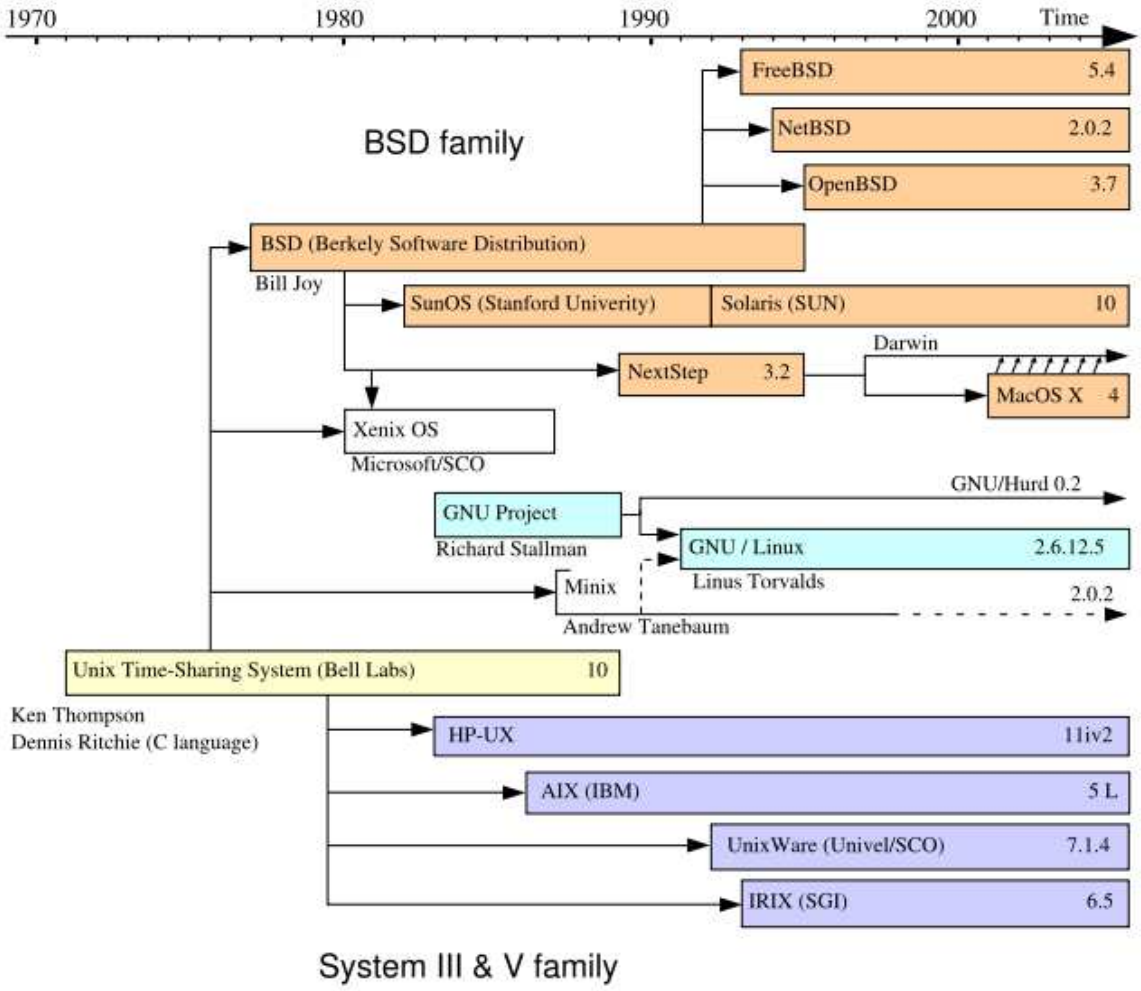

***Figure 2**. The Unix family (Source: wikipedia).*

Microsoft: from MS-DOS to Windows NT

When IBM decided in 1980 to enter the world of personal computing, they proposed Microsoft the development of a new operating system for their new PC. Thus, in August 1981 IBM markets its first PC with **MS-DOS** as operating system. MS-DOS 1.0 was compatible with CP/M, the operating system used by most existing microprocessors until then, but also had significant improvements on it. It had more information about each file, a better allocation algorithm for disk space, and was more efficient. However, it could only contain a single directory of files supporting a maximum of 64 files. It occupied only 8 Kbytes.

