APUNTES DE SISTEMAS INFORMÁTICOS

UD3 - SOFTWARE Y SISTEMA OPERATIVO

Módulo: 0483. Sistemas informáticos
Ciclo Formativo: IF303 - Técnico Superior en Desarrollo de Aplicaciones Web
Profesora: María Paradela García

1. Concepto de software

1.1. Introducción: el alma invisible del ordenador

Cuando hablamos de un sistema informático, solemos pensar primero en lo tangible: la torre del ordenador, el monitor, el teclado, el ratón o incluso los servidores que imaginamos en una gran empresa. Pero esos componentes no hacen nada por sí mismos.

Sin instrucciones, un ordenador es tan útil como una calculadora sin botones: tiene potencial, pero carece de dirección.

El software es precisamente eso: la parte lógica que da sentido a la parte física. Es el conjunto de programas, rutinas y datos que dicen al hardware qué debe hacer, cuándo y cómo.

Un ordenador sin software es como un cuerpo sin sistema nervioso.

El hardware proporciona los órganos; el software, los impulsos eléctricos y las órdenes que los hacen funcionar en coordinación.

1.2. Definición formal y sentido funcional del software

En términos técnicos, se define como:

> "El conjunto de programas, procedimientos y rutinas asociadas a la operación de un sistema informático."
> *(Norma ISO/IEC 2382-1:2015 – Terminología de Tecnología de la Información).*

Esto incluye desde los programas más visibles —un procesador de textos o un navegador— hasta los componentes invisibles que trabajan en segundo plano: controladores, librerías, sistemas operativos, firmware o microcódigo de la CPU.

El software es, por tanto, una entidad intangible pero esencial: se almacena físicamente en soportes (memoria, disco, nube), pero su verdadera naturaleza es lógica y abstracta. Lo que almacenamos son instrucciones codificadas, no el software "vivo". El programa solo existe realmente cuando se ejecuta, cuando el procesador interpreta esas instrucciones una a una.

1.3. Breve historia del software

El software es una invención relativamente reciente, y su historia va ligada a la de la programación.

1.3.1. De los cables al código

En los años 40, las primeras computadoras electrónicas (ENIAC, Colossus, Harvard Mark I) no distinguían entre hardware y software.

Los programas se implementaban conmutando cables y conectores. Para cambiar de programa, literalmente había que recablear el sistema.

Era una tarea manual, lenta y propensa a errores.

En 1945, John von Neumann propuso un nuevo modelo: almacenar las instrucciones del programa en la misma memoria que los datos.

Esa idea revolucionó la informática: permitió que una máquina cambiara de tarea simplemente cargando otro programa.

Así nació la arquitectura de von Neumann, base de casi todos los ordenadores modernos.

1.3.2. Los primeros lenguajes y el nacimiento del compilador

En los años 50 surgieron los primeros lenguajes de programación de alto nivel, como FORTRAN y COBOL. Hasta entonces, se programaba en código máquina —secuencias de ceros y unos— o en ensamblador, una versión simbólica del mismo. La informática era un oficio reservado a ingenieros electrónicos.

Una figura clave fue Grace Hopper, creadora del primer compilador, un programa capaz de traducir un lenguaje comprensible para humanos a lenguaje máquina. Su trabajo dio origen a COBOL y sentó las bases de la programación moderna.

1.3.3. Del laboratorio a la oficina: el software como producto

En los años 60 y 70, los ordenadores empezaron a popularizarse en empresas y universidades. Los fabricantes incluían el software junto con el hardware, sin coste adicional. No existía la noción de “vender programas”: el software era una herramienta accesoria.

Eso cambió en 1969, cuando IBM decidió separar el precio del hardware y del software. Ese gesto dio lugar a una nueva industria. El software pasó a ser un producto independiente, con su propio mercado, licencias, mantenimiento y soporte técnico.

1.3.4. La era del usuario

En los 80, con la llegada de los ordenadores personales, apareció el software de consumo. Los sistemas operativos con interfaz gráfica —como el Apple Lisa, Macintosh o Windows— acercaron la informática a millones de personas. Ya no era necesario conocer comandos: bastaba con hacer clic.

En los 90, Internet lo cambió todo: los programas empezaron a comunicarse entre sí a través de redes. El software se convirtió en una plataforma de servicios, no solo en una herramienta local.

1.3.5. Del escritorio a la nube y la inteligencia artificial

Hoy vivimos en la era del software ubicuo. Ya no lo instalamos, lo usamos: correo web, almacenamiento en la nube, herramientas colaborativas, inteligencia artificial, sistemas operativos móviles. La frontera entre aplicación, servicio y sistema operativo es cada vez más difusa. ChromeOS o Android son buenos ejemplos: gran parte de su funcionalidad depende de Internet.

En resumen, la evolución del software refleja la evolución del propio ser humano: de controlar máquinas manuales a diseñar sistemas que aprenden solos.

1.4. Diferencias entre hardware, software y firmware

Aunque a veces se traten como términos equivalentes, cada uno cumple una función distinta dentro del sistema informático.

Elemento	Qué es	Función	Ejemplo
Hardware	Parte física y tangible del sistema.	Ejecuta las operaciones ordenadas por el software.	CPU, RAM, disco, monitor, teclado.
Software	Parte lógica, intangible.	Da instrucciones y coordina el hardware.	Sistema operativo, navegador, juego.
Firmware	Software embebido en hardware, grabado en memoria no volátil.	Inicializa y controla dispositivos antes del SO.	BIOS/UEFI, firmware de router o impresora.

El firmware se encuentra en un punto intermedio entre hardware y software. Se graba en chips de memoria tipo ROM o Flash y permite que el dispositivo funcione incluso sin sistema operativo. Por ejemplo, el firmware de una impresora controla el movimiento del cabezal, las temperaturas y el consumo de tinta sin depender del ordenador al que está conectada.

> Curiosidad: cuando actualizas la BIOS o el firmware de tu teléfono, en realidad estás reescribiendo un pequeño programa dentro de un chip soldado. Si se interrumpe la actualización, el dispositivo puede quedar inutilizable: el famoso “brick” o “ladrillo”.

1.5. El software como sistema de capas

Para comprender la relación entre hardware y software, conviene imaginar el sistema informático como un conjunto de capas jerárquicas, cada una apoyada sobre la anterior.

- **Capa física (hardware):** procesador, memoria, almacenamiento, dispositivos.
- **Capa lógica básica (firmware):** arranque e inicialización del hardware.
- **Capa de control (sistema operativo):** gestión de recursos y comunicación entre hardware y programas.

- **Capa de aplicación:** programas que ejecuta el usuario.
- **Capa de usuario:** interacción humana.

Este modelo en capas no es arbitrario: responde a una necesidad de orden y aislamiento. Cada capa oculta la complejidad de la inferior y ofrece servicios simplificados a la superior. Por ejemplo, una aplicación no necesita saber cómo funciona un disco SSD; le basta con pedir al sistema operativo que guarde un archivo.

En los sistemas actuales hay capas aún más complejas, como las máquinas virtuales, que permiten ejecutar sistemas dentro de otros, o las API (interfaces de programación), que conectan programas distintos sin necesidad de entender su código interno.

1.6. Clasificación general del software

Tradicionalmente, el software se divide en tres grandes categorías:

- Software de sistema.
- Software de aplicación.
- Software de desarrollo.

1.6.1. Software de sistema

Es el que hace posible el funcionamiento del ordenador.

Incluye el sistema operativo, los controladores de dispositivos y los utilitarios que gestionan, supervisan o mantienen el hardware.

Ejemplos:

- Windows, Linux, macOS.
- Drivers de impresora o tarjeta gráfica.
- Herramientas de diagnóstico o antivirus.

Sin software de sistema, no podríamos ejecutar ningún otro programa.

1.6.2. Software de aplicación

Está orientado al usuario final y a sus tareas concretas.

Puede ser de propósito general (ofimática, diseño, navegación) o especializado (contabilidad, arquitectura, edición de vídeo, etc.).

Ejemplos:

- Google Chrome, LibreOffice, Photoshop, AutoCAD, Spotify, Visual Studio Code.

En la actualidad, muchas aplicaciones ya no se instalan, sino que se ejecutan desde la web o en la nube, lo que nos lleva a modelos como SaaS (Software as a Service).

1.6.3. Software de desarrollo

Son las herramientas que permiten crear nuevo software.

Incluyen compiladores, intérpretes, depuradores, entornos de desarrollo integrados (IDE), librerías y frameworks.

Ejemplos:

- Visual Studio, IntelliJ IDEA, Eclipse, Node.js, Django, React, Git.

El software de desarrollo no solo produce programas, también facilita la colaboración y la automatización: control de versiones, integración continua, despliegue en la nube, etc.

Ejemplo DAW: cuando un desarrollador crea una web en Visual Studio Code, está usando software de desarrollo.

- El navegador que mostrará esa web será software de aplicación.
- El sistema operativo sobre el que corre todo es software de sistema.

1.7. Otras clasificaciones complementarias

Aunque la división clásica es la más usada, existen otros criterios complementarios:

1.7.1. Por distribución

- Propietario: requiere licencia o pago, no se puede modificar (Windows, Adobe Photoshop).
- Libre: código fuente abierto, se puede modificar (Linux, Firefox).
- Freeware: gratuito pero sin acceso al código (Skype, WinRAR).
- Shareware: gratuito por tiempo limitado o con funciones restringidas.
- Dominio público: sin derechos de autor.

1.7.2. Por plataforma

- De escritorio: instalado en ordenadores personales.
- Móvil: diseñado para smartphones y tablets.
- Web o en la nube: ejecutado desde un navegador.
- Embebido: grabado en dispositivos específicos.

1.7.3. Por finalidad

- Educativo, empresarial, científico, industrial, de entretenimiento, etc.

1.7.4. Por modelo de ejecución

- Local: instalado en el equipo.
- Cliente-servidor: distribuido entre usuarios y servidores.
- Distribuido: cooperativo entre varias máquinas.
- Virtualizado o cloud-native: ejecutado en entornos virtuales.

1.8. La importancia del software en el ecosistema digital

El software es la columna vertebral del mundo digital. Todo servicio, aplicación o dispositivo moderno depende de él. La economía, la comunicación, el transporte, la sanidad o la educación se sostienen sobre infraestructuras de software. De hecho, muchas empresas tecnológicas ya no fabrican hardware, sino ecosistemas de software: Google, Meta, Netflix o Amazon son ejemplos claros.

El software además es evolutivo. No se gasta, pero envejece. A medida que cambian las necesidades o el hardware, debe actualizarse. Esto introduce conceptos nuevos: versiones, mantenimiento, compatibilidad, soporte y ciclo de vida. El software tiene una "vida útil" y su gestión es tan importante como su desarrollo.

1.9. Reflexión final: el software como pensamiento hecho máquina

El software no es un complemento del hardware: es su razón de ser. Donde hay un ordenador, hay un sistema que interpreta órdenes humanas convertidas en lógica binaria. Esa traducción —esa alquimia invisible entre idea y ejecución— es el corazón de la informática.

Por eso, como técnico superior en desarrollo de aplicaciones web, comprender el software en toda su profundidad no es opcional: es la base sobre la que construirás todo lo demás. Saber programar sin entender qué es realmente el software sería como saber escribir sin entender un idioma.

El software nos permite extender nuestras capacidades, automatizar tareas, conectar el mundo y materializar pensamiento. En la siguiente sección estudiaremos la pieza más importante del software de sistema: el sistema operativo, ese mediador incansable que hace posible que todas las demás piezas funcionen en armonía.

2. Software de sistema y el sistema operativo

2.1. Introducción: el sistema operativo como cerebro del ordenador

Si en el punto 1 hablábamos del software como la mente del sistema informático, ahora vamos un paso más allá: dentro de esa mente hay una estructura que organiza, controla y da coherencia al resto. Esa estructura es el sistema operativo (SO), el auténtico cerebro del ordenador.

Un ordenador sin sistema operativo puede encenderse, pero no "razona". El sistema operativo coordina todos los recursos: decide qué programa se ejecuta, cuándo, cómo y con qué prioridad. Es, en cierto modo, el director de orquesta del hardware y el traductor que permite que los demás programas trabajen sin preocuparse por los detalles eléctricos.

Podemos imaginárnoslo como una torre de control de aeropuerto: cientos de aviones (procesos) quieren despegar o aterrizar al mismo tiempo (usar la CPU, la memoria o el disco). Sin un controlador que establezca turnos, prioridades y rutas, el resultado sería el caos. El sistema operativo cumple esa función: reparte recursos, evita colisiones y mantiene el orden invisible que permite que todo funcione sin interrupciones.

Históricamente, el sistema operativo surgió cuando los ordenadores dejaron de ser máquinas de un solo uso. En los años 50, cada tarea se programaba directamente sobre el hardware. Con la aparición de los primeros sistemas batch, el ordenador empezó a ejecutar trabajos en cola. A partir de los 60 y 70, aparecieron los sistemas multitarea y multiusuario, y con ellos nació la necesidad de un “gestor central” que lo controlase todo. Así nació el concepto moderno de sistema operativo.

2.2. Concepto y funciones del software de sistema

Dentro de la gran familia del software, el software de sistema es el encargado de gestionar, mantener y coordinar el hardware y los recursos básicos del ordenador. A diferencia del software de aplicación (Word, Chrome, Photoshop), el software de sistema no está orientado al usuario final, sino a servir de base para los demás programas.

El componente más importante del software de sistema es el sistema operativo, pero también forman parte de él otros elementos:

- **Controladores (drivers)**: pequeños programas que permiten que el sistema operativo se comunique con el hardware.
- **Utilidades del sistema**: herramientas para configurar, analizar y mantener el sistema (como el administrador de tareas, los desfragmentadores o los antivirus).
- **Servidores y servicios de red**: programas que gestionan conexiones, impresoras, archivos compartidos o actualizaciones automáticas.

Podemos decir que el software de sistema no “hace cosas” directamente para el usuario, sino que crea el entorno para que las aplicaciones sí puedan hacerlas. Sin él, las aplicaciones no tendrían dónde ejecutarse.

Entre sus funciones más relevantes destacan:

1. **Controlar el hardware** y garantizar que cada componente (CPU, RAM, disco, periféricos) se utilice de manera eficiente.
2. **Proporcionar una interfaz** entre el hardware y los programas, de modo que estos puedan usar los recursos sin conocer los detalles físicos.
3. **Administrar los procesos** que se ejecutan, evitando interferencias entre ellos.
4. **Organizar el almacenamiento** y el acceso a los datos mediante sistemas de archivos.
5. **Velar por la seguridad**, controlando accesos, permisos y uso de recursos.

