Organización de los sistemas operativos

4. Componentes del sistema

Crear un software tan complejo como un sistema operativo no es sencillo, por ello resulta más práctico dividirlo en piezas más pequeñas especializadas en aspectos concretos de la gestión del sistema.

Gestión de procesos

- El **proceso** es la unidad de trabajo. Es quién realiza las tareas que interesan a los usuarios. Cada proceso es asignado un tiempo de CPU y el resto de recursos del sistema.
- Un **proceso** es un programa en ejecución. Un programa se convierte en proceso cuando las instrucciones del programa son cargadas en la memoria desde el archivo del ejecutable y se le asignan recursos para su ejecución.

Los procesos son entidades activas que requieren recursos como CPU, memoria, etc. Algunos recursos se asignan al crear el proceso y otros se solicitan durante su ejecución, como la memoria, que puede aumentar dinámicamente. Al finalizar, el sistema operativo recupera los recursos reutilizables para otros procesos.

Un **programa** es una entidad pasiva; es solo un archivo en disco con instrucciones. No realiza trabajo hasta que sus instrucciones son ejecutadas por la CPU, momento en el cual se convierte en un proceso.

Aunque varios procesos pueden estar basados en el mismo programa, no son el mismo proceso. Cada proceso tiene su propio contador de programa y estado en los registros de la CPU, lo que determina qué instrucciones ejecuta y cómo evoluciona. Aunque varios procesos ejecuten el mismo programa, su secuencia de instrucciones y su estado serán diferentes, por lo que deben considerarse procesos distintos.

Responsabilidades de la gestión de procesos

El componente de gestión de procesos es el responsable de las siguientes actividades:

- Crear y terminar procesos.
- Suspender y reanudar los procesos.
- Proporcionar mecanismos para la sincronización de procesos.
- Proporcionar mecanismos para la comunicación entre procesos.
- Proporcionar mecanismos para el tratamiento de interbloqueos.

Gestión de la memoria principal

La memoria principal es crucial para las operaciones de un sistema operativo moderno, ya que es el único almacenamiento al que la CPU accede directamente. Para que un programa sea ejecutado, primero debe ser copiado a la memoria principal. Además, si un proceso necesita acceder a datos en otro dispositivo de almacenamiento, esos datos deben ser copiados previamente a la memoria principal.

Para mejorar el uso de la CPU y la respuesta del usuario, es necesario tener varios programas en la memoria simultáneamente. Como estos programas comparten la memoria

durante su ejecución, el sistema operativo requiere un componente de gestión de la memoria para organizar y controlar este uso compartido.

Responsabilidad de la gestión de la memoria

El componente de gestión de la memoria debe asumir las siguientes responsabilidades:

- Controlar qué partes de la memoria están actualmente en uso y cuáles no.
- Decidir que procesos añadir o extraer de la memoria cuando hay o falta espacio en la misma.
- Asignar y liberar espacio de la memoria principal según sea necesario.

Gestión del sistema de E/S

El **sistema de E/S** hace de interfaz con el hardware, oculta las peculiaridades del hardware al resto del sistema.

El sistema de E/S consta de:

- **Componente de gestión de memoria especializado en E/S**: Ofrece servicios como buffering, caching y spooling.
- **Interfaz genérica para acceso a los dispositivos**: El sistema proporciona una única interfaz para acceder a ellos, sin que los procesos o componentes del sistema necesiten conocer sus particularidades. Esto permite que los programadores accedan a diferentes dispositivos, sin preocuparse por las características específicas del hardware.
- **Controladores de dispositivo**: Estos controladores conocen las particularidades de cada dispositivo. Las peticiones hechas a la interfaz de E/S genérica son enviadas a los controladores, que las traducen en acciones concretas sobre el hardware del dispositivo.

Buffering

El **buffering** o uso de memoria intermedia es una estrategia en la que los datos se almacenan temporalmente en un búfer, una zona de memoria. El controlador del dispositivo escribe los bloques de datos solicitados en el búfer Una vez lleno, el contenido del búfer se transfiere al proceso que realizó la solicitud para que lo procese. Mientras el proceso trabaja con esos datos, el controlador sigue almacenando nuevos datos en el búfer.

Caching

El **caching** es una técnica en la que el sistema almacena en la memoria principal una copia de los datos que han sido leídos o escritos recientemente en los dispositivos de E/S, como discos duros o memorias USB. Esto mejora la eficiencia, ya que acceder a la memoria principal es mucho más rápido que hacerlo a los dispositivos de E/S. Si el sistema accede frecuentemente a los mismos datos, puede recuperarlos directamente de la memoria principal, acelerando el proceso. Sin embargo, debido a espacio limitado en la memoria principal, solo se almacena la copia de los datos que se utilizan con mayor frecuencia.

Spooling

El **spooling** se emplea en dispositivos que no permiten el acceso simultáneo de varias aplicaciones. Cuando varias aplicaciones envían trabajos a una impresora, el sistema operativo intercepta los datos y los copia en archivos independientes. una vez que la aplicación termina de enviar su trabajo, el archivo se coloca en una cola. Los

trabajos en la cola se procesan uno a uno, evitando el acceso simultáneo al dispositivo. Esto permite que los procesos entreguen sus trabajos y continúen con otras tareas sin tener que esperar a que la impresora esté disponible.

Gestión del almacenamiento secundario

Los dispositivos de E/S dedicados al **almacenamiento secundario** requieren un tratamiento especial. Aunque los programas que se ejecutan deben estar en la **memoria principal**, esta es limitado y no puede alojar todos los datos y programas del sistema. Además, la memoria principal es volátil, por lo que los datos se perderían si ocurre un fallo de alimentación.

Por ello, los ordenadores disponen de **almacenamiento secundario**, que permite almacenar grandes volúmenes de datos de manera permanente. El **gestor de almacenamiento secundario** utiliza el sistema de E/S para acceder a estos dispositivos y ofrecer servicios más complejos al sistema operativo, garantizando el manejo adecuado de los datos almacenados.