When the PC XT appeared (1983), which included a hard drive, Microsoft developed the second version of MS-DOS, with support for hard disk and hierarchical directories. It also incorporated some Unix features, such as I/O redirection.

In 1984, with the PC/AT, the Intel 80286 processor offered extended addressing and memory protection mechanisms. Microsoft introduced the version 3.0 of MS-DOS, which did not take advantage of the new processor. There were several notable updates in this release. Version 3.1 included network support. From here successive versions of MS-DOS are appearing without major structural changes.

There are two remarkable facts behind the success of MS-DOS: (a) the appearance, with the blessing of IBM, of cheap PC clones to which Microsoft provided software —Microsoft kept MS-DOS as proprietary operating system—, and (b) maintaining compatibility with previous versions. The latter resulted, however, in MS-DOS being a less developed system than others from their competitors.

After IBM choose its own operating system OS/2 for PCs, Microsoft released **Windows 3.0** in 1990, copying the idea of the graphical user interface previously marketed by Apple. Windows is just an interface for MS-DOS and does not provide true multitasking. Still it was a great success and its use spread rapidly.

Windows 95/98. In 1995 Microsoft had already released Windows NT, a new operating system designed from scratch for the server market, but the hardware of personal computers of the time was very limited to support it. Moreover, Windows 3.11 was ridiculously primitive compared to other less prevalent systems like Mac OS from Apple, which had long offered multitasking, memory protection and 32-bit addressing. In light of this, Microsoft decided to redesign Windows 3.11 to provide these features, while remaining compatible with 16-bit applications of Windows 3.x and MS-DOS, marketing it under the name of Windows 95. The Windows 98 and Windows ME (Millennium Edition) systems are a continuation of Windows 95.

Windows NT/2000/XP/Vista/7. In 1988, Microsoft hired Digital engineers with experience in the development of VMS, for a new operating system project called Windows NT (*New Technology*). The aim is to develop an operating system that integrates new design concepts: client/server architecture based on a microkernel and multiprocessor support. The microkernel structure was diluted through successive versions. Early versions —from NT 3.1 in 1993, to NT 5.0, traded as Windows 2000— are aimed at workstations and servers. In 2001 version 5.1 is released, marketed as Windows XP, which includes for the first time a specific version for home use, ending Windows 95/98 and, thus, the support line of 16-bit applications. Windows XP includes versions for 64-bit processors. NT 6.0 (Windows Vista), launched in 2007, represents a significant revision of the architecture, including a new graphical interface and new protection mechanisms, in addition to many services. This results in a high avidity of resources which obsoletes much of the personal computer park. The successor is launched in 2009, NT 6.1 (Windows 7), which refines the implementation to improve performance and also updates forms of user interaction.

Mac OS

In 1979 Xerox PARC gave Apple the rights to use its graphical interface, which included elements such as icons and mouse. Apple included this interface in the personal computer Lisa (1980), which pioneered the Macintosh (1984) and the Mac OS operating system. Apart from its advanced graphical interface, Mac OS offered cooperative multiprogramming (a form of time-sharing in which each task is responsible for giving the processor to another task). In its early years, the Macintosh was a huge success, but its relatively high price and closed system strategy motivated

that Microsoft, mainly thanks to its partnership with IBM, imposed its MS-DOS, despite the delay in introducing a decent graphical interface.

Mac OS evolved to version 9 (1999). In 2000, Apple sells the new Mac OS X, derived from NeXTSTEP, an operating system based on the Mach 3.0 microkernel. Mac OS X incorporates BSD Unix code and provides its system call interface. Later Apple adopted Intel as hardware platform, in substitution of Motorola.

Apple has adapted Mac OS X for mobile devices, marketed under the name iOS. Apple's leading position in this market ensures a good spread of iOS.

Bibliography

A.S. Tanenbaum: *Modern Operating Systems (3rd edition).* Prentice-Hall, 2008.

W. Stallings: *Operating Systems (5th edition).* Prentice-Hall, 2004.

Wikipedia: http://en.wikipedia.org