Sin este conjunto de funciones, el hardware sería una colección de piezas inconexa

2.3. Estructura general de un sistema operativo

Un sistema operativo puede representarse como un conjunto de capas, donde cada nivel ofrece servicios al siguiente. Esa estructura modular hace posible que los sistemas sean escalables, seguros y mantenibles.

De forma general, podemos distinguir cuatro niveles principales:

1. **Nivel núcleo o kernel** – controla directamente el hardware, gestiona procesos, memoria y dispositivos.
2. **Nivel de servicios del sistema** – proporciona utilidades y funciones básicas (red, impresión, reloj, energía, etc.).
3. **Nivel de interfaz** – incluye las bibliotecas y APIs que permiten a los programas comunicarse con el sistema.
4. **Nivel de usuario** – engloba las aplicaciones y entornos gráficos o de comandos que utilizamos.

En un arranque típico, el flujo de control sería:

Firmware (BIOS/UEFI) → Kernel → Servicios del sistema → Interfaz de usuario.

Cada una de estas capas cumple una función específica, y si alguna falla, el sistema entero se vuelve inestable. Por eso los sistemas operativos modernos están diseñados con mecanismos de aislamiento y recuperación ante errores: si un proceso se bloquea, no debe arrastrar al resto.

2.4. Componentes internos:

2.4.1. Kernel o núcleo

El **kernel** es el corazón del sistema operativo. Es el primer componente que se carga al arrancar y el último que se apaga. Su función principal es **administrar los recursos del sistema** y actuar como intermediario entre el hardware y el software.

El kernel se encarga de:

- Planificar qué procesos se ejecutan y cuándo.
- Gestionar la memoria RAM y la virtual.
- Administrar los dispositivos de entrada y salida.
- Mantener la comunicación entre procesos (IPC).
- Proteger el sistema frente a accesos indebidos o conflictos.

Podemos compararlo con un director de tráfico en una autopista con miles de vehículos circulando a la vez: regula el flujo, evita colisiones y mantiene la fluidez.

Existen tres modelos principales de kernel:

- **Monolítico:** todo el sistema está integrado en un único bloque de código (Linux).
- **Microkernel:** solo incluye las funciones mínimas y delega el resto en procesos externos (macOS, Minix).
- **Híbrido:** combina ambos enfoques (Windows, Android).

> **Curiosidad:** el núcleo de Linux, desarrollado originalmente por Linus Torvalds en 1991, tiene más de 30 millones de líneas de código. Es uno de los proyectos colaborativos más grandes de la historia de la ingeniería.

2.4.2. Controladores o drivers

Los **drivers** son programas intermedios que permiten al sistema operativo comunicarse con los dispositivos físicos. Cada hardware (impresora, tarjeta de sonido, cámara, GPU) necesita su propio controlador. Sin ellos, el sistema no sabría cómo usar el dispositivo.

Por ejemplo, cuando conectamos una impresora nueva, el sistema busca un controlador compatible. Ese pequeño archivo traduce las órdenes del sistema operativo (“imprime esta página”) en señales eléctricas que la impresora entiende.

Antiguamente, los usuarios debían instalar manualmente cada driver desde disquetes o CDs. Hoy, la mayoría se descargan automáticamente o vienen integrados en el sistema operativo.

Los drivers son una de las causas más frecuentes de fallos: una versión incompatible puede provocar desde la pérdida de sonido hasta una pantalla azul. Por eso, los fabricantes publican actualizaciones constantes de controladores.

2.4.3. Servicios y daemons

Un servicio (en Windows) o daemon (en sistemas Unix y Linux) es un programa que se ejecuta en segundo plano sin intervención del usuario. Gestiona tareas esenciales del sistema: red, impresión, sincronización de hora, seguridad, actualizaciones o indexación de archivos.

El término “daemon” viene del griego *daimon*, un espíritu que realiza tareas invisibles entre los dioses y los hombres. Una metáfora perfecta: los demonios informáticos trabajan en silencio, invisibles, pero su ausencia paraliza el sistema.

Ejemplos:

1. En Windows: `Spooler` (cola de impresión), `Windows Update`, `Bluetooth Service`.
2. En Linux: `sshd` (conexiones seguras), `cron` (tareas programadas), `NetworkManager`.

2.4.4. Sistema de archivos

El **sistema de archivos** organiza cómo se almacenan y se recuperan los datos en los dispositivos.
Define la estructura jerárquica de carpetas, nombres, extensiones y permisos.

Sin un sistema de archivos, el disco sería una masa amorfa de bits. El sistema operativo utiliza tablas de asignación y directorios para saber qué bloques de datos pertenecen a cada archivo y dónde se encuentran físicamente.

Ejemplos de sistemas de archivos:

- **NTFS** (Windows)
- **EXT4** (Linux)
- **APFS** (macOS)
- **FAT32** (usado en memorias USB y cámaras)

Los sistemas modernos soportan permisos, compresión, cifrado y journaling, una técnica que guarda un registro de operaciones pendientes para recuperar datos tras un fallo.

NTFS (New Technology File System), utilizado por Windows, es uno de los más avanzados en sistemas de escritorio. Soporta archivos muy grandes, permisos detallados de usuario, cifrado, compresión y el registro (*journaling*) de todas las operaciones. Gracias a ese registro, si el equipo se apaga bruscamente, el sistema puede reconstruir el estado anterior sin perder información.

EXT4 (Fourth Extended File System) es el estándar en la mayoría de distribuciones Linux. Es estable, eficiente y extremadamente robusto. Su diseño permite gestionar millones de archivos con baja fragmentación y excelente rendimiento. Además, maneja muy bien las particiones grandes y el acceso simultáneo de varios procesos, algo vital en servidores.

APFS (Apple File System), introducido por Apple en 2017, está optimizado para unidades de estado sólido (SSD). Se centra en la velocidad, la integridad de los datos y el cifrado completo del disco. Una de sus ventajas es la capacidad de crear *snapshots*, copias instantáneas del sistema que permiten volver atrás sin reinstalar.

Por último, **FAT32 (File Allocation Table)** es el más veterano. Data de los años 90 y se sigue usando en memorias USB, tarjetas SD o cámaras porque es extremadamente compatible: casi todos los sistemas operativos lo reconocen. Sin embargo, tiene limitaciones importantes: no admite archivos mayores de 4 GB y carece de medidas de seguridad modernas.

En la práctica, cada uno cumple una función: NTFS reina en entornos Windows; EXT4 domina los sistemas Linux; APFS gobierna el ecosistema Apple; y FAT32 sigue siendo el punto de encuentro entre todos.

Entender esta diversidad ayuda a comprender por qué a veces un pendrive formateado en macOS no funciona en Windows, o por qué un servidor Linux gestiona millones de archivos sin inmutarse mientras un equipo doméstico se ralentiza con una carpeta llena de vídeos

2.4.5. Shell e interfaces de usuario

El shell es la interfaz que permite al usuario comunicarse con el sistema operativo. Puede ser textual (línea de comandos) o gráfica (entorno de ventanas). Ambas conviven en la mayoría de los sistemas actuales.

- **CLI (Command Line Interface):** usada por administradores y técnicos, ofrece control total mediante comandos (`bash`, `PowerShell`).
- **GUI (Graphical User Interface):** usada por la mayoría de los usuarios, prioriza la accesibilidad y la intuición.

En Linux, por ejemplo, el entorno gráfico GNOME o KDE se apoya sobre un shell como *bash* o *zsh*. En Windows, detrás de los iconos y menús hay un shell llamado *Explorer.exe*.

La línea de comandos, lejos de estar obsoleta, es insustituible en entornos profesionales por su velocidad, precisión y capacidad de automatización.

2.5. Funciones esenciales del sistema operativo

El sistema operativo es un gestor universal de recursos.

Su misión principal es permitir que un conjunto de programas, usuarios y dispositivos compartan un mismo hardware sin interferencias ni pérdidas de rendimiento.

Para conseguirlo, organiza su trabajo en una serie de funciones básicas, que actúan como los órganos vitales del sistema: gestión de procesos, gestión de memoria, gestión de almacenamiento, gestión de entrada/salida y gestión de usuarios y seguridad.

2.5.1. Gestión de procesos

Un proceso es un programa en ejecución.

Cuando hacemos doble clic sobre un icono, no se ejecuta directamente el archivo; lo que hace el sistema operativo es crear un proceso en memoria, asignarle recursos y dejarlo correr. Cada proceso tiene su propio espacio de memoria, su identificador (PID) y un estado: listo, en ejecución o en espera.

El sistema operativo coordina todos esos procesos mediante un planificador de tareas (scheduler). El planificador decide qué proceso puede usar la CPU en cada instante, por cuánto tiempo y en qué orden. En los sistemas modernos, esta decisión se toma miles de veces por segundo.

En un equipo con decenas de programas abiertos —el navegador, el editor de texto, el cliente de correo, el antivirus, los servicios del sistema— todos “creen” que están ejecutándose simultáneamente. Pero en realidad, el procesador solo ejecuta uno cada vez; el sistema operativo intercambia tan rápido entre ellos que el usuario percibe multitarea real.

Los sistemas operativos aplican distintos algoritmos de planificación, según el objetivo:

- **Round Robin:** reparte el tiempo de CPU por turnos iguales (útil en entornos compartidos).
- **Por prioridad:** asigna más tiempo a los procesos más importantes.
- **Multinivel:** combina ambos métodos para equilibrar rendimiento y respuesta.

Una buena gestión de procesos garantiza estabilidad.Si un programa se cuelga, el sistema puede cerrarlo sin detener el resto, algo que no era posible en los ordenadores de los años 80, cuando un error en una aplicación obligaba a reiniciar la máquina completa.

> Ejemplo: cuando abres veinte pestañas en Chrome, cada una es un proceso distinto. Si una falla, las demás continúan funcionando porque el sistema las aísla unas de otra

2.5.2. Gestión de memoria

La memoria es el recurso más delicado del sistema: es limitada, volátil y constantemente demandada por todos los programas. El sistema operativo decide cómo se reparte, qué procesos la ocupan y cuándo se libera.

Cuando un programa se ejecuta, el sistema le asigna un espacio de direcciones: una zona exclusiva de memoria que no puede invadir ningún otro proceso. Esto evita que un error en un programa corrompa los datos de otro.

Para maximizar el rendimiento, los sistemas modernos utilizan memoria virtual. Consiste en simular una memoria más grande de la que existe físicamente, almacenando temporalmente parte de la información en un disco duro (archivo de paginación o *swap*). Cuando un programa necesita datos que están disco, el sistema los trae a la RAM y envía otros al almacenamiento, intercambiando continuamente páginas de memoria.

Este mecanismo permite tener abiertas decenas de aplicaciones a la vez, aunque la memoria física no sea suficiente. El precio es el tiempo: acceder al disco es miles de veces más lento que acceder a la RAM. Por eso, cuando equipo se queda corto de memoria, notamos que “va lento”: el sistema está paginando constantemente.

Para optimizar el uso de la memoria, el sistema operativo utiliza técnicas como:

- **Segmentación:** divide la memoria en áreas lógicas (código, datos, pila).
- **Paginación:** divide la memoria en bloqeus del mismo tamaño, lo que simplifica la gestión.
- **Protección:** impide que un proceso acceda al espacio de otro.

> **Curiosidad:** en los primeros PC con MS-DOS, no existía memoria virtual. Sin programa pedía más RAM de la disponible, simplemente fallaba. El sistema de memoria virtual evita este colapso fingiendo disponer de más memoria.

2.5.3. Gestión de almacenamiento

El almacenamiento es el medio donde se guardan los datos de forma permanente. El sistema operativo no se limita a escribir y leer archivos: organiza la información, gestiona el espacio libre y mantiene la coherencia entre los datos y su representación física.

Para ello utiliza un sistema de archivos, que define cómo se nombran, guardan y estructuran los archivos en el disco duro. Ya lo vimos antes, NTFS, EXT4, APFS, FAT32, etc. El sistema operativo traduce los comandos del usuario en operaciones físicas sobre el disco.

Ejemplo: cuando guardas un documento, no se escribe entero de una vez. El sistema divide el archivo en bloques, los distribuye en diferentes sectores del disco, actualiza la tabla de asignación y registra la operación en el diario (journal). Todo esto ocurre en milisegundos.

Además, el sistema operativo se encarga de tareas como:

- Mantener la coherencia de los datos (evitar pérdidas tras un corte de energía).

- Controlar el espacio disponible y eliminar archivos temporales.
- Gestionar cachés para acelerar accesos repetitivos.
- Coordinar varios dispositivos de almacenamiento (discos, SSD, unidades externas, red…)

> **Ejemplo**: en un servidor de bases de datos, la eficiencia del sistema de archivos puede marcar la diferencia entre un sistema que responde en milisegundos y otro que se colapsa con miles de peticiones simultáneas.

2.5.4. Gestión de E/S

La entrada/salida (E/S) es todo lo que implica comunicación entre el ordenador y el exterior: teclado, ratón, monitor, impresora, red, micrófono, USB, etc. Cada dispositivo tiene su propio ritmo y protocolo de comunicación. El sistema operativo actúa como intermediario para que todos puedan funcionar coordinadamente.

Cuando el usuario pulsa una tecla, por ejemplo, el teclado envía una señal eléctrica. El controlador traduce esa señal en un código que el sistema operativo entiende, y este lo envía al programa activo, que decide qué hacer con él (mostrar una letra, ejecutar una orden, etc.).

La E/S es, por tanto, un proceso de **traducción constante** entre señales físicas y datos digitales. El sistema operativo usa técnicas como:

- **Buffering:** almacena temporalmente los datos en memoria para compensar diferencias de velocidad entre dispositivos.
- **Spooling:** gestiona colas de impresión o tareas pendientes.
- **Interrupciones:** permiten que el procesador atienda dispositivos solo cuando lo necesitan, evitando pérdida de tiempo.

> **Curiosidad:** el ratón original de los primeros Macintosh usaba una conexión serial de apenas 1200 baudios. Hoy, un ratón moderno envía cientos de eventos por segundo, y aun así el sistema operativo los procesa sin que lo notemos.

2.5.5. Gestión de usuarios y seguridad

Por último, el sistema operativo es responsable de garantizar que solo los usuarios autorizados puedan acceder a los recursos y que cada uno lo haga con los permisos adecuados. Esto incluye no solo el acceso físico al equipo, sino también los permisos sobre archivos, dispositivos y servicios.