Responsabilidades de la gestión del almacenamiento secundario

El gestor del almacenamiento secundario es el responsable de:

- Gestionar el espacio libre en discos duros y resto de dispositivos de almacenamiento secundario.
- Asignar el espacio de almacenamiento.
- Planificar el acceso a los dispositivos, de tal forma que se ordenen las operaciones de forma eficiente.

Gestión del sistema de archivos

Los ordenadores almacenan información en diversos medios físicos, cada uno con características propias. El acceso a estos medios se controla a través de dispositivos específicos, como el controlador de disco o la unidad de DVD-ROM, también con sus particularidades. Aunque el sistema de E/S y la gestión del almacenamiento secundario facilitan el acceso a estos dispositivos, aún no es suficientemente cómodo para los programas usarlos directamente de forma constante.

Para simplificar este acceso, el sistema operativo proporciona una **visión lógica uniforme** del almacenamiento, abstracta de las características físicas de los dispositivos. Esto se logra mediante la definición de un **archivo**, que es una unidad lógica con la que los procesos trabajan para guardar y recuperar datos. Un archivo es una colección de datos relacionados, identificados por un nombre, y tratados como una unidad de información en el almacenamiento secundario.

Para el sistema operativo, un archivo es generalmente solo una **colección de bytes**, y ofrece servicios para leer, escribir, identificar y manipular esa colección. El formato o la organización de la información en el archivo, como la codificación de una imagen en formato JPEG, es responsabilidad de las aplicaciones, no del sistema operativo.

Además, los archivos se organizan en **directorios** para facilitar su uso y administración.

Responsabilidades de la gestión del sistema de archivos

El sistema de archivos utiliza al gestor del almacenamiento secundario y al sistema de E/S y es responsable de las siguientes actividades:

- Crear y borrar archivos.
- Crear y borrar directorios para organizar los archivos.
- Soportar operaciones básicas para la manipulación de archivos y directorios: lectura y escritura de datos, cambio de nombre, cambio de permisos, etc.
- Mapear en memoria archivos del almacenamiento secundario.
- Hacer copias de seguridad de los archivos en sistemas de almacenamiento estables y seguros.

Gestión de red

El componente de red se responsabiliza de la comunicación con otros sistemas interconectados mediante una red de ordenadores.

Protección y seguridad

La **protección** en un sistema informático es un mecanismo que controla el acceso de los procesos y usuarios a los recursos del sistema. Es fundamental en sistemas con múltiples usuarios y ejecución concurrente de procesos, ya que garantiza que solo los procesos autorizados puedan utilizar los recursos.

La protección mejora la **fiabilidad** del sistema al detectar elementos que no funcionan correctamente. Un recurso sin protección es vulnerable al uso indebido por parte de usuarios no autorizados o incompetentes.

Ejemplos de mecanismos de protección incluyen:

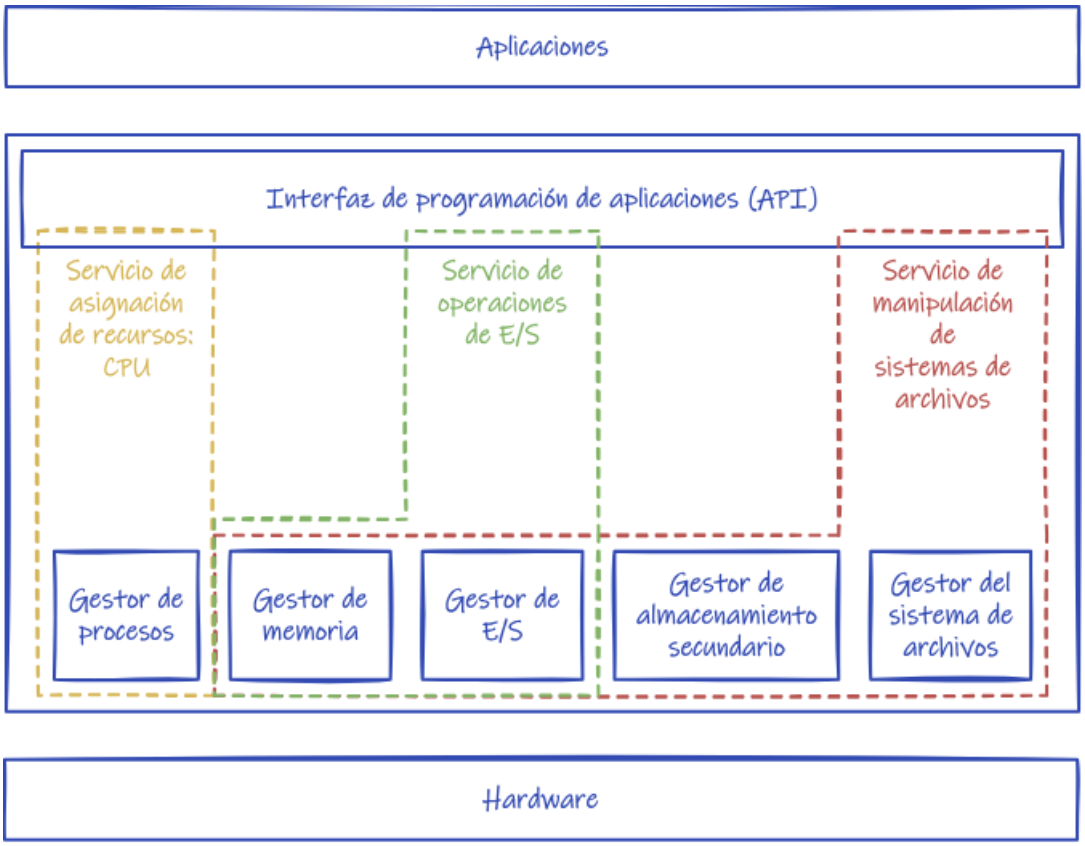
- **Hardware de direccionamiento de memoria**, que asegura que cada proceso se ejecute en su propio espacio de direcciones.
- **Temporizadores**, que impiden que un proceso monopolice la CPU indefinidamente.
- Protección de **registros de dispositivos de E/S**, que evita que los usuarios accedan directamente a estos dispositivos, preservando su integridad.
- **Permisos sobre archivos**, que limitan el acceso solo a procesos autorizados.

