eBPF: <https://github.com/Microsoft/ebpf-for-windows/>

eBPF has resulted in a new generation of tooling that allows developers to easily diagnose problems, innovate quickly, and extend operating system functionality (Mark Russinovich, Chief Technology Officer at Microsoft Azure, 2021)

**eBPF** (often aliased **BPF**) is a technology that can run sandboxed programs in a privileged context such as the operating system kernel.

It is used to safely and efficiently extend the capabilities of the kernel at runtime without requiring to change kernel source code or load kernel modules.

Safety is provided through an in-kernel verifier which performs static code analysis (the analysis of computer programs performed without executing them, in contrast with dynamic program analysis, which is performed on programs during their execution) and rejects programs which crash, hang or otherwise interfere with the kernel negatively.

Loaded programs which passed the verifier are either interpreted or in-kernel JIT compiled for native execution performance.

The execution model is event-driven (**event-driven programming** is a programming paradigm in which the flow of the program is determined by events such as user actions, sensor outputs, or message passing from other programs or threads) and with few exceptions run-to-completion (a scheduling model in which each task runs until it either finishes, or explicitly yields control back to the scheduler), meaning, programs can be attached to various hook points in the operating system kernel and are run upon triggering of an event.

eBPF use cases include (but are not limited to) networking such as XDP, tracing and security subsystems.

Given eBPF's efficiency and flexibility opened up new possibilities to solve production issues, Brendan Gregg famously coined eBPF as "superpowers for Linux".

Linus Torvalds expressed that "BPF has actually been really useful, and the real power of it is how it allows people to do specialized code that isn't enabled until asked for".

Due to its success in Linux, the eBPF runtime has been ported to other operating systems such as Windows.

SANDBOX

In computer security, a **sandbox** is a security mechanism for separating running programs, usually in an effort to mitigate system failures and/or software vulnerabilities from spreading.

The isolation metaphor is taken from the idea of children who do not play well together, so each is given their own sandbox to play in alone.

It is often used to execute untested or untrusted programs or code, possibly from unverified or untrusted third parties, suppliers, users or websites, without risking harm to the host machine or operating system.

A sandbox typically provides a tightly controlled set of resources for guest programs to run in, such as storage and memory scratch space.

Network access, the ability to inspect the host system, or read from input devices are usually disallowed or heavily restricted.

In the sense of providing a highly controlled environment, sandboxes may be seen as a specific example of virtualization (the act of creating a virtual, not actual, version of something at the same abstraction level).

Sandboxing is frequently used to test unverified programs that may contain a virus or other malicious code without allowing the software to harm the host device.

PRIVILEGIO

In computing, **privilege** is defined as the delegation of authority to perform security-relevant functions on a computer system.

A privilege allows a user to perform an action with security consequences.

Examples of various privileges include the ability to create a new user, install software, or change kernel functions.

Users who have been delegated extra levels of control are called privileged.

Users who lack most privileges are defined as unprivileged, regular, or normal users.

Privileges can either be automatic, granted, or applied for.

RUNTIME SYSTEM

In computer programming, a **runtime system** or **runtime environment** is a sub-system that exists both in the computer where a program is created, as well as in the computers where the program is intended to be run.

The name comes from the compile time and runtime division from compiled languages, which similarly distinguishes the computer processes involved in the creation of a program (compilation) and its execution in the target machine (the run time).

Most programming languages have some form of runtime system that provides an environment in which programs run.

This environment may address a number of issues including the management of application memory, how the program accesses variables, mechanisms for passing parameters between procedures, interfacing with the operating system, and otherwise.

The compiler makes assumptions depending on the specific runtime system to generate correct code.

Typically the runtime system will have some responsibility for setting up and managing the stack and heap, and may include features such as garbage collection (a form of automatic memory management that attempts to reclaim memory which was allocated by the program, but is no longer referenced, called *garbage*), threads of execution (the smallest sequence of programmed instructions that can be managed independently by a scheduler), or other dynamic features built into the language.

Every programming language specifies an execution model, and many implement at least part of that model in a runtime system.

One possible definition of runtime system behaviour, among others, is "any behaviour not directly attributable to the program itself".

This definition includes putting parameters onto the stack before function calls, parallel execution of related behaviours, and disk I/O.

By this definition, essentially every language has a runtime system.

EXECUTION

**Execution** in computer and software engineering is the process by which a computer or virtual machine reads and acts on the instructions of a computer program.

Each instruction of a program is a description of a particular action which must be carried out, in order for a specific problem to be solved.

Execution involves repeatedly following a 'fetch–decode–execute' cycle (The **instruction cycle** that the CPU follows from boot-up of the computer until it has shut down in order to process instructions) for each instruction done by control unit (a component of a computer's CPU that directs the operation of the processor).

As the executing machine follows the instructions, specific effects are produced in accordance with the semantics of those instructions.

SISTEMA OPERATIVO

An **operating system** (**OS**) is system software that manages computer hardware and software resources, and provides common services for computer programs.

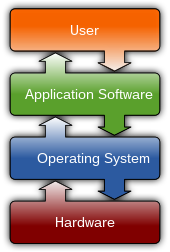
Hardware is typically directed by the software to execute any command or instruction.