Cada usuario tiene una cuenta, un identificador (UID) y un conjunto de permisos que determinan lo que puede hacer: leer, escribir, ejecutar o administrar. En sistemas multiusuario como Linux o Windows Server, esta jerarquía es esencial para mantener la integridad del sistema.

El sistema operativo también implementa mecanismos de autenticación (contraseñas, certificados, huellas digitales) y de autorización, que verifican qué acciones están permitidas. Además, registra todas las actividades relevantes en los logs del sistema, un historial que permite auditar lo ocurrido en caso de error o ataque.

Otra función crítica es la protección contra software malicioso. Aunque los antivirus sean aplicaciones independientes, el sistema operativo proporciona el entorno y los permisos necesarios para que puedan detectar comportamientos sospechosos y aislar procesos peligrosos.

> **Ejemplo:** cuando un programa intenta modificar archivos del sistema sin autorización, Windows o Linux lo bloquean mediante un control de permisos.

Este principio de “mínimo privilegio” es fundamental: cada proceso debe tener solo los permisos imprescindibles para su tarea, nunca más.

Con estas cinco funciones —procesos, memoria, almacenamiento, E/S y seguridad—, el sistema operativo mantiene el equilibrio entre potencia y estabilidad.

Gracias a él, el usuario puede interactuar con una máquina que, sin esta capa de control, sería inusable.

2.6. Tipos de sistemas operativos

Uno de los grandes aciertos de la ingeniería informática ha sido la diversidad.

No existe un único tipo de sistema operativo universal; cada entorno —desde un reloj digital hasta un superordenador— necesita uno adaptado a su función, a su capacidad y a sus usuarios.

Por eso hablamos de **t**ipos de sistemas operativos, clasificados según distintos criterios.

Entender estas categorías no es solo cuestión de teoría: permite comprender por qué Windows, Linux o Android se comportan de forma distinta, y por qué existen sistemas más simples o más especializados según la situación.

2.6.1. Por usuarios

Monousuario

Los sistemas monousuario están diseñados para que una sola persona utilice el equipo a la vez. Esto no significa que solo puedan ejecutar un programa, sino que todo el entorno (permisos, configuraciones, archivos) pertenece a un único usuario.

Ejemplos clásicos de sistemas monousuario fueron MS-DOS o Windows 95/98, donde no existía una gestión real de cuentas. El sistema arrancaba, cargaba el entorno y cualquiera que encendiera el equipo tenía acceso completo a todo.

A nivel técnico, un sistema monousuario no necesita controlar privilegios complejos ni separar procesos por identidad. Esto simplifica el diseño, pero también implica riesgo de seguridad: un solo error o virus afecta a todo el sistema.

Hoy casi ningún sistema moderno es puramente monousuario, aunque algunos dispositivos domésticos o sistemas embebidos mantienen esa lógica. Por ejemplo, la consola de una Smart TV o el firmware de un router no gestionan múltiples cuentas simultáneas: operan con un único perfil.

Multiusuario

Un sistema multiusuario permite que varias personas utilicen el mismo ordenador o red al mismo tiempo (conectadas local o remotamente) o de forma alterna, manteniendo separadas sus configuraciones, permisos y archivos.

Ejemplos de este tipo son Unix, GNU/Linux, macOS o Windows Server. Cada usuario tiene su cuenta propia, protegida por contraseña, y el sistema mantiene un control riguroso de quién accede, qué procesos ejecuta y qué recursos usa.

Los sistemas multiusuario implementan mecanismos como:

- Control de identidad (UID, contraseñas, certificados).
- Asignación de permisos y grupos de usuarios.
- Aislamiento de procesos y archivos para evitar interferencias.
- Registro de actividades en logs, con fines de auditoría.

En un entorno de red o de empresa, el multiusuario es esencial. Por ejemplo, en un servidor de aplicaciones web, cientos o miles de usuarios pueden conectarse simultáneamente; el sistema operativo debe garantizar que cada uno vea solo su información y no interfiera con los demás.

> **Ejemplo:** en un servidor Linux, si un usuario intenta modificar un archivo del sistema sin permisos, recibirá un mensaje de “Acceso denegado”. Ese simple mensaje es el resultado de años de desarrollo en control multiusuario y seguridad.

2.6.2. Por tareas

Monotarea

Un sistema monotarea solo puede ejecutar una tarea a la vez. Mientras esa tarea se ejecuta, el resto del sistema permanece a la espera. Los primeros ordenadores domésticos funcionaban así: el procesador se dedicaba por completo a un único proceso hasta que terminaba.

> Ejemplo: el MS-DOS original. Si el usuario imprimía un documento, no podía hacer nada más hasta que la impresora acabara. Aunque parezca prehistórico, ese modelo era sencillo y eficiente para máquinas con poca memoria o sin necesidad de multitarea.

Hoy en día, los sistemas monotarea aún existen en entornos embebidos: controladores industriales, cajeros automáticos o dispositivos médicos, donde la fiabilidad y la simplicidad son más importantes que la flexibilidad. Un marcapasos, por ejemplo, solo tiene que hacer una cosa —controlar impulsos cardíacos— y hacerla sin errores. No necesita multitarea.

Multitarea

Los sistemas multitarea permiten ejecutar varios procesos de manera aparentemente simultánea. La CPU no hace varias cosas a la vez (a menos que tenga varios núcleos), sino que cambia de tarea a una velocidad tan alta que para el usuario parece que todo sucede a la vez.

Hay dos formas de multitarea:

- **Cooperativa:** cada programa cede voluntariamente el control de la CPU. Si uno no lo hace, bloquea el sistema (ejemplo: Mac OS clásico o Windows 3.1).
- **Preventiva:** el sistema operativo decide cuándo detener un proceso y pasar al siguiente, garantizando que ninguno monopolice la CPU. Es el modelo actual en Windows, Linux y macOS.

En un entorno multitarea moderno, un fallo en una aplicación no debería afectar a las demás. Por eso puedes seguir escuchando música mientras descargas un archivo y editas un documento: el sistema reparte los recursos de forma ordenada.

> **Curiosidad:** en los primeros ordenadores personales, mover el ratón y escuchar música al mismo tiempo era impensable. Hoy lo damos por hecho porque la multitarea preventiva se ha convertido en la norma.

2.6.3. Por propósito (general, tiempo real, embebido, distribuido)

No todos los sistemas operativos sirven para lo mismo. Podemos clasificarlos también según la finalidad para la que fueron diseñados.

De propósito general

Son los que usamos a diario en ordenadores personales, portátiles o servidores. Permiten ejecutar gran variedad de programas: ofimática, desarrollo, diseño, navegación, comunicación, etc. Su objetivo es el equilibrio entre funcionalidad, rendimiento y compatibilidad.

> **Ejemplos**: Windows, macOS, GNU/Linux, ChromeOS.

Se caracterizan por:

- Entornos gráficos completos.
- Soporte para multitud de periféricos.
- Actualizaciones frecuentes.
- Posibilidad de ejecutar aplicaciones de terceros.

Estos sistemas son polivalentes y configurables, lo que los hace idóneos para la enseñanza, la empresa o el uso doméstico.

De tiempo real (RTOS)

Los sistemas de tiempo real están diseñados para responder a los eventos en un tiempo máximo garantizado. Aquí no basta con que “funcione”: debe hacerlo con precisión temporal. Un retraso de un segundo puede ser catastrófico.

Se utilizan en aviación, robótica, automatización industrial, automóviles, equipos médicos o sistemas de defensa.

Estos sistemas priorizan la predictibilidad sobre la potencia. Están optimizados para ejecutarse en dispositivos con pocos recursos, y cada proceso tiene un tiempo asignado exacto.

Ejemplos de RTOS: VxWorks, QNX, FreeRTOS, o el sistema de control de vuelo de los Airbus A320.

> Ejemplo: cuando el sensor de un airbag detecta una colisión, el software de tiempo real debe activar el mecanismo en milisegundos. No puede esperar a que el sistema “termine otra tarea”.

Embebidos

Los sistemas embebidos están integrados dentro de otros dispositivos electrónicos, formando parte de su hardware. No son visibles para el usuario, pero controlan su funcionamiento interno.

Se utilizan en electrodomésticos, vehículos, relojes, routers, robots industriales o cámaras. No ejecutan programas externos: su software está diseñado para una tarea concreta y permanece inalterable durante años.

Ejemplo: el firmware de una lavadora que controla el motor, el llenado de agua y la temperatura. Su prioridad es la estabilidad, no la flexibilidad. Algunos sistemas embebidos modernos (como los de los coches eléctricos o drones) ya incluyen pequeñas variantes de Linux adaptadas al hardware, combinando estabilidad con capacidad de actualización.

Distribuidos

Un sistema operativo distribuido gestiona un conjunto de ordenadores como si fueran uno solo. El usuario o la aplicación no necesitan saber dónde se ejecuta cada proceso: el sistema reparte la carga entre varios nodos conectados en red.

Este tipo de sistemas es habitual en centros de datos, supercomputadores y servicios en la nube. Por ejemplo, los servidores de Google o Amazon, ejecutan miles de procesos distribuidos entre máquinas en diferentes países.

Las ventajas son enormes: escalabilidad, tolerancia a fallos y rendimiento cooperativo. Si un nodo falla, otro puede asumir su carga sin que el usuario lo note.

> **Ejemplo***:* cuando ves un vídeo en YouTube, no proviene de un solo servidor, sino del nodo más cercano y menos saturado.

2.6.4. Por interfaz (CLI, GUI, VUI)

La interfaz es el punto de contacto entre el usuario y el sistema operativo. Según su diseño y propósito, distinguimos tres grandes tipos:

CLI (Command Line Interface)

La interfaz de línea de comandos es textual. El usuario escribe órdenes que el sistema interpreta y ejecuta. Requiere conocimientos técnicos, pero ofrece precisión, rapidez y automatización.

Ejemplos:

- **Bash** en Linux.
- **PowerShell** en Windows.
- **Zsh** o **Fish** en entornos Unix.

La CLI sigue siendo esencial en entornos profesionales. Permite hacer en segundos lo que en una interfaz gráfica requeriría decenas de clics. Además, los comandos pueden encadenarse en scripts, lo que permite automatizar tareas complejas.

> **Curiosidad:** el propio sistema operativo Linux puede instalarse y configurarse completamente desde la línea de comandos, sin interfaz gráfica. Por eso sigue siendo la herramienta preferida de los administradores de sistemas.

GUI (Graphical User Interface)

La interfaz gráfica es la más extendida. Representa visualmente carpetas, ventanas, iconos y menús, permitiendo operar mediante ratón, teclado o pantalla táctil.

Fue popularizada por Apple en 1984 con el Macintosh y más tarde adoptada por Microsoft en Windows 3.0 (1990). La GUI democratizó la informática: ya no hacía falta saber comandos, bastaba con reconocer símbolos.

Hoy todas las plataformas principales (Windows, macOS, Linux con GNOME o KDE, Android, iOS) usan GUI, adaptadas a distintos tamaños y dispositivos.

VUI (Voice User Interface)

Las interfaces por voz son la evolución natural de la interacción humano-máquina. Permiten comunicarse con el sistema mediante lenguaje hablado, gracias a algoritmos de reconocimiento de voz e inteligencia artificial.

Ejemplos: Siri, Alexa, Google Assistant, Cortana. Estas interfaces se integran en móviles, coches, altavoces y electrodomésticos inteligentes.

Aunque aún no sustituyen completamente al teclado o al ratón, la tendencia apunta a entornos híbridos donde la voz, el gesto y el texto convivan.

2.7. Familias y ejemplos reales: Windows, macOS, GNU/Linux, Android, iOS, ChromeOS

Conocer los distintos tipos de sistemas operativos no basta: hay que ponerles nombre y entender su filosofía.

Cada familia de sistemas representa una forma diferente de concebir la informática, marcada por su historia, su arquitectura y su público objetivo.

Las cinco grandes familias actuales son: Windows, macOS, GNU/Linux, Android, iOS y ChromeOS.

Todas derivan —directa o indirectamente— de las ideas surgidas en los años 70 en torno a UNIX, el sistema operativo que sentó las bases de la informática moderna.

2.7.1. Windows

Origen y evolución

Windows es el sistema operativo desarrollado por Microsoft, la compañía fundada por Bill Gates y Paul Allen en 1975. Su primera versión, Windows 1.0, apareció en 1985 como una interfaz gráfica que funcionaba sobre MS-DOS. Durante años no fue un sistema operativo independiente, sino una capa visual que facilitaba el uso del ordenador mediante ventanas, iconos y menús desplegables.

El gran salto llegó con Windows 95, que integró MS-DOS, introdujo el botón de inicio, la barra de tareas y un nuevo modelo de multitarea. A partir de Windows 2000 y XP, el sistema se reescribió sobre un núcleo moderno basado en NT, mucho más estable y seguro. Ese núcleo —NT— es la base de todas las versiones actuales: Windows 10, 11 y sus ediciones Server.

Características técnicas

- Interfaz gráfica altamente integrada con el sistema.
- Compatibilidad con la mayoría del hardware y software comercial.
- Sistema de archivos NTFS con soporte de permisos, compresión y cifrado.
- Arquitectura híbrida: combina un microkernel con servicios en modo usuario.
- Amplio soporte de redes, Active Directory, virtualización (Hyper-V) y contenedores.

Ventajas

- Alta compatibilidad y facilidad de uso.
- Ecosistema enorme de aplicaciones.
- Amplio soporte profesional y técnico.

Limitaciones

- Sistema cerrado (propietario).
- Menor transparencia interna para el usuario.
- Más vulnerable a malware por su popularidad.

Usos reales

Windows domina el mercado de **escritorio** y **entornos empresariales**, desde oficinas hasta aulas. También tiene versiones específicas para servidores y dispositivos embebidos (Windows Server, Windows IoT).

> **Curiosidad:** en 2023, más del 70 % de los ordenadores personales del mundo seguían ejecutando alguna versión de Windows. Su logotipo es probablemente el icono informático más reconocido de la historia.

2.7.2. macOS

Origen y evolución

macOS (antes llamado OS X o Mac OS) es el sistema operativo de Apple, exclusivo de sus ordenadores Mac. Su historia está profundamente ligada al nacimiento de la informática personal. En 1984, el Macintosh fue el primer ordenador comercial con una interfaz gráfica intuitiva y un ratón como elemento central de control. Aquella idea, inspirada en los prototipos de Xerox PARC, revolucionó la industria y dio forma al paradigma de las ventanas, los iconos y los menús que seguimos usando.