Sin embargo, aunque un sistema tenga mecanismos de protección, puede seguir siendo vulnerable a fallos y accesos no autorizados. Por ello, se necesitan **mecanismos de seguridad** que defiendan el sistema frente a ataques internos y externos, como virus, gusanos, ataques de denegación de servicio, robo de identidad y uso no autorizado del sistema.

---

5. Servicios del sistema

Un sistema operativo proporciona un entorno para la ejecución de programas. Ese entorno debe proporcionar ciertos servicios a los programas y a los usuarios de esos programas. Estos servicios son proporcionados gracias al funcionamiento coordinado de los diferentes componentes del sistema.



Aunque cada sistema operativo proporciona servicios diferentes, es posible identificar unas pocas clases comunes.

Servicios que garantizan el funcionamiento eficiente del sistema

La **asignación de recursos** es crucial cuando hay múltiples usuarios o trabajos ejecutándose simultáneamente, ya que los recursos del sistema deben ser distribuidos de manera eficiente. Ejemplos de recursos incluyen:

- **CPU**, asignada por el planificador del gestor de procesos.
- **Memoria principal**, administrada por el gestor de memoria.
- **Almacenamiento de archivos**, controlado por el sistema de archivos y el gestor de almacenamiento secundario.

El objetivo de esta asignación es garantizar el máximo rendimiento del sistema.

La **monitorización** permite hacer un seguimiento de qué recursos utilizan los usuarios y en qué cantidad. Esto es útil para diversos fines, como:

- **Facturación** por el uso de recursos (por ejemplo, tiempo de CPU).
- **Optimización del rendimiento** del sistema.
- **Limitación** de los recursos que cada usuario puede consumir.

En cuanto a **protección y seguridad**, la protección asegura que el acceso a los recursos esté controlado, impidiendo que un usuario acceda a información de otro o que un proceso interfiera con otros. La **seguridad** incluye la defensa frente a amenazas externas, lo que implica obligar a los usuarios a autenticarse antes de acceder a los recursos y proteger el sistema de intentos no autorizados, especialmente a través de la red.

Servicios útiles para el usuario

Los sistemas operativos deben ofrecer varias funcionalidades clave para facilitar la interacción con los usuarios y gestionar los recursos de manera eficiente:

1. **Interfaz de usuario**: Los sistemas operativos que permiten la interacción directa con el usuario deben proporcionar una interfaz adecuada. Esta interfaz puede variar dependiendo del propósito del sistema (por ejemplo, una interfaz gráfica en sistemas de escritorio o una interfaz de línea de comandos en servidores).
2. **Operaciones de E/S**: Los programas a menudo necesitan realizar operaciones de entrada y salida (E/S), ya sea con archivos o dispositivos. Por motivos de eficiencia y protección, los usuarios y procesos no suelen tener acceso directo a los dispositivos. El sistema operativo proporciona los medios para que los programas soliciten estas operaciones a los componentes correspondientes del sistema.
3. **Manipulación de sistemas de archivos**: Los programas requieren interactuar con archivos y directorios, lo que incluye la capacidad de leer, escribir, crear, borrar, buscar archivos y listar información sobre ellos.
4. **Comunicaciones**: Los procesos necesitan intercambiar información entre ellos. Esto puede ocurrir tanto en el mismo ordenador como entre diferentes equipos conectados a una red.
5. **Detección de errores**: El sistema operativo debe ser capaz de detectar errores en varias áreas y tomar medidas para mantener una computación segura y consistente. Estos errores pueden incluir fallos de hardware (como fallos de energía o de memoria), errores en la E/S (como falta de papel en la impresora) y errores en los programas de usuario (como desbordamientos aritméticos o accesos no permitidos a la memoria).

Interfaz de usuario

La **interfaz de usuario** es un servicio clave en los sistemas operativos diseñados para que los usuarios interactúen directamente con ellos. Existen varios tipos de interfaces de usuario:

1. **Interfaz de línea de comandos (CLI)**: Permite a los usuarios introducir comandos directamente para que el sistema operativo los ejecute. A veces está integrada en el núcleo del sistema, pero normalmente es un programa separado llamado **shell** (como en MS-DOS o UNIX) que se ejecuta cuando el usuario inicia sesión.
2. **Interfaz de proceso por lotes**: Los comandos y directivas se organizan en archivos que se ejecutan posteriormente. Este tipo de interfaz es común en sistemas no interactivos, como los sistemas de procesamiento por lotes antiguos y sistemas multiprogramados. También se utiliza en sistemas modernos de tiempo compartido y de escritorio, donde el **shell** permite ejecutar comandos de manera interactiva o mediante **scripts** que ejecutan una lista de órdenes automáticamente.
3. **Interfaz gráfica de usuario (GUI)**: Proporciona una experiencia más visual, con ventanas y menús controlados mediante el ratón, como los que se encuentran en sistemas operativos modernos.

Aunque la interfaz de usuario puede variar entre sistemas y usuarios, no se considera un componente esencial del sistema operativo, sino un servicio que este ofrece.

Además de la interfaz de usuario, los sistemas operativos modernos incluyen una **colección de programas del sistema**, cuyo propósito es facilitar el entorno para ejecutar y desarrollar programas. Estos programas del sistema incluyen:

- Herramientas para manipular archivos y directorios.

- Programas para obtener información del sistema (como fecha, hora, memoria, espacio en disco).
- Herramientas de desarrollo (intérpretes, compiladores, enlazadores, depuradores).
- Programas de comunicaciones (clientes de correo electrónico, navegadores web).

Asimismo, muchos sistemas operativos incluyen **utilidades del sistema o programas de aplicación**, diseñados para resolver necesidades comunes de los usuarios, como:

- Editores y procesadores de texto.
- Hojas de cálculo.
- Sistemas de bases de datos.
- Juegos.

Estas aplicaciones complementan el sistema operativo ofreciendo soluciones prácticas para el usuario.

---

6. Interfaz de programación de aplicaciones

Un sistema operativo proporciona un entorno controlado para la ejecución de programas. Dicho entorno debe proporcionar ciertos servicios que pueden ser accedidos por los programas a través de una interfaz de programación de aplicaciones o API (Application Programming Interface).