A combination of hardware and software forms a usable computing system.

**Software** is a set of computer programs and associated documentation and data.

It is the set of instructions that can be stored and run by hardware.

**Computer hardware** includes the physical parts of a computer.

From the hardware the system is built and which actually performs the work. 

KERNEL

The **kernel** is a computer program (a sequence or set of instructions in a programming language for a computer to execute) at the core of a computer's operating system and generally has complete control over everything in the system.

It is the portion of the operating system code that is always resident in memory and facilitates interactions between hardware and software components.

A full kernel controls all hardware resources (e.g. I/O, memory, cryptography) via device drivers, arbitrates conflicts between processes concerning such resources, and optimizes the utilization of common resources e.g. CPU & cache usage, file systems, and network sockets.

On most systems, the kernel is one of the first programs loaded on startup (after the bootloader).

It handles the rest of startup as well as memory, peripherals, and input/output (I/O) requests from software, translating them into data-processing instructions for the central processing unit.

The critical code of the kernel is usually loaded into a separate area of memory, which is protected from access by application software or other less critical parts of the operating system.

The kernel performs its tasks, such as running processes, managing hardware devices such as the hard disk, and handling interrupts, in this protected kernel space.

In contrast, application programs such as browsers, word processors, or audio or video players use a separate area of memory, user space.

This separation prevents user data and kernel data from interfering with each other and causing instability and slowness, as well as preventing malfunctioning applications from affecting other applications or crashing the entire operating system.

Even in systems where the kernel is included in application address spaces, memory protection is used to prevent unauthorized applications from modifying the kernel.

The kernel's interface is a low-level abstraction layer.

When a process requests a service from the kernel, it must invoke a system call, usually through a wrapper function.

There are different kernel architecture designs.

* Monolithic kernels run entirely in a single address space with the CPU executing in supervisor mode, mainly for speed.
* Microkernels run most but not all of their services in user space, like user processes do, mainly for resilience and modularity

The Linux kernel is monolithic, although it is also modular, for it can insert and remove loadable kernel modules at runtime.

This central component of a computer system is responsible for executing programs. The kernel takes responsibility for deciding at any time which of the many running programs should be allocated to the processor or processors.

SYSTEM CALL

A system call (syscall) is the programmatic way in which a computer program requests a service from the operating system on which it is executed.

This may include hardware-related services (for example, accessing a hard disk drive or accessing the device's camera), creation and execution of new processes, and communication with integral kernel services (routines that provide the runtime environment to programs executing in kernel mode) such as process scheduling.

Kernel extensions call kernel services. In contrast, application programs call library routines.

System calls provide an essential interface between a process and the operating system.

WRAPPER FUNCTION

A **wrapper function** is a function (another word for a *subroutine*) in a software library or a computer program whose main purpose is to call a second subroutine or a system call with little or no additional computation.

Wrapper functions are used to make writing computer programs easier by abstracting away the details of a subroutine's underlying implementation.

LOADABLE KERNEL MODULE

In computing, a **loadable kernel module** (**LKM**) is an object file that contains code to [extend](https://en.wikipedia.org/wiki/Extensibility) the running [kernel](https://en.wikipedia.org/wiki/Kernel_(operating_system)), or so-called *base kernel*, of an operating system.

LKMs are typically used to add support for new [hardware](https://en.wikipedia.org/wiki/Computer_hardware) (as [device drivers](https://en.wikipedia.org/wiki/Device_driver)) and/or [filesystems](https://en.wikipedia.org/wiki/Filesystem), or for adding [system calls](https://en.wikipedia.org/wiki/System_call).

When the functionality provided by an LKM is no longer required, it can be unloaded in order to free [memory](https://en.wikipedia.org/wiki/Computer_storage) and other resources.

Most current [Unix-like](https://en.wikipedia.org/wiki/Unix-like) systems and [Microsoft Windows](https://en.wikipedia.org/wiki/Microsoft_Windows) support loadable kernel modules under different names.

Advantages

Without loadable kernel modules, an operating system would have to include all possible anticipated functionality compiled directly into the base kernel.

Much of that functionality would reside in memory without being used, wasting memory, and would require that users rebuild and reboot the base kernel every time they require new functionality.

Disadvantages

One minor criticism of preferring a modular kernel over a static kernel is the so- called *fragmentation penalty*.

The base kernel is always unpacked into real contiguous memory by its setup routines; thus, the base kernel code is never fragmented.

Once the system is in a state in which modules may be inserted, for example once the filesystems have been mounted that contain the modules, it is likely that any new kernel code insertion will cause the kernel to become fragmented, thereby introducing a minor performance penalty.

INTERPRETED OR IN-KERNEL JIT COMPILED FOR NATIVE EXECUTION PERFORMANCE

* In computer science, an **interpreter** is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.
* In computing, a **compiler** is a computer program that translates computer code written in one programming language (the *source* language) into another language (the *target* language)
* In computing, **just-in-time** (**JIT**) **compilation** (also **dynamic translation** or **run-time compilations**) is a way of executing computer code that involves compilation during execution of a program (at run time) rather than before execution.

This may consist of source code translation but is more commonly bytecode translation to machine code, which is then executed directly.