En 2001, Apple reemplazó el sistema clásico por Mac OS X, construido sobre un núcleo UNIX llamado Darwin, que incorporaba un microkernel Mach y componentes del proyecto BSD. Desde entonces, macOS ha evolucionado con la filosofía de combinar estabilidad técnica con diseño estético.

Características técnicas

- Basado en UNIX certificado (POSIX compliant).
- Kernel híbrido Mach/BSD con excelente gestión de memoria y procesos.
- Sistema de archivos APFS (Apple File System) optimizado para SSD.
- Entorno gráfico Aqua y shell Unix integrado (zsh, bash).
- Ecosistema cerrado e integrado con hardware propio.

Ventajas

- Estabilidad y rendimiento constante.
- Seguridad por diseño y control centralizado de software.
- Integración perfecta con el ecosistema Apple (iPhone, iPad, iCloud).

Limitaciones

- Poca compatibilidad con hardware no Apple.
- Menor flexibilidad para personalización o reparación.
- Precio más elevado del hardware.

Usos reales

macOS es muy popular en entornos **creativos y profesionales**: diseño gráfico, edición de vídeo, música o desarrollo de software. Su estabilidad y fluidez visual lo convierten en un sistema de referencia en la industria multimedia.

> **Curiosidad***:* los sistemas UNIX son tan sólidos que la base técnica de macOS apenas ha cambiado en más de veinte años. Apple ha preferido evolucionar su estética y su ecosistema antes que reinventar el núcleo.

2.7.3. GNU/Linux

Origen y filosofía

GNU/Linux no es un sistema operativo único, sino una familia de sistemas basados en el kernel Linux y en herramientas del proyecto GNU. Su creador, Linus Torvalds, desarrolló el núcleo Linux en 1991 como un proyecto personal de aprendizaje. Lo publicó en Internet bajo licencia libre, y la comunidad empezó a mejorarlo colaborativamente. Desde entonces, se ha convertido en el mayor proyecto de código abierto del mundo.

Linux se distribuye en versiones llamadas distribuciones o *distros*: Ubuntu, Debian, Fedora, Arch, Red Hat, Kali, etc. Todas comparten el mismo núcleo, pero difieren en su configuración, gestor de paquetes, entorno gráfico y objetivos.

Características técnicas

- Código abierto, gratuito y totalmente modificable.
- Basado en el modelo UNIX.
- Soporte nativo de redes y sistemas multiusuario.
- Altamente configurable: puede funcionar en un superordenador o en un microcontrolador.
- Kernel monolítico modular: permite cargar y descargar componentes en tiempo real.

Ventajas

- Estabilidad y rendimiento excepcionales.
- Gran seguridad y bajo riesgo de virus.
- Comunidad activa y actualizaciones constantes.
- Versatilidad: desde servidores hasta móviles (Android se basa en él).

Limitaciones

- Curva de aprendizaje más pronunciada.
- Algunas aplicaciones comerciales no tienen versión nativa.
- Requiere conocimientos técnicos para configuraciones avanzadas.

Usos reales

Linux es el sistema operativo dominante en servidores, supercomputadores y dispositivos de red. El 100 % de los 500 superordenadores más potentes del mundo ejecutan alguna distribución de Linux. También está en routers, coches, televisores, satélites, móviles y hasta cafeteras inteligentes.

> **Curiosidad:** cada vez que haces una búsqueda en Google, ves una serie en Netflix o usas WhatsApp, tu solicitud pasa por decenas de servidores que corren sobre Linux.

2.7.4. Android

Origen y relación con Linux

Android es un sistema operativo móvil desarrollado por Google, basado en el núcleo Linux pero con una arquitectura adaptada al entorno táctil. Fue lanzado oficialmente en 2008 y se convirtió rápidamente en el sistema dominante en el mercado de smartphones.

Android utiliza una máquina virtual llamada ART (Android Runtime) que permite ejecutar aplicaciones escritas en Java o Kotlin. Cada app funciona en su propio entorno aislado, lo que mejora la seguridad y la estabilidad.

Características técnicas

- Núcleo Linux adaptado a procesadores ARM.
- Arquitectura modular con servicios propios (telefónica, sensores, GPS, cámara).
- Soporte multitarea y multiusuario.
- Interfaz táctil optimizada.
- Integración con los servicios de Google.

Ventajas

- Abierto y ampliamente personalizable.
- Compatible con una enorme variedad de dispositivos y marcas.
- Ecosistema gigantesco de aplicaciones (Google Play).

Limitaciones

- Fragmentación: versiones distintas en cada fabricante.
- Dependencia de Google y sus servicios.
- Vulnerabilidad a malware en tiendas no oficiales.

Usos reales

Android no se limita a los teléfonos: también se utiliza en tablets, televisores, coches, relojes, cámaras y sistemas domóticos. Es el sistema operativo más extendido del planeta, con miles de millones de dispositivos activos.

> **Curiosidad***:* cada versión de Android se nombraba con postres (Cupcake, Donut, KitKat, Oreo…) hasta Android 9. A partir de Android 10, Google decidió abandonar esa tradición para mantener una imagen más profesional.

2.7.5. iOS

Origen y evolución

iOS es el sistema operativo móvil de Apple, lanzado en 2007 junto con el primer iPhone. Está basado en la misma arquitectura que macOS (Darwin/UNIX), pero adaptado a pantallas táctiles y a hardware móvil.

Características técnicas

- Kernel híbrido Mach/BSD.
- Sistema de archivos APFS.
- Arquitectura cerrada y control total de Apple sobre hardware y software.
- Ejecución aislada de aplicaciones mediante *sandboxing*.
- Alto nivel de optimización energética y de rendimiento gráfico.

Ventajas

- Gran fluidez, estabilidad y eficiencia.
- Ecosistema controlado que reduce el riesgo de malware.
- Integración impecable con los servicios Apple (iCloud, AirDrop, Continuity).

Limitaciones

- Sistema cerrado: sin instalación externa de apps fuera de App Store (sin *jailbreak*).
- Menor personalización y mayor dependencia de Apple.

Usos reales

iOS domina el mercado premium de smartphones y tablets. También sirve de base para otros sistemas de la marca: iPadOS, watchOS y tvOS. Su diseño busca consistencia y seguridad más que variedad.

> **Curiosidad:** el “deslizamiento para desbloquear” de las primeras versiones de iOS fue tan icónico que Apple lo registró como patente; hoy forma parte de la historia del diseño de interfaces.

2.7.6. ChromeOS

Origen y filosofía

ChromeOS es el sistema operativo de Google orientado a la nube. Apareció en 2011 y está basado en el núcleo de Linux y en el navegador Chrome. Su concepto rompe con el modelo tradicional: el sistema operativo no es un entorno pesado con decenas de programas instalados, sino una plataforma para ejecutar aplicaciones web.

Características técnicas

- Núcleo Linux + entorno minimalista basado en Chrome.
- Arranque ultrarrápido y consumo mínimo.

- Actualizaciones automáticas y sistema de archivos protegido.
- Sin necesidad de instalación de programas locales (todo se ejecuta en la nube).
- Integración total con la cuenta de Google.

Ventajas

- Simplicidad y bajo mantenimiento.
- Ideal para educación, teletrabajo y entornos de oficina.
- Muy seguro frente a virus.

Limitaciones

- Depende de la conexión a Internet.
- Funcionalidad limitada sin acceso a la nube.
- Escasa capacidad para ejecutar software profesional complejo.

Usos reales

ChromeOS se utiliza sobre todo en el sector educativo y en empresas que necesitan equipos ligeros, baratos y seguros: los famosos Chromebooks. Es un ejemplo claro de hacia dónde se dirige la informática: menos instalación local y más servicios distribuidos.

> **Curiosidad***:* ChromeOS puede ejecutar ahora aplicaciones Android y Linux, lo que le ha permitido romper su antiguo límite y convertirse en un sistema híbrido entre escritorio y móvil.

Todos estos sistemas —Windows, macOS, Linux, Android, iOS y ChromeOS— tienen filosofías distintas, pero comparten un mismo objetivo: convertir hardware en utilidad. Cada uno responde a un contexto histórico y a un tipo de usuario. Windows democratizó la informática personal, macOS la hizo estética y estable, Linux la hizo libre, Android la hizo ubicua, iOS la hizo coherente y ChromeOS la llevó a la nube.

En definitiva, entender estas familias no es solo cuestión de nombres: es comprender cómo cada sistema operativo refleja una forma de pensar la tecnología. Porque detrás de cada interfaz hay decisiones de ingeniería, de negocio y de cultura que determinan nuestra manera de relacionarnos con los ordenadores.

2.8. Arquitecturas internas y modelos de kernel (monolítico, microkernel, híbrido)

Detrás de cualquier sistema operativo moderno se esconde una arquitectura interna cuidadosamente diseñada para coordinar millones de operaciones por segundo sin que el usuario perciba el más mínimo desfase. Comprender esa arquitectura es entender cómo se mantiene el orden en el caos: cómo se comunican los procesos, cómo se aíslan los errores y por qué un sistema puede seguir funcionando aunque falle una parte de él.

El corazón de esa arquitectura es el kernel, o núcleo del sistema operativo. En torno a él se organizan los demás componentes: bibliotecas, servicios, controladores y la interfaz de usuario. La forma en que estas piezas se interconectan define el tipo de arquitectura del sistema.

2.8.1. Estructura básica de un sistema operativo

Un sistema operativo no es una aplicación más: es un sistema de capas, donde cada nivel abstrae la complejidad del anterior. De abajo a arriba, la estructura general suele ser:

1. **Hardware:** procesador, memoria, dispositivos de entrada/salida.
2. **Kernel:** gestiona los recursos físicos y ofrece servicios básicos a niveles superiores.
3. **Bibliotecas y servicios del sistema:** funciones reutilizables que los programas llaman mediante APIs (Application Programming Interfaces).
4. **Shell o entorno de usuario:** interpreta los comandos o gestiona la interfaz gráfica.
5. **Aplicaciones:** software que utiliza las funciones del sistema operativo para realizar tareas concretas.

El diseño de un sistema operativo busca **equilibrar tres objetivos**:

- **Rendimiento** (responder rápido).
- **Estabilidad** (evitar bloqueos).
- **Seguridad** (proteger los recursos).

El modo en que se organiza el kernel —qué tareas asume directamente y cuáles delega— determina el comportamiento del sistema entero.

2.8.2. Modelos de kernel

Los modelos de kernel son distintas formas de organizar el núcleo del sistema operativo. Aunque todos cumplen las mismas funciones básicas (gestión de procesos, memoria, dispositivos y llamadas al sistema), difieren en cómo distribuyen esas funciones y en cómo se comunican entre sí los distintos módulos.

Kernel monolítico

Es el modelo más antiguo y, aún hoy, el más extendido. En un kernel monolítico, todas las funciones del sistema —gestión de memoria, controladores, sistema de archivos, red, planificación, etc.— residen dentro de un único bloque de código que se ejecuta en modo privilegiado (*modo kernel*).

Ventajas:

- Rendimiento muy alto (no hay pasos intermedios ni conmutaciones de contexto).
- Comunicación interna más rápida.
- Simplicidad conceptual.

Inconvenientes:

- Si un módulo falla, puede colapsar todo el sistema.
- Las actualizaciones o correcciones requieren recompilar el núcleo completo.

- Difícil de mantener en sistemas grandes.

> **Ejemplo***:* el kernel de **Linux** es monolítico, aunque admite módulos cargables dinámicamente (*Loadable Kernel Modules*), lo que permite añadir controladores sin reiniciar. Es un ejemplo de cómo se ha modernizado un modelo clásico para hacerlo flexible.

Microkernel

El enfoque opuesto. En un microkernel, solo las funciones esenciales del sistema operativo se ejecutan en modo núcleo: gestión de procesos, comunicación entre procesos (IPC) y administración de memoria básica. Todo lo demás —sistemas de archivos, drivers, interfaz, servicios— se ejecuta en modo usuario, como procesos independientes.

Ventajas:

- Mayor estabilidad y seguridad: un fallo en un servicio no compromete el núcleo.
- Mantenimiento más sencillo.
- Estructura modular y escalable.

Inconvenientes:

- Menor rendimiento: hay más pasos de comunicación entre procesos.
- Mayor complejidad en el diseño del sistema de mensajes.

> ***Ejemplo:*** el sistema Minix, diseñado por Andrew Tanenbaum como herramienta educativa, fue el primer microkernel moderno y sirvió de inspiración a Linus Torvalds para crear Linux. También lo usan macOS (kernel XNU) y QNX, muy empleado en automoción.

Kernel híbrido

Los **kernels híbridos** buscan combinar lo mejor de ambos mundos: la velocidad del monolítico y la estabilidad del microkernel. El núcleo mantiene las funciones esenciales en modo kernel, pero permite ejecutar algunos servicios como procesos separados cuando conviene.

Ventajas:

- Buen equilibrio entre rendimiento y modularidad.
- Capacidad de aislamiento ante errores sin sacrificar velocidad.

Inconvenientes:

- Mayor complejidad en la implementación.
- Dificultad para mantener la coherencia entre los módulos internos.

Ejemplo*:* Windows NT, macOS (Darwin) y Android usan arquitecturas híbridas. El kernel de Windows, por ejemplo, ejecuta en modo núcleo componentes como el planificador y el gestor de memoria, pero delega otros —como los servicios de usuario y gráficos— a procesos separados.

Nanokernel y Exokernel (modelos experimentales)

Los nanokernels llevan la idea de simplificación al extremo: el núcleo solo gestiona interrupciones y comunicación básica, dejando toda la lógica de gestión a procesos externos. Su aplicación práctica es limitada, pero se usa en sistemas de investigación o en entornos donde se requiere un control absoluto del hardware.

El exokernel, propuesto en el MIT, va un paso más allá: elimina la abstracción tradicional de recursos. En lugar de ofrecer una capa uniforme, permite que cada aplicación gestione directamente el hardware a través de una interfaz segura. Es una idea poderosa pero difícil de aplicar en entornos comerciales.

> **Curiosidad***:* el concepto de exokernel influyó en los hipervisores modernos (como VMware o KVM), donde los sistemas invitados acceden directamente al hardware virtualizado bajo supervisión.