Interfaces de programación de aplicaciones

Algunas de las API disponibles para los desarrolladores de aplicaciones son Windows API y POSIX.

Windows API

La **Windows API** es la interfaz de programación de aplicaciones de Microsoft Windows con la que casi todas las aplicaciones deben interactuar de alguna manera. Anteriormente conocida como **Win32 API**, el nombre actual abarca todas las versiones de la API, como **Win16** (usada en versiones de 16 bits) y **Win64** (la variante adaptada para arquitecturas de 64 bits).

La Windows API está compuesta por funciones escritas en C, principalmente almacenadas en **librerías de enlace dinámico (DLL)** como:

- **kernel32.dll**: para operaciones del núcleo del sistema.
- **user32.dll**: para gestión de ventanas y entrada de usuario.
- **gdi32.dll**: para gráficos y dispositivos de salida.

Con el tiempo, se han añadido otras librerías adicionales para ampliar la funcionalidad de la API.

La Windows API ofrece un **amplio conjunto de servicios**, que incluyen:

- E/S a archivos y dispositivos.
- Gestión de procesos, hilos y memoria.
- Manejo de errores.
- Registro de Windows.
- Interfaz para dispositivos gráficos (pantallas, impresoras).
- Gestión de ventanas.

- Comunicaciones en red.

Estos servicios permiten a las aplicaciones interactuar con los distintos componentes del sistema operativo y el hardware.

POSIX

POSIX (Portable Operating System Interface for Unix) es un conjunto de estándares que define una interfaz de programación de aplicaciones (API) para sistemas operativos, permitiendo que un programa pueda ejecutarse en diferentes sistemas compatibles con POSIX.

El lenguaje **C**, originalmente diseñado para implementar sistemas UNIX, tenía una biblioteca estándar similar a la del sistema UNIX. Sin embargo, con el tiempo, las bibliotecas de sistemas UNIX de diferentes fabricantes divergen, lo que hace difícil desarrollar programas portables entre ellos. Para resolver este problema, el **IEEE** creó el estándar **POSIX**, que define una API común para sistemas UNIX y similares, como **GNU/Linux**, lo que hace que la mayoría de estos sistemas sean compatibles con POSIX.

La API POSIX es un **superconjunto** de la biblioteca estándar de C. En sistemas POSIX, la biblioteca estándar de C es parte de la biblioteca del sistema, y muchas de las funciones POSIX se encuentran en la **libc**, con algunas funciones específicas en otras bibliotecas, como **libm** (matemáticas) o **libpthread** (hilos).

A veces, los desarrolladores añaden funciones adicionales no incluidas en POSIX, como sucede con BSD o la librería del sistema GNU. Además, el estándar POSIX ha tenido varias revisiones (la primera en 1988), cada una añadiendo nuevas características.

Consideraciones al usar extensiones avanzadas:

- Un programa que solo use funciones de una versión de POSIX puede ejecutarse en cualquier sistema compatible con esa versión.
- Si se usan funciones no estándar, como las de GNU/Linux o macOS, el programa solo funcionará en esos sistemas.

Para manejar estas diferencias, los sistemas POSIX ofrecen **macros de test de características** que permiten controlar qué funcionalidades del sistema están disponibles. Algunas de las macros más comunes incluyen:

- **_POSIX_C_SOURCE**: Activa definiciones de POSIX según el valor asignado (por ejemplo, 200809L activa POSIX.1-2008).
- **_XOPEN_SOURCE**: Activa la especificación de portabilidad X/Open junto con ciertas extensiones POSIX.
- **_BSD_SOURCE**: Activa funcionalidades específicas de los sistemas BSD.
- **_DEFAULT_SOURCE**: Activa las especificaciones por defecto, como POSIX.1-2008 e ISO C99, junto con algunas extensiones adicionales.
- **_GNU_SOURCE**: Activa _DEFAULT_SOURCE y extensiones específicas de los sistemas GNU.

Ejemplo de uso:

En el programa de ejemplo en C:

- La macro **_POSIX_C_SOURCE 200809L** activa funciones de la especificación POSIX.1-2008, lo que permite utilizar funciones como **mkstemp()**. Esta macro asegura que el

programa solo usará funciones compatibles con esa versión de POSIX, evitando errores de compilación en sistemas que no soporten versiones más recientes.

En resumen, POSIX estandariza la API para asegurar la portabilidad de programas entre sistemas UNIX y similares, y el uso de macros de test de características permite a los desarrolladores controlar las especificaciones que su código utilizará, garantizando compatibilidad y portabilidad.

Llamadas al sistema

Para que un programa acceda a los servicios del sistema operativo, no basta con invocar una función de manera directa. Para invocar una función, un programa debe conocer la **dirección de memoria** de su punto de entrada, es decir, dónde se encuentra la primera instrucción de esa función. Sin embargo, el **código del núcleo del sistema** (kernel) puede estar en cualquier parte de la memoria principal y, además, generalmente está **protegido contra accesos directos**, lo que hace que las direcciones de las funciones del núcleo no sean accesibles o conocidas por los procesos de usuario.

Para superar esta limitación, se utiliza un mecanismo especial llamado **llamada al sistema**. Este procedimiento permite que un proceso solicite servicios al sistema operativo, ya que los procesos de usuario no pueden acceder directamente al código del núcleo debido a las restricciones de seguridad y protección de la memoria. Las llamadas al sistema son intermediarios controlados que permiten a los programas usar los servicios del sistema operativo de manera segura y estructurada.

Invocar llamadas al sistema

Una **llamada al sistema** generalmente se invoca mediante una instrucción en lenguaje ensamblador que genera una **excepción**. Esta excepción es una interrupción que la CPU lanza al detectar instrucciones especiales o al ocurrir un error, como una división por cero o un acceso indebido a la memoria. Por ejemplo, en arquitecturas como **MIPS** o **Intel x86**, se usa la instrucción **syscall**, que provoca la excepción. Esta excepción hace que la CPU salte a una **rutina en el núcleo del sistema operativo**, deteniendo la ejecución del proceso que hizo la llamada.

Cuando se realiza una llamada al sistema, es necesario que el sistema operativo identifique la **operación específica** que está solicitando el proceso. Esto se hace asignando un **número identificativo** de la llamada a un registro específico de la CPU. Por ejemplo, en **Linux para x86**, la llamada al sistema **open()** (para abrir archivos) se identifica con el número **2** en sistemas de 64 bits o con el número **5** en sistemas de 32 bits. Este número se coloca en el registro **v0** en MIPS o **eax** en x86 antes de la instrucción **syscall**.

Los números que identifican cada llamada al sistema varían según el sistema operativo, mientras que el registro donde se almacena el número y otros detalles como la instrucción usada dependen de la **arquitectura de la CPU**. Este mecanismo garantiza que la comunicación entre un programa y el sistema operativo sea eficiente y controlada.

Paso de argumentos

Una **llamada al sistema** requiere más información que solo su identificador, como por ejemplo, los parámetros adicionales necesarios para completar la operación. En el caso de abrir un archivo, se debe proporcionar su nombre y el modo de apertura (lectura o escritura). Existen tres métodos principales para pasar estos parámetros adicionales:

1. Mediante registros de la CPU:

- Los parámetros se cargan en los registros de la CPU antes de invocar la llamada al sistema.
- Es el método más eficiente, pero está limitado por la cantidad de registros disponibles.
- Utilizado en sistemas como **Linux para MIPS** y la mayoría de sistemas operativos en **x86-64**.

Ejemplo en Linux MIPS para la syscall `write()`:

```
lw      $a0, FILEDES   # Cargar el descriptor de archivo en el registro a0
la      $a1, BUFFER    # Cargar la dirección del buffer en el registro a1
lw      $a2, SIZE      # Cargar el tamaño en el registro a2
li      $v0, 40004     # Cargar el identificador de la llamada write() en el registro v0
syscall                # Invocar la llamada al sistema
```

Al terminar, el número de bytes escritos se devuelve en el registro **v0**.

2. Mediante tabla en memoria:

- Los parámetros se almacenan en una tabla en la memoria principal, y la dirección de dicha tabla se coloca en un registro específico.
- Este método permite pasar un número ilimitado de parámetros.
- Fue utilizado en sistemas como **Microsoft Windows 2000** y en **Linux para x86 de 32 bits** cuando se exceden 6 parámetros.

3. Mediante la pila del proceso:

- Los parámetros se colocan en la pila del proceso, de manera similar a como se manejan los argumentos al llamar a funciones.
- Al igual que el método de tabla en memoria, no está limitado en cuanto al número de parámetros.
- Es utilizado en sistemas **UNIX BSD** y en **Windows XP y posteriores** para sistemas **x86 de 32 bits**.

Ejemplo de invocación de la llamada `write()` en Linux MIPS:

- Para escribir datos en un archivo en **Linux MIPS**, los parámetros que necesita son:
  - **FILEDES**: Descriptor del archivo donde se escribirán los datos.
  - **BUFFER**: Dirección en memoria del buffer que contiene los datos.
  - **SIZE**: Número de bytes a escribir.
- El identificador de la llamada `write()` es **4004** en Linux para MIPS, que se carga en el registro **v0** antes de ejecutar la instrucción `syscall`.

El sistema operativo, independientemente del método utilizado, es responsable de verificar la **validez de los parámetros** antes de ejecutar la operación solicitada. Esto es esencial para garantizar que no se produzcan errores o abusos en el uso de los recursos del sistema. Por lo tanto, el sistema no confía en que los procesos gestionen correctamente sus propios parámetros; en su lugar, realiza una validación estricta para mantener la estabilidad y seguridad del sistema.

Este procedimiento asegura que, aunque la llamada al sistema parezca una simple instrucción para el programa, detrás de escena el sistema operativo asume el control

y ejecuta las tareas correspondientes, verificando la integridad y validez de los datos proporcionados.

Librería del sistema

Las **llamadas al sistema** permiten que los procesos invoquen los servicios ofrecidos por el sistema operativo, pero debido a que se realizan mediante instrucciones en lenguaje ensamblador (como vimos en el **Ejemplo 6.1**), su uso directo no es muy cómodo. Por ello, los programas no suelen invocar las llamadas al sistema directamente, sino que lo hacen a través de funciones de la **librería del sistema**.

Características de la librería del sistema:

- **Parte del sistema operativo**: Se distribuye junto con el sistema operativo y está integrada en él.
- **Interfaz para los programas**: Es una colección de clases o funciones que proporcionan acceso a los servicios del sistema operativo a los programas, utilizando internamente las llamadas al sistema.

Funciones de la librería:

- Algunas funciones de la librería son **traducciones directas** de las llamadas al sistema, como las funciones **write()** o **close()** en POSIX, que invocan directamente la correspondiente llamada al sistema.
- Otras funciones de la librería pueden ser **más complejas** y abstraer detalles de bajo nivel, proporcionando una interfaz más sencilla o con mayor funcionalidad que la llamada al sistema base.

Interfaz de programación:

- La librería del sistema constituye la **interfaz de programación de aplicaciones (API)** del sistema operativo. Es la forma preferida y recomendada para que los programas soliciten servicios al sistema operativo.
- Invocar directamente las llamadas al sistema se considera el **último recurso** y no es la forma habitual de interactuar con el sistema.

Uso y funcionamiento:

- Las funciones de la librería del sistema se **invocan como cualquier otra función** dentro del programa, ya que se cargan en la región de memoria asignada al proceso.
- La mayoría de estas librerías están implementadas en **C**, lo que permite su uso directo tanto por programas en **C** como en **C++**.

En resumen, aunque las llamadas al sistema son fundamentales para la interacción entre los programas y el sistema operativo, los desarrolladores generalmente interactúan con el sistema operativo a través de las funciones de la librería del sistema, que facilitan y optimizan el uso de esos servicios.