2.9. Mecanismos de comunicación: interrupciones, llamadas al sistema, planificadores

Sea cual sea su arquitectura, todos los sistemas operativos necesitan mecanismos eficientes para comunicarse internamente y coordinar el acceso a los recursos. Estos mecanismos son los cimientos invisibles que hacen que los programas parezcan convivir pacíficamente cuando, en realidad, compiten por los mismos recursos.

2.9.1. Interrupciones

Una interrupción es una señal que informa al procesador de que algo requiere su atención inmediata. Puede generarla un dispositivo (por ejemplo, el teclado al pulsar una tecla) o el propio software (una excepción o error).

El procesador detiene temporalmente la tarea que estaba ejecutando, guarda su estado y atiende la interrupción mediante un programa especial: el manejador de interrupciones. Una vez resuelta, el sistema retoma la ejecución donde la dejó.

Las interrupciones son esenciales para la eficiencia: sin ellas, la CPU tendría que estar consultando constantemente cada dispositivo para saber si necesita algo (lo que se llama *polling*). Gracias a las interrupciones, la CPU puede concentrarse en tareas útiles hasta que algo requiera atención.

> **Ejemplo:** cuando insertas una memoria USB, el sistema operativo no está “esperando” activamente que ocurra. Es el dispositivo quien envía una interrupción al bus del sistema, y el SO responde cargando el driver correspondiente.

2.9.2. Llamadas al sistema (System Calls)

Las llamadas al sistema son el puente entre las aplicaciones y el kernel. Cuando un programa necesita realizar una operación que involucra al hardware (leer un archivo, crear un proceso, acceder a la red), no puede hacerlo directamente: debe pedírselo al sistema operativo.

Esa petición se realiza mediante una llamada al sistema, que pasa del modo usuario al modo kernel. El núcleo ejecuta la operación y devuelve el resultado. Este mecanismo mantiene la seguridad y el aislamiento: las aplicaciones nunca acceden directamente al hardware.

> **Ejemplo***:* cuando en Python escribes `open("archivo.txt")`, el intérprete no abre el archivo por sí mismo: invoca una llamada al sistema (`sys_open` en Linux, `CreateFile` en Windows), que realiza la operación a nivel de kernel.

2.9.3. Planificadores (Schedulers)

El planificador es el componente que decide qué proceso usa la CPU en cada instante. Funciona como un árbitro: reparte el tiempo de procesador entre cientos de procesos activos según su prioridad, estado y tipo de tarea.

Hay varios niveles de planificación:

- **Planificación a largo plazo:** decide qué procesos entran en el sistema.
- **A medio plazo:** gestiona la suspensión o reactivación de procesos.
- **A corto plazo:** asigna la CPU en intervalos de milisegundos.

Los sistemas modernos combinan distintos algoritmos según el contexto: por ejemplo, un servidor puede priorizar procesos de red, mientras un PC de sobremesa prioriza los que afectan a la interfaz gráfica.

> **Curiosidad***:* el planificador de Linux, llamado CFS (Completely Fair Scheduler), intenta repartir la CPU de forma equitativa entre todos los procesos, simulando que cada uno dispone de su propio procesador virtual.

2.10. Estabilidad, mantenimiento y logs del sistema

La estabilidad de un sistema operativo depende tanto de su arquitectura como de su capacidad de detectar, registrar y recuperar errores.

Los sistemas modernos implementan varias estrategias:

- **Aislamiento de procesos:** evita que un fallo afecte a otros.
- **Protección de memoria:** cada proceso tiene su espacio propio.
- **Supervisión de servicios:** si un demonio falla, otro lo reinicia automáticamente.
- **Logs del sistema:** registros detallados de cada evento relevante (inicio, errores, advertencias, actualizaciones).

Los logs son esenciales para la administración profesional. Permiten diagnosticar fallos, auditar accesos y detectar comportamientos anómalos. En Windows están en el Visor de eventos; en Linux, en `/var/log/syslog` o `/journalctl`. Un administrador experimentado puede leer esos registros como un médico lee un electrocardiograma.

> **Ejemplo:** en 1997, la sonda Mars Pathfinder dejó de responder a la NASA durante horas. El fallo se debió a un error de *prioridad de procesos* (una condición de carrera). Gracias al análisis de los logs, los ingenieros detectaron el problema y lo solucionaron con un simple parche de software… desde 190 millones de kilómetros de distancia.

2.11. Reflexión: por qué ningún software puede existir sin un sistema operativo

Todo software necesita un entorno donde vivir. El sistema operativo no es un programa más, sino el ecosistema que da sentido a todos los demás. Gestiona la energía, el tiempo, el espacio y las reglas del juego digital. Sin él, ni el mejor código del mundo podría ejecutarse.

Podríamos decir que el sistema operativo es la infraestructura invisible sobre la que se construye toda la tecnología moderna. Es el equivalente digital de un sistema nervioso: no se ve, pero si falla, todo el organismo colapsa. Y, paradójicamente, cuanto mejor funciona, menos se nota que existe.

Para reflexionar el sistema operativo no solo organiza recursos, organiza la confianza. Cuando abrimos un archivo, enviamos un mensaje o hacemos una compra online, confiamos en que el sistema hará exactamente lo que esperamos, sin error ni traición. Esa fiabilidad no es magia: es ingeniería.

3. Software de aplicación y software de desarrollo

3.1. Introducción: del usuario al creador

Cuando un usuario enciende su ordenador, lo primero que ve no son bits ni comandos, sino herramientas: un navegador, un editor de texto, un juego, una hoja de cálculo. Todo eso es software de aplicación, creado por alguien con conocimientos técnicos que ha sabido traducir una necesidad humana en un conjunto de instrucciones para una máquina.

Esa persona (o equipo) que crea aplicaciones es precisamente lo que estás formándote para ser en el ciclo de Desarrollo de Aplicaciones Web (DAW). Así que, en esta parte, daremos un paso simbólico: dejamos de ser usuarios del software para convertirnos en productores.

> **Idea clave:** Todo software de aplicación existe porque alguien, antes, escribió software de desarrollo. Piensa en un carpintero. Usa herramientas (sierra, martillo) para fabricar muebles. En informática, el software de desarrollo son esas herramientas, y el software de aplicación es el mueble que construyes con ellas.

3.2. Software de aplicación

El software de aplicación es el conjunto de programas diseñados para realizar tareas concretas que resuelven las necesidades del usuario. Se apoya sobre el sistema operativo y el software de sistema, que le proporcionan los recursos necesarios (memoria, almacenamiento, dispositivos de entrada/salida…). Sin el sistema operativo, las aplicaciones no tendrían dónde ejecutarse; sin aplicaciones, el usuario no tendría motivos para usar el ordenador.

Las aplicaciones cubren todos los ámbitos imaginables: desde la productividad y la gestión de empresas hasta la creatividad, el ocio o la educación.

Por ejemplo:

- Escribir un documento → Microsoft Word, LibreOffice Writer, Google Docs.
- Navegar por Internet → Chrome, Firefox, Safari.
- Editar una foto → GIMP, Photoshop, Canva.
- Escuchar música → Spotify, VLC, Audacity.
- Gestionar una empresa → ERP, CRM, gestor de inventario

Hoy existen millones de aplicaciones, pero todas comparten tres características fundamentales:

- **Tienen un propósito definido.** Resuelven un problema o facilitan una tarea: escribir un texto, editar una imagen, comunicar a personas, analizar datos…
- **Dependen del software de sistema.** Usan las funciones del sistema operativo, los controladores y las librerías del sistema.
- **Ofrecen una interfaz de usuario.** Ya sea gráfica o por comandos, permiten interactuar de forma sencilla con operaciones complejas.

3.2.1. Propósito, ejemplos, tendencias actuales

El software de aplicación se puede clasificar según el propósito que cumple:

Tipo de aplicación	Ejemplo	Finalidad
Ofimática	Microsoft 365, LibreOffice	Crear y editar documentos, hojas de cálculo, presentaciones.
Diseño y multimedia	Adobe Photoshop, GIMP, DaVinci Resolve	Edición de imágenes, audio y vídeo.
Gestión empresarial	SAP, Odoo, Sage	Control de facturación, recursos humanos, inventario.
Educativo	Moodle, Kahoot, Scratch	Aprendizaje y evaluación digital.
Comunicaciones	Outlook, Slack, WhatsApp Web	Intercambio de mensajes, trabajo colaborativo.
Entretenimiento	Steam, Netflix, Spotify	Juegos, películas, música.

Tendencias actuales

- Aplicaciones en la nube: funcionan desde el navegador (Google Workspace, Canva, Figma).

- Modelos de suscripción: el software deja de venderse como producto y pasa a ofrecerse como servicio (Software as a Service).
- Integración multiplataforma: los mismos datos y funciones disponibles en PC, móvil y web.
- IA y automatización: asistentes inteligentes, generación automática de contenido, personalización adaptativa.

> **Curiosidad:** El término “app”, tan popular hoy, se generalizó a partir de 2008 con la creación de la App Store de Apple. Antes se hablaba simplemente de “programas”.

3.2.2. Aplicaciones locales, web, móviles, PWA

Según el entorno en el que se ejecuten, las aplicaciones pueden clasificarse en:

Tipo	Características	Ejemplo	Ventajas	Limitaciones
Locales (desktop)	Se instalan en el equipo del usuario.	VLC, Photoshop	Rápidas, funcionan sin conexión, aprovechan el hardware.	Dependientes del sistema operativo.
Web	Se ejecutan desde un navegador.	Google Docs, Trello	Accesibles desde cualquier dispositivo.	Requieren conexión y navegador compatible.
Móviles	Diseñadas para smartphones o tablets.	WhatsApp, TikTok	Portátiles, optimizadas para pantalla táctil.	Limitadas por batería y espacio.
PWA (Progressive Web Apps)	Webs que se comportan como apps nativas.	Twitter Web, Spotify Web	No necesitan instalación, funcionan offline parcialmente.	No todas las APIs del sistema están disponibles.

> **Caso práctico:**
> Piensa en Microsoft Teams. Existe como app de escritorio, app móvil y versión web.
> Detrás hay un mismo servicio en la nube, pero tres interfaces adaptadas al contexto del usuario.
>
> **Analogía:** Si una aplicación fuera una tienda, la versión de escritorio sería la tienda física, la versión web sería la tienda online, y la PWA sería un híbrido que puedes visitar incluso sin conexión total.

3.2.3. Suites integradas y ecosistemas (Office, Adobe, Google Workspace)

Una suite de software es un conjunto de aplicaciones que comparten una interfaz común y se integran entre sí. Ejemplos clásicos son Microsoft 365, Google Workspace o Adobe Creative Cloud. Estas suites representan un cambio de paradigma: el usuario ya no trabaja con programas aislados, sino dentro de un ecosistema de servicios que sincronizan información entre dispositivos y usuarios.

> **Ejemplo:** Una hoja de cálculo en Google Sheets puede actualizar en tiempo real un informe de Google Docs y generar una presentación en Slides, todo conectado por la nube.

3.3. Software de desarrollo

Si el software de aplicación es el destino visible, el software de desarrollo es el camino que permite construirlo. Son las herramientas que los programadores utilizan para escribir, traducir, probar y desplegar aplicaciones.

Podemos considerarlo el equivalente digital a un taller de ingeniería.

3.3.1. Editores, compiladores, intérpretes

- Editores de texto: Son el punto de partida. Permiten escribir código en distintos lenguajes y resaltan la sintaxis. Algunos ofrecen autocompletado o depuración ligera.
  Ejemplos: VS Code, Sublime Text, Notepad++.
- Compiladores: Traducen el código fuente a lenguaje máquina o intermedio. Detectan errores antes de ejecutar y generan archivos binarios listos para correr.
  Ejemplo: GCC (C/C++), javac (Java).
- Intérpretes: Ejecutan el código línea a línea, sin compilarlo previamente. Permiten una programación más ágil y experimental.
  Ejemplo: Python, Node.js, PHP.

> **Nota:** Lenguajes como Java o C# usan un modelo mixto: compilan a un código intermedio (bytecode) que luego interpreta una máquina virtual

3.3.2. IDEs modernos (VS Code, IntelliJ, Eclipse)

Un IDE (Integrated Development Environment) es una herramienta que agrupa editor, compilador, depurador y gestor de proyectos en un solo entorno. Es la evolución natural de los programas anteriores.

Los IDE modernos ofrecen:

- Resaltado de sintaxis y autocompletado inteligente (IntelliSense).
- Detección de errores en tiempo real.
- Integración con control de versiones (Git).
- Emuladores y entornos de prueba integrados.
- Extensiones y personalización.

Ejemplos:

- Visual Studio Code: gratuito, multiplataforma, extensible.
- IntelliJ IDEA / PyCharm / WebStorm: de JetBrains, muy potentes para Java, Python o JavaScript.
- Eclipse: veterano, ampliamente usado en entornos empresariales.
- Android Studio: especializado en desarrollo móvil.

> **Caso práctico:**
> Imagina que vas a crear una web con HTML, CSS y JavaScript.
> En VS Code puedes abrir el proyecto, escribir el código con autocompletado, ejecutar un servidor local y usar Git sin salir del entorno. Todo en una sola ventana. Eso es la eficiencia de un IDE moderno.
>
> **Curiosidad tecnológica:**
> El término IDE surgió en los años 80, cuando empresas como Borland integraron el editor y compilador en una única interfaz (Turbo Pascal). Hoy, Visual Studio Code y JetBrains continúan esa filosofía, pero en la nube.

3.3.3. Frameworks, librerías y APIs

El desarrollo moderno sería impensable si cada programador tuviera que reinventar la rueda. Por eso existen los frameworks, las librerías y las APIs: piezas de software que otros ya construyeron para que tú puedas concentrarte en crear lo nuevo.

> **Analogía:**
> Si construir una aplicación fuera levantar un edificio, el lenguaje de programación sería el cemento, las librerías serían los ladrillos prefabricados, y el framework sería el plano que te indica cómo encajarlos.

Librerías

Una librería es un conjunto de funciones o clases reutilizables que resuelven problemas comunes. Por ejemplo, en lugar de programar desde cero cómo mostrar una fecha o calcular una raíz cuadrada, se importa una librería que ya lo hace.

Ejemplo:

- En Python, la librería math incluye funciones matemáticas.
- En JavaScript, React utiliza librerías como axios o moment.js para manejar datos o fechas.

Las librerías son modulares y flexibles, ideales cuando el desarrollador quiere decidir cómo combinarlas.