Librería estándar

Diferentes lenguajes de programación, como Python, Java, o Ruby, también necesitan acceder a los **servicios del sistema operativo**, pero no siempre pueden usar directamente las funciones de la **librería del sistema** escrita en C, como lo hacen C y C++. Para facilitar este acceso, los lenguajes suelen incluir una **librería estándar** que proporciona una interfaz más amigable y adaptada a las particularidades

del lenguaje, permitiendo a los programadores interactuar con el sistema operativo de manera más sencilla.

Librerías estándar y acceso a servicios del sistema operativo

- Las **librerías estándar** de cada lenguaje no son parte del sistema operativo, sino que forman parte de las herramientas de desarrollo de dicho lenguaje.
- Estas librerías suelen ofrecer abstracciones y funciones adicionales, como estructuras de datos, algoritmos o utilidades para manipular archivos y realizar tareas comunes.
- Al final, las funciones de las **librerías estándar** invocan a la **librería del sistema** del sistema operativo para realizar operaciones fundamentales, como la lectura o escritura de archivos, la gestión de procesos, o la comunicación en red.

Ejemplo: Acceso a archivos

En lenguajes como **C** o **C++**, el manejo de archivos y otros recursos se abstrae a través del concepto de **flujos (streams)**. Los flujos no solo se utilizan para leer y escribir en archivos, sino también para gestionar la entrada y salida de datos por teclado, monitor, impresoras, e incluso conexiones de red.

- **Streams** ofrecen una interfaz más versátil, con características adicionales como el manejo de **texto o binarios**, y el uso de **buffering**, lo que mejora el rendimiento al acumular datos antes de enviarlos en bloque.
- Aunque las librerías estándar de C y C++ abstraen el acceso a archivos a través de flujos, al final de todo el proceso, es el sistema operativo quien gestiona los archivos y los recursos físicos, y por ello la librería estándar usa las **funciones de la librería del sistema**, como la llamada al sistema **open()**.

Abstracciones y pérdida de control

- Las **librerías estándar** suelen abstraer detalles del sistema operativo para simplificar el desarrollo, pero a veces esto implica una **pérdida de control** sobre características específicas del sistema.
  - Por ejemplo, la llamada al sistema **open()** permite especificar permisos para el archivo y crear archivos temporales, lo cual no es siempre posible con las interfaces de **streams** en C o C++, ya que estas abstraen conceptos de permisos y temporalidad que no son comunes a todas las fuentes de flujo.

Portabilidad y control

- Usar las funciones de la **librería estándar** de un lenguaje garantiza **portabilidad**, ya que el programa puede ejecutarse en cualquier sistema operativo que soporte dicho lenguaje y su librería estándar.
- Sin embargo, si se decide usar directamente las funciones de la **librería del sistema**, se gana control sobre funcionalidades específicas, pero se pierde portabilidad. Esto se debe a que el programa se vuelve dependiente de las características de un sistema operativo específico, limitando su uso a aquellos sistemas con una **librería del sistema** compatible.

En resumen, aunque las **librerías estándar** ofrecen una capa de abstracción que simplifica la programación y garantiza portabilidad, en algunos casos puede ser necesario acceder directamente a las funciones de la **librería del sistema** para aprovechar características específicas del sistema operativo. No obstante, esto puede comprometer la portabilidad del programa a otras plataformas.

Con todas las piezas juntas

En la **Figura 6.1**, se muestra cómo interactúan los diferentes elementos en el contexto de la programación en **C** y **Python** en **Microsoft Windows**, con el ejemplo de invocar la apertura de archivos mediante las funciones **fopen()** en C y **file()** en Python.

Descripción del proceso en Windows:

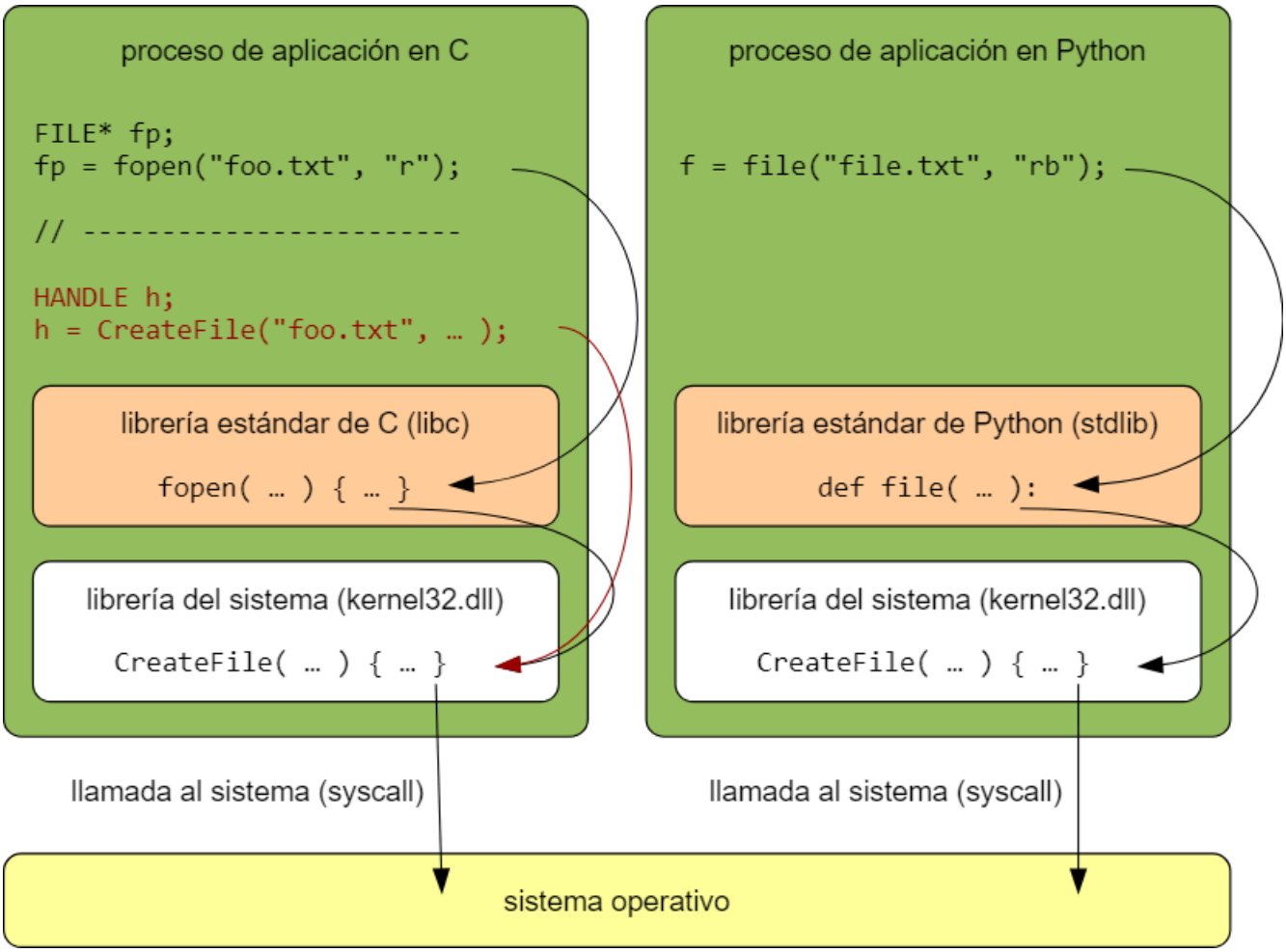
- **Librerías estándar**:
  - En C, se puede usar la función **fopen()** de la librería estándar para abrir un archivo.
  - En Python, se utiliza la función **file()** (aunque en versiones más recientes de Python, la función es simplemente **open()**).
- Ambas funciones de las librerías estándar, en última instancia, invocan la función **CreateFile()** de la **librería del sistema** de Windows. Esta función es la que directamente interactúa con el núcleo del sistema operativo, ejecutando la operación de abrir un archivo.
- La función **CreateFile()** de Windows ofrece una gran cantidad de opciones y características, como especificar permisos, modos de acceso, acciones cuando el archivo ya existe, entre otras, lo que la hace más flexible que **fopen()**.