Frameworks

Un framework es un conjunto más amplio y estructurado de herramientas, librerías y reglas que definen cómo debe organizarse un proyecto. No solo ofrece funciones: marca el camino.

Ejemplo**:**

- Django (Python): impone la arquitectura MVC y facilita el desarrollo web.
- Laravel (PHP): ofrece gestión de rutas, plantillas, bases de datos y autenticación.
- Angular (JavaScript): estructura el código en componentes reutilizables.

APIs (Application Programming Interface)

Las APIs son “puentes” que permiten que diferentes programas se comuniquen entre sí. En lugar de dar acceso directo al código interno, exponen solo las funciones necesarias..

Ejemplo**:**

- Google Maps API permite insertar mapas interactivos en tu web sin tener que programar la cartografía.
- Twitter API permite publicar tweets desde una app externa.

> **Caso práctico:**
> Imagina que estás creando una app de viajes.
> Usas una API de clima para mostrar la temperatura, una API de vuelos para buscar trayectos y otra de Google Maps para la ubicación.
> Tu app es el punto de unión de múltiples servicios que cooperan entre sí.

3.3.4. Control de versiones (Git, GitHub, GitLab)

En los inicios de la programación, los equipos trabajaban sobre los mismos archivos en un disco compartido. Si dos personas editaban el mismo código, los cambios se pisaban entre sí. Hoy eso sería un caos. Por eso existe el control de versiones, una de las competencias esenciales del desarrollador moderno.

Git: el estándar de facto

Git es un sistema de control de versiones distribuido creado por Linus Torvalds (sí, el mismo de Linux) en 2005. Permite que varios programadores trabajen en paralelo sobre un mismo proyecto sin perder la historia de los cambios.

- Cada desarrollador tiene una copia completa del repositorio en su ordenador.
- Los cambios se registran mediante *commits* (como “puntos de guardado”).
- Git puede “fusionar” modificaciones de distintos usuarios y resolver conflictos.

> 💡 **Analogía:**
> Git es como un “historial de Google Docs”, pero para código.
> Puedes volver atrás, comparar versiones o recuperar cualquier estado anterior del proyecto.

GitHub, GitLab y otros repositorios remotos

Los servicios como GitHub, GitLab o Bitbucket son plataformas que alojan repositorios Git en la nube.

Añaden colaboración, documentación, gestión de incidencias y visibilidad pública o privada.

Plataforma	Características destacadas
GitHub	Propiedad de Microsoft. Mayor comunidad open source del mundo. Ideal para proyectos públicos y portfolios profesionales.
GitLab	Permite despliegue, integración continua y repositorios privados gratuitos. Muy usado en entornos empresariales.
Bitbucket	Integración nativa con Jira y Trello. Orientado a equipos de empresa.

> 🧠 **Caso práctico:**
> En DAW, si trabajas en grupo en una web, cada miembro puede crear su rama (`branch`), hacer sus cambios y luego fusionarlos (`merge`) con la principal (`main`).
> Git registra quién hizo qué, cuándo y por qué.

3.3.5. Integración continua y despliegue

Antes, los equipos desarrollaban durante meses y solo al final juntaban todo el código. El resultado solía ser un desastre: errores ocultos, incompatibilidades y estrés. De ahí nació la integración continua (CI), una práctica que consiste en combinar y probar el código con frecuencia, preferiblemente varias veces al día.

Integración continua (CI)

Cada vez que un desarrollador sube código a GitHub o GitLab:

1. Se ejecutan pruebas automáticas.
2. Se comprueba la compilación.
3. Se genera un informe.

Si algo falla, el sistema avisa al equipo inmediatamente. Esto evita el temido “funcionaba en mi ordenador”.

> Herramientas habituales:
> Jenkins, GitHub Actions, GitLab CI/CD, Travis CI, CircleCI.

Despliegue continuo (CD)

El siguiente paso es automatizar también el despliegue: que la nueva versión se publique sola si pasa las pruebas. Así se consiguen actualizaciones rápidas, seguras y constantes.

- En entornos web, el código puede desplegarse automáticamente en servidores como AWS, Azure o Vercel.
- En entornos educativos, puede configurarse para que al subir una práctica, se despliegue en un servidor de pruebas.

> **Analogía:**
> Imagina una fábrica de coches donde cada vehículo pasa por una cinta de montaje, control de calidad y entrega automática. CI/CD es eso mismo, pero para el software.

3.4. Ecosistemas actuales de desarrollo: cloud, contenedores, DevOps

El desarrollo moderno no termina al compilar el programa: el software debe vivir en un entorno, ser escalable, actualizable y accesible desde cualquier lugar.

De ahí nacen tres pilares: la nube, los contenedores y DevOps.

Cloud Computing

El cloud computing o computación en la nube consiste en usar recursos de otros servidores a través de Internet. Ya no necesitas tener tus propios equipos físicos: puedes alquilar capacidad de cómputo, almacenamiento o bases de datos.

Tipos de servicios cloud:

- **IaaS (Infraestructura como servicio):** Amazon EC2, Google Compute Engine.
- **PaaS (Plataforma como servicio):** Heroku, Render, Google App Engine.
- **SaaS (Software como servicio):** Gmail, Canva, Slack.

> **Ejemplo DAW:**
> Subir una web a Vercel o Netlify en lugar de mantener tu propio servidor.
> Pagas solo por lo que usas y puedes escalar fácilmente.

Contenedores (Docker)

Un contenedor empaqueta una aplicación y todas sus dependencias en un mismo entorno portátil. Así, “funciona igual” en cualquier equipo.

- **Docker** es la herramienta más conocida.
- Los contenedores son ligeros, rápidos y reproducibles.
- Evitan el clásico “en mi máquina iba bien”.

> **Analogía:**
> Un contenedor Docker es como una fiambrera: guarda tu aplicación con todos sus ingredientes listos para usar, y puedes llevarla a cualquier cocina (servidor).

DevOps

DevOps une *Development* (desarrollo) y *Operations* (operaciones). Es una filosofía de trabajo colaborativo donde los equipos de desarrollo y de infraestructura trabajan juntos desde el principio.

Sus objetivos son:

- Automatizar procesos.
- Reducir errores humanos.
- Acortar el tiempo entre una idea y su puesta en producción.

Herramientas comunes: Docker, Kubernetes, Jenkins, Terraform, GitHub Actions.

> **Analogía:** Si antes los desarrolladores eran los “cocineros” y los de sistemas los “camareros”, DevOps une cocina y servicio: todos colaboran para que el plato llegue perfecto a la mesa.

3.5. El rol del técnico superior en DAW como productor de software

Llegados aquí, ya no hablamos de usuarios de tecnología, sino de profesionales que la crean, la mantienen y la mejoran. El técnico superior en Desarrollo de Aplicaciones Web se convierte en productor de software, no solo en su programador.

Sus funciones combinan la técnica con la visión global del producto:

Competencia	Ejemplo práctico
Diseñar soluciones	Traducir las necesidades del cliente en una aplicación funcional.
Implementar código eficiente	Usar buenas prácticas de programación y control de versiones.
Colaborar en equipo	Integrar su trabajo con el de otros desarrolladores y diseñadores.
Mantener y mejorar	Actualizar dependencias, optimizar rendimiento, corregir errores.
Documentar y versionar	Crear documentación técnica y usar Git de forma profesional.
Asegurar la calidad	Aplicar pruebas automáticas y CI/CD.

> **Ejemplo:**
> En una empresa web, podrías estar encargado de integrar un sistema de reservas online. Tu tarea no sería solo programar las páginas, sino también configurar el servidor, desplegar actualizaciones y mantener la base de datos segura.

El técnico DAW no es un operario del código: es un **constructor digital**, un eslabón esencial entre las ideas y la realidad tecnológica.

3.6. Reflexión: “Desarrollar es crear herramientas para crear herramientas”

El desarrollo de software tiene una naturaleza única: cada programa que se crea puede ser usado para construir otros programas. Un IDE, un compilador o un framework son herramientas que engendran nuevas herramientas.

Esto hace del desarrollo una de las actividades más poderosas (y también más responsables) del mundo digital. Cada línea de código puede multiplicar su impacto exponencialmente.

> **Ejemplo:**Linus Torvalds creó Git para resolver un problema puntual de control de versiones. Hoy, Git es la base de millones de proyectos de software libre y empresarial en todo el planeta. Su herramienta creó miles de herramientas más.

Por eso, el técnico superior en DAW debe verse no solo como un programador, sino como un diseñador de posibilidades: alguien que amplía los límites de lo que otros pueden hacer.

4. Licencias y modelos de distribución del software

4.1. Introducción: por qué existen las licencias

Cuando creas un programa, automáticamente eres su autor. Tu código es una obra intelectual, igual que una canción, una novela o una pintura. Y, como toda obra, está protegida por la ley.

Ahora bien, a diferencia de un libro, el software no se “consume”: se usa, se copia, se modifica y se distribuye. Y eso plantea una pregunta clave: ¿Hasta qué punto quieres que otros puedan usar, copiar o modificar tu trabajo?

Aquí entran en juego las licencias de software, que son contratos legales entre el autor y los usuarios. Definen qué se puede hacer con el programa y qué no.

4.2. Propiedad intelectual y software

La propiedad intelectual protege la autoría de las creaciones del espíritu humano. En el caso del software, se protege el código fuente, la documentación y el diseño. El autor tiene derechos morales (nadie puede atribuirse su obra) y patrimoniales (decidir cómo se explota).

En la práctica, esos derechos pueden gestionarse de dos maneras:

1. Manteniendo el control exclusivo → **software propietario**.
2. Compartiendo los derechos → **software libre o de código abierto**.

> **Curiosidad legal:** En España, la Ley de Propiedad Intelectual (RDL 1/1996) considera el software una “obra literaria”, igual que una novela. Así que, legalmente, tú y García Márquez estáis en la misma categoría… solo que tú escribes en JavaScript.

4.2.1. Licencias tradicionales

Antes de la era digital, las licencias de software eran documentos en papel que acompañaban al disquete o CD-ROM. Permitían instalar una copia del programa y prohibían modificar o redistribuir. Era la época de Microsoft DOS, Lotus 1-2-3 y los videojuegos con clave de activación en la caja.

A esas licencias se las conoce como **EULA** (*End User License Agreement*), que literalmente significa *acuerdo de licencia de usuario final*.

> **Ejemplo real:** Cuando instalas Windows o Adobe Photoshop y aparece el texto de “Acepto los términos y condiciones”, estás firmando un EULA, aunque no lo leas (como el 99 % de la humanidad).

4.2.2. Propietario, libre, copyleft, freeware, shareware, dominio público

A medida que Internet facilitó el intercambio de software, surgieron distintos modelos según el grado de libertad que conceden al usuario:

Tipo de licencia	Descripción	Ejemplo	Libertades del usuario
Propietario	El código fuente no se comparte. Solo puede usarse bajo las condiciones del autor.	Windows, Microsoft Office	Uso limitado. Prohibida la modificación o redistribución.
Libre	Permite usar, estudiar, modificar y redistribuir el programa.	Linux, LibreOffice, GIMP	Libertad total (con obligación de mantener esa libertad).
Copyleft	Variante del software libre que **obliga** a mantener la libertad en las versiones derivadas.	GNU GPL	Si mejoras el software, también debe ser libre.
Freeware	Se distribuye gratis, pero no se puede modificar ni vender.	Skype, Chrome	Gratuito, pero sin libertad de modificación.
Shareware	Se distribuye gratuitamente de forma limitada (tiempo o funciones).	WinRAR, algunos antivirus	Gratuito para probar; luego, de pago.
Dominio público	Sin derechos de autor. Cualquiera puede usarlo o modificarlo libremente.	Clásicos como el algoritmo RSA original	Total libertad.

4.2.3. Licencias modernas y modelos de distribución comercial

Con la expansión de Internet y la aparición de nuevos modelos de negocio digitales, el software dejó de venderse como un “producto” físico en caja y pasó a ofrecerse como un servicio continuo.

A la vez, surgieron distintas formas de licenciar ese acceso dependiendo del tipo de usuario, del volumen de compra o del entorno en que se utiliza.

4.2.3.1. Modelos clásicos de distribución

Licencia Retail

- Es la licencia tradicional que se vende al usuario final, generalmente en tiendas o distribuidores oficiales.
- Permite instalar el software en un único equipo, aunque a veces autoriza su transferencia a otro si se desinstala del primero.
- Ejemplo: comprar una copia individual de Microsoft Office o Photoshop en formato digital.

Ventaja: sencilla y legal para uso personal o profesional individual.
Inconveniente: no es económica para grandes volúmenes o instituciones.

Licencia por volumen (Volume Licensing)

Pensada para empresas, centros educativos o administraciones que necesitan instalar el software en muchos equipos.

- Ofrece una clave maestra o múltiples activaciones a precio reducido.
- Permite gestión centralizada, despliegues en red y soporte técnico ampliado.
- Ejemplo: licencias de Microsoft 365 Education o Windows 11 Enterprise distribuidas por KMS (Key Management Service).

Licencia OEM (Original Equipment Manufacturer)

Incluida de fábrica en los equipos nuevos. El software viene preinstalado y vinculado al hardware.

- Es más barata, pero no transferible: si cambias de equipo, pierdes la licencia.
- Ejemplo: Windows 11 preinstalado en un portátil HP o Lenovo.

Licencias educativas y académicas

Dirigidas a estudiantes, docentes y centros de formación. Ofrecen precios reducidos o uso gratuito con fines no comerciales.

- Ejemplo: Microsoft 365 A1 para centros educativos, JetBrains Student Pack, o Autodesk para formación.
- En muchos casos exigen correo institucional o verificación del centro.

Estas licencias han sido clave para democratizar el acceso al software profesional entre el alumnado de FP y universidad.

4.2.3.2. Modelos contemporáneos (era cloud y digital

La digitalización trajo consigo nuevas fórmulas para acceder y pagar el software. Hoy, lo que más se vende no es una copia, sino el derecho temporal a usar una plataforma o servicio.

SaaS (Software as a Service)

Modelo dominante en la actualidad. El usuario no instala el software, sino que accede a él desde Internet, generalmente mediante suscripción mensual o anual.

- Ejemplos: Google Workspace, Adobe Creative Cloud, Notion, Canva.
- El mantenimiento, actualizaciones y copias de seguridad dependen del proveedor.
- El usuario paga por uso o capacidad.

Ventaja: siempre actualizado, sin preocuparte por instalaciones.
Inconveniente: dependes de la conexión y del proveedor (si deja de prestar servicio, pierdes acceso).