Diferencias entre C y Python en Windows:

- En un programa en **C**, el programador puede optar por usar directamente la función **CreateFile()** de la librería del sistema si necesita funcionalidades más avanzadas. Sin embargo, hacerlo compromete la **portabilidad**, ya que **CreateFile()** es específico de Windows.
- Por otro lado, en **Python**, no se puede invocar **CreateFile()** directamente de forma sencilla, ya que el lenguaje abstrae este acceso a través de su propia **librería estándar**.

Ventajas y desventajas:

- Usar **CreateFile()** en C proporciona acceso a todas las características del sistema operativo, pero compromete la portabilidad, ya que esta función solo está disponible en **Windows**.
- **fopen()** es más portátil, ya que está disponible en cualquier compilador de C, pero tiene menos opciones en comparación con **CreateFile()**.



Similaridad en GNU/Linux (Figura 6.2):

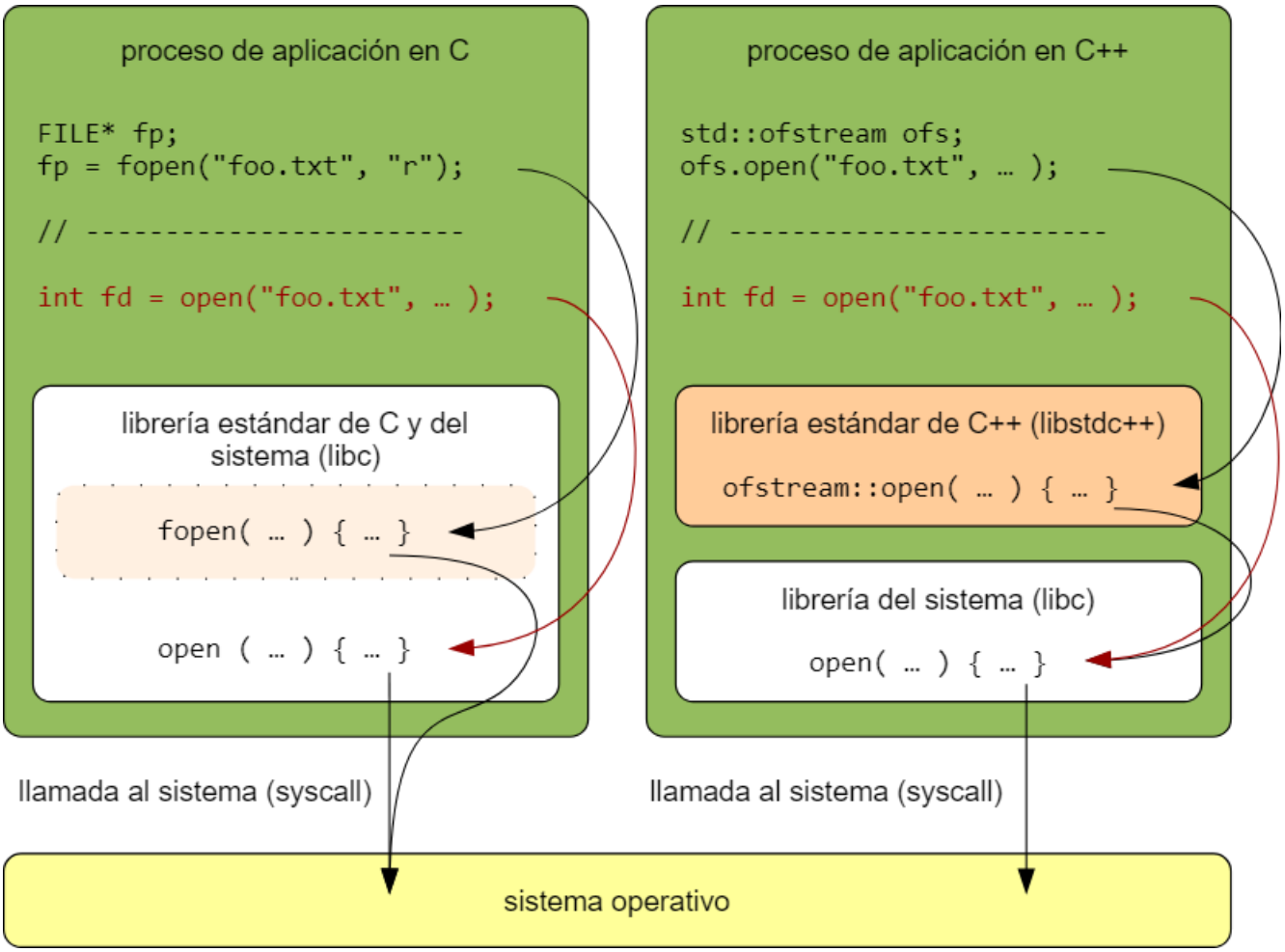
En **GNU/Linux**, las funciones equivalentes son **open()** y **fopen()**. La **Figura 6.2** muestra cómo funciona este proceso en un entorno compatible con **POSIX**. La principal diferencia es que en **Linux**, tanto **fopen()** como la llamada al sistema **open()** se encuentran en la misma **librería del sistema**, ya que POSIX se diseñó como un **superconjunto** de la librería estándar de C.