Freemium

Combina “free” (gratuito) y “premium” (de pago). El usuario puede usar una versión básica sin coste, pero debe pagar por funciones avanzadas, almacenamiento adicional o eliminación de publicidad.

- Ejemplos: Spotify, Trello, Slack, Dropbox, Zoom.
- Estrategia común en startups tecnológicas y apps móviles.

El modelo freemium se basa en el 80/20: la mayoría usa la versión gratuita, pero el 20 % de usuarios premium sostiene el negocio.

BYOL (Bring Your Own License)

Literalmente, *“trae tu propia licencia”*. Se utiliza sobre todo en entornos cloud o virtualizados: el cliente ya posee una licencia legítima y la reutiliza al migrar su software a otro entorno (por ejemplo, de un servidor local a la nube).

- Ejemplo: una empresa que lleva su licencia de Windows Server o SQL Server a Amazon Web Services (AWS) mediante BYOL.
- Ventaja: ahorro y flexibilidad legal en migraciones a la nube.

Modelos cloud híbridos y suscripción flexible

Algunos proveedores combinan SaaS con instalación local o virtualización:

- **Microsoft 365:** parte del software (Word, Excel) se instala, pero el resto funciona en la nube.

- **JetBrains:** licencia anual con opción de renovación o uso perpetuo de la última versión activada.
- **AWS o Azure:** cobran por horas de uso o capacidad de cómputo (modelo “pay as you go”).

Tendencia actual: Las licencias dejan de ser permanentes para convertirse en derechos de uso dinámicos, ajustados al consumo y gestionados online.

Conclusión del apartado

El concepto de “licencia moderna” ya no se limita al documento legal que acompaña al software, sino que define el modo en que accedemos a la tecnología: comprándola, alquilándola o compartiéndola.

Reflexión final: En la era de la nube, el usuario ya no “posee” el software; lo utiliza como servicio. El reto ético y profesional del técnico DAW consiste en entender estos modelos para poder elegir y diseñar soluciones sostenibles, legales y justas.

4.3. Licencias open source comunes (GPL, MIT, Apache, CC)

Aunque “software libre” y “open source” se usan a menudo como sinónimos, no son exactamente lo mismo. El software libre se centra en la libertad del usuario, mientras que el open source se enfoca en la transparencia del código y la colaboración técnica.

El movimiento *open source* nació a finales de los 90, cuando empresas como Netscape y Sun Microsystems quisieron liberar parte de su código sin asociarse al activismo del software libre.

> Apunte histórico**:** En 1998, Eric S. Raymond y Bruce Perens crearon la *Open Source Initiative (OSI)*, que definió 10 principios de código abierto: redistribución libre, acceso al código fuente, integridad del autor, neutralidad tecnológica, etc.
> Gracias a eso, el modelo open source se volvió compatible con el mundo empresarial.

Comparativa:

- GPL: garantiza libertad, pero puede ser difícil de compatibilizar con software cerrado.
- MIT y Apache: más flexibles, ampliamente usadas en proyectos comerciales y educativos.
- CC (Creative Commons): usada en documentación, imágenes o recursos educativos más que en código.

> **Caso real:**
> Google usa la licencia Apache 2.0 en Android, porque le permite mantener partes propietarias (como Google Play).
> En cambio, Linux usa GPL, porque busca garantizar que siempre sea libre, incluso si alguien lo modifica.

4.4. Modelos de negocio y sostenibilidad económica

Una idea común es que el software libre “no da dinero”. Nada más lejos de la realidad. Lo que cambia no es el valor, sino cómo se genera ese valor.

Principales modelos:

1. **Servicios y soporte:** Red Hat o Canonical (Ubuntu) ofrecen soporte técnico profesional para software libre. El código es gratuito, pero el conocimiento experto se paga.
2. **Freemium:** El producto básico es gratuito, pero se cobra por funciones premium (Dropbox, Canva, Slack).
3. **Suscripción y SaaS:** Se paga una cuota mensual por usar la aplicación en la nube (Adobe Creative Cloud, Notion).
4. **Publicidad y datos:** El servicio es gratuito, pero el negocio está en los anuncios o la analítica (Google, Meta).
5. **Donaciones y crowdfunding:** Proyectos comunitarios (como Blender o Wikipedia) sobreviven gracias al apoyo voluntario.

4.5. Ética profesional y uso legal del software

El uso responsable del software es una cuestión tanto técnica como moral. Los desarrolladores deben actuar con honestidad, respetar licencias y fomentar buenas prácticas.

Buenas prácticas éticas:

- Leer y respetar las licencias antes de integrar librerías externas.
- No usar software pirateado, ni siquiera “solo para probar”.
- Citar las fuentes del código ajeno.
- Promover el conocimiento abierto siempre que sea posible.
- Informar de vulnerabilidades en lugar de explotarlas.

Caso de reflexión: Si descargas una plantilla web sin leer su licencia y la vendes como propia, no solo estás infringiendo la ley: estás dañando la reputación del sector. La ética profesional empieza en los pequeños gestos.

4.6. Caso práctico: lectura crítica de un contrato EULA

Contexto: Imagina que vas a instalar un programa gratuito llamado *PhotoMaster 3.0*. Antes de hacerlo, aparece una ventana con 8 páginas de “Términos y condiciones”.

Pocos usuarios las leen, pero un técnico DAW debe entender lo que firma.

Extracto hipotético:

> “El usuario acepta que los datos de uso anónimos podrán compartirse con terceros para mejorar la experiencia del servicio.
> La empresa no se responsabiliza de pérdidas de datos derivadas del uso del software.
> El usuario no podrá desensamblar ni modificar el código, ni redistribuir copias del programa.”

Análisis crítico:

- “Datos de uso anónimos” puede implicar rastreo.
- “No se responsabiliza de pérdidas” significa que tú asumes el riesgo.
- “No modificar ni redistribuir” indica licencia propietaria.

Conclusión:
Leer (aunque sea por encima) las cláusulas principales te protege como usuario y como profesional. Cuando desarrolles software, redactarás tú mismo tus propios términos, así que conviene conocer el terreno.

4.7. Reflexión: libertad, responsabilidad y sostenibilidad en el ecosistema digital

El software es una herramienta, pero también es un vehículo de valores. Cada elección —licencia, modelo de negocio o plataforma— tiene implicaciones éticas, sociales y económicas.

Reflexión final:

- La libertad sin responsabilidad genera caos.
- La propiedad sin ética genera desigualdad.
- El equilibrio entre ambas crea sostenibilidad.

El desarrollador del siglo XXI debe aspirar a un punto medio: crear herramientas útiles, sostenibles y éticas, que respeten tanto los derechos del autor como los del usuario.

5. Virtualización e instalación de sistemas operativos

5.1. Concepto y necesidad de virtualización

En los primeros tiempos de la informática, cada servidor tenía una única función: uno para correo, otro para bases de datos, otro para web… Esto implicaba gasto, espacio y mantenimiento. Pero ¿y si un solo equipo potente pudiera “simular” varios ordenadores independientes?

Eso es la virtualización: un proceso que permite ejecutar varios sistemas operativos y aplicaciones en un mismo hardware físico, como si cada uno tuviera su propio ordenador.

Definición: La virtualización consiste en crear una capa de abstracción entre el hardware físico y los sistemas operativos que lo usan. Esa capa permite dividir los recursos físicos (CPU, RAM, disco, red) en varios entornos virtuales independientes llamados máquinas virtuales (VM).

¿Por qué es necesaria la virtualización?

1. Ahorro de costes y energía: un servidor físico puede albergar decenas de máquinas virtuales.
2. Entornos de prueba y aprendizaje: permite experimentar sin miedo; si algo falla, se borra la VM y se empieza de nuevo.
3. Compatibilidad: ejecutar varios sistemas operativos distintos en el mismo equipo.
4. Aislamiento: un error o virus en una máquina virtual no afecta al resto.

5. Seguridad y recuperación: mediante copias de seguridad y “instantáneas” (snapshots).

> **Ejemplo:** En el ciclo de DAW, el alumnado puede tener una VM con Ubuntu Server para practicar administración Linux, otra con Windows 11 para probar software, y otra con Kali Linux para auditorías de red, y todo en el mismo ordenador.

5.2. Tipos de virtualización: completa, paravirtualización, contenedores

La palabra “virtualización” abarca varias tecnologías distintas, según el grado de aislamiento y la forma en que se gestiona el hardware.

5.2.1. Virtualización completa

Cada máquina virtual actúa como un ordenador completo, con su propio sistema operativo invitado (guest). El hipervisor emula el hardware y traduce las instrucciones al sistema físico.

- **Ejemplo**: VirtualBox, VMware Workstation.
- **Ventajas**: alta compatibilidad y aislamiento total.
- **Inconvenientes**: mayor consumo de recursos.

5.2.2. Paravirtualización

El sistema operativo invitado sabe que está virtualizado y coopera con el hipervisor para comunicarse directamente con el hardware.De este modo, se reduce la sobrecarga.

- **Ejemplo**: Xen, KVM (en modo paravirtualizado).
- **Ventajas**: más rendimiento.
- **Inconvenientes**: necesita sistemas invitados modificados para soportarlo.

5.2.3. Virtualización a nivel de sistema operativo (contenedores)

No se crean máquinas virtuales completas, sino entornos aislados que comparten el mismo kernel. Cada contenedor ejecuta aplicaciones independientes, pero todos usan el mismo sistema base.

- **Ejemplo**: Docker, LXC, Podman.
- **Ventajas**: ligeros, rápidos y portables.
- **Inconvenientes**: menor aislamiento (no se puede usar otro kernel).

> Analogía: Si la virtualización completa son varios apartamentos en un edificio,los contenedores son habitaciones en un mismo piso compartido: independientes, pero con el mismo techo.

5.3. Hipervisores: tipo I y tipo II

El hipervisor es el software (o firmware) que se encarga de crear, gestionar y ejecutar las máquinas virtuales. Funciona como el “director de orquesta” que reparte los recursos entre los sistemas virtuales.

5.3.1. Tipo I – Bare Metal

Se ejecuta directamente sobre el hardware físico, sin sistema operativo intermedio. Se usa en entornos empresariales o servidores.

- Ejemplos: VMware ESXi, Microsoft Hyper-V Server, Xen.
- Muy eficiente, estable y seguro.

5.3.2. Tipo II – Hosted

Se instala sobre un sistema operativo existente. El hipervisor funciona como una aplicación más dentro del sistema anfitrión.

- Ejemplos: VirtualBox, VMware Workstation, Parallels Desktop.
- Ideal para entornos educativos y de pruebas.

> **Ejemplo:** En los equipos del aula, VirtualBox (tipo II) nos permitirá ejecutar Linux dentro de Windows sin alterar el equipo original.

5.4. Herramientas actuales: VirtualBox, VMware, Hyper-V

VirtualBox (Oracle)

- Gratuito, multiplataforma y muy usado en educación.
- Compatible con casi cualquier sistema operativo invitado.
- Permite “instantáneas”, carpetas compartidas, portapapeles bidireccional y clonación rápida.

> **Uso educativo típico:** Instalar una VM con Ubuntu o Windows Server para prácticas de red, permisos y configuración de usuarios.

VMware

- Disponible en varias versiones: **Workstation** (Windows/Linux) y **Fusion** (macOS).
- Mayor rendimiento que VirtualBox en entornos profesionales.
- Admite integración con plataformas de nube y servidores ESXi.

Hyper-V (Microsoft)

- Incluido en versiones profesionales de Windows.
- Ideal para virtualizar entornos de Windows Server.
- Permite crear conmutadores virtuales, checkpoints y configuraciones avanzadas de red.

> **Nota:** En DAW o entornos docentes, suele preferirse VirtualBox por su sencillez y porque permite combinar sistemas operativos distintos con facilidad.

5.5. Configuración de máquinas virtuales

Crear una máquina virtual implica decidir qué recursos del equipo físico se asignarán al entorno virtual.

Parámetros principales:

1. **Nombre y tipo de sistema operativo invitado.**
   Ejemplo: “Ubuntu Server 22.04 (Linux 64 bits)”.
2. **Memoria RAM asignada.**
   Recomendado: mínimo 2 GB para Linux, 4 GB para Windows.
3. **Procesadores virtuales.**
   Depende del número de núcleos físicos.
4. **Disco duro virtual.**
   - Tamaño: 20–40 GB según el SO.
   - Formato: VDI (VirtualBox), VMDK (VMware).
   - Modo: dinámico (crece según uso) o fijo.
5. **Adaptadores de red.**
   - NAT: acceso a Internet a través del host.
   - Puente: conexión directa a la red local.
   - Interna: solo comunicación entre VMs.
6. **Periféricos compartidos:** portapapeles, carpetas, USB, etc.

> **Analogía:** Crear una VM es como amueblar un piso: decides cuánta luz (RAM), cuántas habitaciones (CPU) y cuántos armarios (almacenamiento) tendrás.

5.6. Instalación básica de un sistema operativo paso a paso

Ejemplo práctico: instalación de **Ubuntu Desktop en VirtualBox**.

1. **Descargar la ISO** desde la web oficial.
2. **Crear una nueva VM** con el asistente de VirtualBox.
3. **Seleccionar la ISO** como unidad de arranque.
4. **Configurar idioma y teclado.**
5. **Particionar el disco virtual** (automático o manual).
6. **Crear usuario y contraseña.**
7. **Instalar actualizaciones y reiniciar.**
8. **Instalar “Guest Additions”** para mejorar gráficos e integración.

5.7. Proceso de arranque: BIOS/UEFI, MBR, GPT, gestor de arranque

Cada vez que encendemos un ordenador, comienza una secuencia precisa llamada proceso de arranque (*boot process*). En una máquina virtual, este proceso se emula igual que en un equipo físico.

Fases principales:

1. **POST (Power-On Self Test):** el firmware (BIOS o UEFI) comprueba memoria, CPU y periféricos.
2. **Localización del gestor de arranque:** busca en el disco el código que iniciará el sistema operativo.
3. **Carga del sistema operativo:** el gestor (GRUB, Windows Boot Manager, etc.) transfiere el control al kernel.

BIOS vs. UEFI

- **BIOS:** sistema clásico basado en texto, con particiones MBR (máx. 4 primarias).
- **UEFI:** interfaz moderna con soporte para particiones GPT (hasta 128 primarias), arranque más rápido y funciones de seguridad (Secure Boot).

> **Analogía:**
> BIOS es como un despertador analógico que enciende el sistema;
> UEFI es un smartphone que además te muestra la agenda del día.

MBR y GPT

- **MBR (Master Boot Record):** sector inicial del disco que guarda la tabla de particiones y el código de arranque.
- **GPT (GUID Partition Table):** reemplazo moderno con mayor capacidad y redundancia.

5.8. Integración entre host y máquina virtual

Una de las ventajas de la virtualización moderna es que el sistema anfitrión y las VMs pueden interactuar como si fueran compañeros de trabajo.

Ejemplos de integración:

- **Carpetas compartidas:** permiten intercambiar archivos sin red.
- **Portapapeles bidireccional:** copiar y pegar texto entre host y VM.
- **Puertos USB virtualizados:** conectar dispositivos físicos (pendrives, cámaras).
- **Acceso remoto:** control de la VM desde otro ordenador.

5.9. Copias, instantáneas y clonación

Instantáneas (Snapshots)

Permiten guardar el estado exacto de una VM en un momento concreto. Si algo falla, se puede volver atrás instantáneamente.

Ejemplo: antes de instalar una actualización o probar un software desconocido.

Clonación

Crea una copia completa de una VM para usarla como base en otros equipos o prácticas.

Ejemplo: el profesor prepara una VM con Ubuntu configurado y la distribuye al alumnado para ahorrar tiempo.

5.10. Ventajas y limitaciones en entornos educativos y profesionales

Ventajas	Limitaciones
Entornos seguros de experimentación.	Mayor consumo de recursos.
Portabilidad (mover VMs entre equipos).	Menor rendimiento que el hardware real.
Facilidad para documentar prácticas.	Configuraciones complejas de red avanzada.
Aislamiento frente a errores o malware.	Dependencia del hipervisor.

> **Ejemplo real:** En DAW, la virtualización permite a cada alumno tener su "laboratorio personal" sin alterar el sistema del aula. En empresas, facilita el despliegue de entornos de prueba, preproducción y producción.

5.11. Caso práctico: instalación documentada en VirtualBox

El alumnado debe instalar Ubuntu Server 22.04 en una VM de VirtualBox y documentar el proceso.

Tareas:

1. Crear la VM con 2 GB de RAM y 20 GB de disco.
2. Instalar el sistema y configurar usuario, red e idioma.
3. Tomar **capturas de pantalla** de cada fase.
4. Guardar una **instantánea** tras la instalación.
5. Exportar la VM a un archivo `.ova` para compartirla.

Evaluación:

- Claridad en los pasos.
- Comprensión de conceptos (RAM, disco, red).
- Capacidad para resolver incidencias.

Extensión opcional: Instalar una segunda VM con Windows y probar la comunicación entre ambas (ping, carpetas compartidas, servidor web).

5.12. Reflexión: la virtualización como laboratorio seguro para aprender

La virtualización ha transformado la forma de enseñar y aprender informática. Antes, un error podía dejar inservible un equipo. Hoy, se puede romper, restaurar y repetir cuantas veces sea necesario.

En definitiva, la virtualización no solo ahorra recursos: democratiza el aprendizaje. Permite experimentar sin miedo, explorar nuevos sistemas y entender la informática desde dentro, sin riesgo y con total libertad.

6. Mantenimiento, actualización y buenas prácticas

6.1. El ciclo de vida del software

Todo software, como cualquier producto, nace, crece, madura y muere. Ningún programa es eterno: necesita actualizaciones, correcciones y, tarde o temprano, sustitución.

Fases del ciclo de vida:

- Desarrollo y pruebas: se crea el producto, se corrigen errores iniciales.
- Lanzamiento: el software llega a los usuarios finales.
- Mantenimiento: se publican actualizaciones y parches.
- Madurez: se estabiliza; recibe menos cambios.
- Fin de soporte: deja de recibir actualizaciones.

> Ejemplo: Windows XP (2001-2014) pasó más de una década operativo, pero tras finalizar el soporte, usarlo hoy supone un riesgo de seguridad enorme.

6.2. Tipos de actualizaciones y parches

Las actualizaciones son modificaciones que mejoran el rendimiento o la seguridad de un software ya instalado. Se dividen según su propósito y alcance.

Tipo	Descripción	Ejemplo
Parches de seguridad	Corrigen vulnerabilidades críticas.	“Patch Tuesday” de Microsoft.
Actualizaciones menores	Mejoran estabilidad o corrigen errores (bug fixes).	De la versión 1.3.0 a 1.3.1.
Actualizaciones mayores	Incorporan nuevas funciones o rediseños completos.	De Windows 10 a Windows 11.
Service Packs	Conjunto acumulativo de actualizaciones.	Windows 7 SP1.
Actualizaciones automáticas	Se instalan sin intervención del usuario.	Ubuntu o Chrome.

6.3. Desinstalación y limpieza del sistema

Tan importante como instalar software es mantener limpio el sistema. Los programas generan archivos temporales, cachés y claves de registro que ocupan espacio o ralentizan el equipo.

Buenas prácticas de limpieza:

- Desinstalar aplicaciones no utilizadas.
- Vaciar carpetas temporales (%TEMP%, /tmp).
- Usar herramientas de limpieza seguras (BleachBit, CCleaner con precaución).
- Revisar los programas que se ejecutan al inicio (Administrador de tareas → “Inicio”).
- Eliminar controladores antiguos.
- Desfragmentar (en discos HDD) o ejecutar TRIM (en SSD).

6.4. Copias de seguridad y restauración

El respaldo de datos es el salvavidas de cualquier técnico.

No se trata de si fallará el sistema, sino de cuándo fallará.

Tipos de copia de seguridad:

Tipo	Descripción	Ejemplo
Completa	Copia todo el contenido.	Imagen del sistema.
Incremental	Copia solo lo que cambió desde la última copia.	Respaldos diarios.
Diferencial	Copia lo modificado desde la última copia completa.	Respaldos semanales.
En la nube	Se guarda en servidores remotos.	Google Drive, OneDrive, AWS S3.
Local	En discos externos o NAS.	Backup semanal en disco USB.

Herramientas comunes:

- Windows Backup & Restore, Time Machine (macOS), Déjà Dup (Linux).
- Copias automatizadas mediante scripts o cron jobs.
- Snapshots en máquinas virtuales.

> **Caso práctico:** Antes de reinstalar un sistema, el alumnado debe crear una copia de seguridad de sus documentos y exportar configuraciones o snapshots de las VMs.

6.5. Monitorización y optimización del rendimiento

El rendimiento de un sistema depende de múltiples factores: hardware, procesos en ejecución, espacio libre o configuración de red.

Herramientas básicas:

- Administrador de tareas / Monitor de recursos (Windows).
- top, htop, iotop, free, df (Linux).
- Activity Monitor (macOS).

Estas herramientas permiten observar:

- Uso de CPU y RAM.
- Procesos activos.
- Actividad del disco y red.
- Temperatura y carga del sistema.

Prácticas de optimización:

- Desactivar servicios innecesarios.
- Limitar aplicaciones de inicio.
- Mantener controladores actualizados.
- Limpiar archivos temporales y logs antiguos.
- Optimizar disco y memoria virtual.
- Usar software ligero en equipos antiguos (por ejemplo, Lubuntu en lugar de Ubuntu).

> **Ejemplo real:** En un equipo con 4 GB de RAM, sustituir Chrome por Firefox puede reducir el consumo un 20 %. A escala de aula o empresa, ese pequeño cambio se traduce en un gran ahorro energético.

6.6. Buenas prácticas en entornos profesionales

Un técnico DAW no solo instala software: garantiza su fiabilidad. Y eso requiere metodología.

Principios básicos:

1. **Documentar siempre los cambios.**
   Cada actualización, reinstalación o configuración debe quedar registrada.
2. **Usar versiones estables y compatibles.**
   Evitar betas o software sin soporte.
3. **Aplicar la regla 3-2-1 de copias de seguridad:**
   - 3 copias,
   - en 2 soportes distintos,
   - y 1 en una ubicación remota.
4. **Seguridad ante todo:**

    - Actualizaciones firmadas digitalmente.
    - Antivirus y firewall activos.
    - Contraseñas seguras y gestión centralizada.
5. **Automatizar tareas repetitivas:**
   Scripts de mantenimiento, limpieza o monitorización.

> **Caso real:** En una empresa web, el equipo de sistemas programa copias automáticas nocturnas, monitoriza los servidores con Zabbix y mantiene un registro de intervenciones en GitLab Wiki. Así, cualquier técnico puede recuperar el historial si ocurre un fallo.

6.7. Documentación técnica y control de versiones de sistemas

La documentación técnica es el esqueleto invisible que mantiene ordenado todo el trabajo de un equipo. Un sistema bien documentado ahorra tiempo, errores y conflictos.

Qué debe incluir:

- Especificaciones de hardware y software.
- Versiones instaladas.
- Cambios realizados (logs de mantenimiento).
- Usuarios y permisos.
- Plan de copias y recuperación.

Hoy en día, incluso la infraestructura puede versionarse: con herramientas como Git, Ansible o Terraform, los técnicos guardan la configuración de servidores igual que los desarrolladores guardan su código.

6.8. Reflexión final: el software como sistema vivo

Un sistema operativo no es un producto estático; es un organismo en evolución. Respira, cambia y se adapta a su entorno. Los parches son vacunas; las actualizaciones, alimento; las copias, su memoria.

> **Un técnico no mantiene ordenadores: mantiene ecosistemas digitales.**
> **Cada actualización que aplicas, cada error que corriges, cada copia que verificas, forma parte de la salud de ese ecosistema.**

7. Cierre de unidad

7.1. Síntesis global

A lo largo de esta unidad hemos recorrido todo el ecosistema del software, desde los fundamentos hasta las prácticas profesionales:

Bloque	Contenido esencial	Competencia desarrollada
1. Concepto y clasificación del software	Diferencia entre software de sistema y de aplicación.	Identificación de componentes del sistema informático.
2. El sistema operativo	Funciones, estructura, kernel, mecanismos de comunicación y logs.	Comprensión del papel del SO como gestor de recursos.
3. Software de aplicación y desarrollo	Tipos de software, IDEs, frameworks, Git, DevOps.	Entorno profesional del desarrollador DAW.
4. Licencias y modelos de distribución	Propiedad intelectual, licencias libres y comerciales, ética profesional.	Uso legal y responsable del software.
5. Virtualización e instalación	Creación de máquinas virtuales, hipervisores, proceso de arranque.	Configuración y despliegue seguro de entornos.
6. Mantenimiento y buenas prácticas	Ciclo de vida, copias de seguridad, monitorización y documentación.	Administración y sostenibilidad de sistemas.

En conjunto, esta unidad te convierte en un usuario consciente, un técnico competente y un desarrollador responsable.

7.2. Glosario técnico

Software: Conjunto de programas que hacen funcionar el hardware.

Kernel: Núcleo del sistema operativo que gestiona los recursos.

Interrupción: Señal que pausa un proceso para atender otro prioritario.

System Call: Puerta controlada que permite a un programa acceder al SO.

Framework: Estructura base para desarrollar aplicaciones.

Librería: Conjunto de funciones reutilizables.

API: Interfaz que permite que diferentes programas se comuniquen.

Git: Sistema de control de versiones distribuido.

Virtualización: Técnica que permite ejecutar varios sistemas en un mismo hardware.

Hipervisor: Software que gestiona las máquinas virtuales.

Snapshot: Imagen congelada del estado de una VM.

Parche: Actualización que corrige errores o vulnerabilidades.

Backup: Copia de seguridad de datos o sistemas.

DevOps: Cultura de integración entre desarrollo y operaciones.

Licencia: Acuerdo legal que regula el uso del software.

7.3. Cuadro comparativo de tipos de software

Criterio	Software de sistema	Software de aplicación	Software de desarrollo
Finalidad	Gestionar recursos del hardware y ofrecer servicios básicos.	Resolver tareas del usuario final.	Crear y mantener otros programas.
Ejemplos	Windows, Linux, macOS, Android.	Word, Photoshop, Chrome.	Visual Studio Code, Git, Docker.
Relación con el usuario	Indirecta. El usuario no lo usa directamente.	Directa: interfaz gráfica o aplicación.	Técnica: programadores y administradores.
Instalación	Generalmente preinstalado con el hardware.	Por descarga o instalación manual.	Por elección del desarrollador.
Dependencia	Necesario para ejecutar cualquier otro software.	Depende del sistema operativo.	Depende del lenguaje y entorno de desarrollo.
Ejecutado por	El propio sistema operativo.	El usuario.	El desarrollador o técnico.

7.4. Actividades de repaso y autoevaluación

A. Comprensión teórica

- Explica la diferencia entre software libre y software open source.
- Indica tres ventajas de la virtualización en entornos educativos.
- ¿Qué es una system call y para qué sirve?
- Describe la diferencia entre un hipervisor tipo I y tipo II.
- ¿Por qué se recomienda documentar las actualizaciones del sistema?

B. Ejercicios prácticos

- Crea una máquina virtual con Linux en VirtualBox, instala un navegador y haz una captura de la configuración de red.
- Investiga qué licencia usa VS Code y qué implicaciones tiene.

- Simula con un compañero un flujo de trabajo en Git: `clone`, `commit`, `push`, `pull`.
- Realiza una copia de seguridad de tus documentos en la nube y genera un informe breve del proceso.
- Analiza las actualizaciones automáticas de tu sistema: ¿qué ventajas e inconvenientes encuentras?

7.5. Casos de aplicación profesional: instalación de un entorno DAW completo

Eres técnico de un centro educativo que debe preparar **un laboratorio de desarrollo web completo** para un grupo de 1º de DAW. Cada equipo debe tener todo lo necesario para trabajar en proyectos web modernos.

Tareas a realizar

1. Instalar una **máquina virtual** con Ubuntu 22.04.
2. Configurar el entorno de desarrollo:
    - **VS Code**, **Git**, **Node.js**, **Apache** o **Nginx**.
3. Verificar conectividad a Internet y acceso a carpetas compartidas.
4. Crear un **usuario alumno** con permisos restringidos.
5. Documentar todo el proceso (capturas + explicación).
6. Realizar una **instantánea** al finalizar.

Criterios de evaluación

- Instalación funcional y sin errores.
- Uso correcto de recursos (RAM, disco, red).
- Documentación técnica clara y estructurada.
- Aplicación de buenas prácticas (nombres, rutas, permisos).
- Evidencia de comprensión del proceso (no solo ejecución mecánica).