- **fopen()** en C y **std::ofstream::open()** en C++ invocan internamente la llamada al sistema **open()**, que se encarga de interactuar con el sistema operativo para abrir un archivo.
- Al igual que en Windows, los programas en C y C++ pueden llamar directamente a la función **open()** de la librería del sistema si necesitan características adicionales que **fopen()** no ofrece, como la manipulación de permisos específicos o la creación de archivos temporales.

Resumen:

- **En Windows**, los programas en C pueden usar **fopen()** para portabilidad o **CreateFile()** para características avanzadas, mientras que en Python solo se puede usar la función de la librería estándar.
- **En GNU/Linux**, la llamada a **fopen()** y **open()** se unifican dentro de la misma librería, manteniendo compatibilidad con el estándar **POSIX**, lo que simplifica la interfaz y ofrece portabilidad sin renunciar a las funcionalidades avanzadas.

Este diseño en sistemas **POSIX** permite que las funciones de la librería estándar de C y las llamadas al sistema estén más integradas, facilitando el acceso a características avanzadas sin perder la portabilidad entre sistemas operativos compatibles con **POSIX**.



---

7. Operación del sistema operativo

Dado el sistema operativo y los procesos de usuarios comparten los recursos del sistema informática, necesitamos estar seguros de que un error en un programa solo afecte al proceso que lo ejecuta. Por eso, es necesario establecer mecanismos de protección frente a los errores en los programas que se ejecutan en el sistema.

Software controlado mediante interrupciones

Los sucesos que requieren la atención del sistema casi siempre se indican mediante una interrupción:

- Cuando un proceso comete un error, lo que se genera es una excepción en la CPU. Esta excepción despierta al sistema operativo para que haga lo que sea más conveniente.
- Cuando un proceso necesita un servicio, lo que hace es lanzar una llamada al sistema, que no es más que ejecutar una instrucción que lanza una excepción en la CPU. Esta excepción despierta al sistema operativo para que atienda la petición.
- Cuando los dispositivos de E/S requieren la atención del sistema operativo, se genera una interrupción en la CPU, que despierta al sistema operativo.

Esto funciona así porque el sistema operativo configura la CPU durante el arranque para que si ocurre cualquier interrupción o excepción, la ejecución, salte a rutinas en el código del núcleo, con el objeto de darles el tratamiento adecuado.

Si ningún proceso realiza una acción ilegal o pide un servicio, ni ningún dispositivo de E/S pide la atención del sistema, el sistema operativo permanece

inactivo esperando a que algo ocurra.

Operación en modo dual

Para proteger el sistema de programas con errores, es necesario poder distinguir entre la ejecución del código del sistema operativo y del código de los programas de usuario, de tal forma que el código de los programas de usuario esté más limitado en lo que puede hacer que el del sistema operativo.

El método que utilizan la mayor parte de los sistemas operativos consiste en utilizar algún tipo de soporte en la CPU que permita diferencias entre varios modos de ejecución y restringir la utilización de las instrucciones peligrosas, llamas **instrucciones privilegiadas**, para que solo puedan ser utilizadas en el modo en el que se ejecuta el código del sistema operativo.

Modos de operación

- En el **modo usuario**, en el que se ejecuta el código de los procesos de los usuarios. Si se hace un intento de ejecutar una instrucción privilegiada en este modo, el hardware la trata como ilegal y genera una excepción que es interceptada por el sistema operativo, en lugar de ejecutar la instrucción.
- En el **modo privilegiado**, también denominado **modo supervisor**, **modo del sistema** o **modo kernel**, se ejecuta el código de las tareas del sistema operativo. La CPU es la encargada de garantizar que las instrucciones privilegiadas solo pueden ser ejecutadas en este modo.

El modo actual de operación puede venir indicado por un **bit de modo** en alguno de los registros de configuración de la CPU.

Comúnmente, en el grupo de las **instrucciones privilegiadas** se suelen incluir:

- La instrucción para conmutar al modo usuario desde el modo privilegiado.
- Las instrucciones para acceder a dispositivos de E/S.
- Las instrucciones necesarios para la gestión de las interrupciones.

Niveles de Privilegio en Procesadores x86

- **Modos de operación**: La arquitectura Intel x86 soporta 4 niveles de privilegio, donde el modo 0 es el más alto, y el modo 3 es el más bajo. Los modos 1 y 2 están diseñados para controladores de dispositivos, que requieren más privilegios que los usuarios, pero menos que el núcleo.
- **Uso en sistemas operativos**: Sistemas populares como Windows, macOS, Linux y Android solo utilizan los modos 0 y 3, ya que usar más modos no ofrece ventajas significativas y complica la portabilidad a procesos con solo dos modos.
- **Modo virtualización**: Procesadores x86 modernos incluyen un modo -1 para gestionar máquinas virtuales, donde el sistema operativo anfitrión supervisa al núcleo virtualizado que se ejecuta en modo 0.
- **Modos real y protegido**: Los procesadores x86 inician en el modo real por compatibilidad con CPUs antiguas, y luego el sistema operativo arranca en modo protegido, habilitando el direccionamiento de 32 o 64 bits y el uso de los niveles de privilegio para la supervisión del sistema.

Ejecución de instrucciones

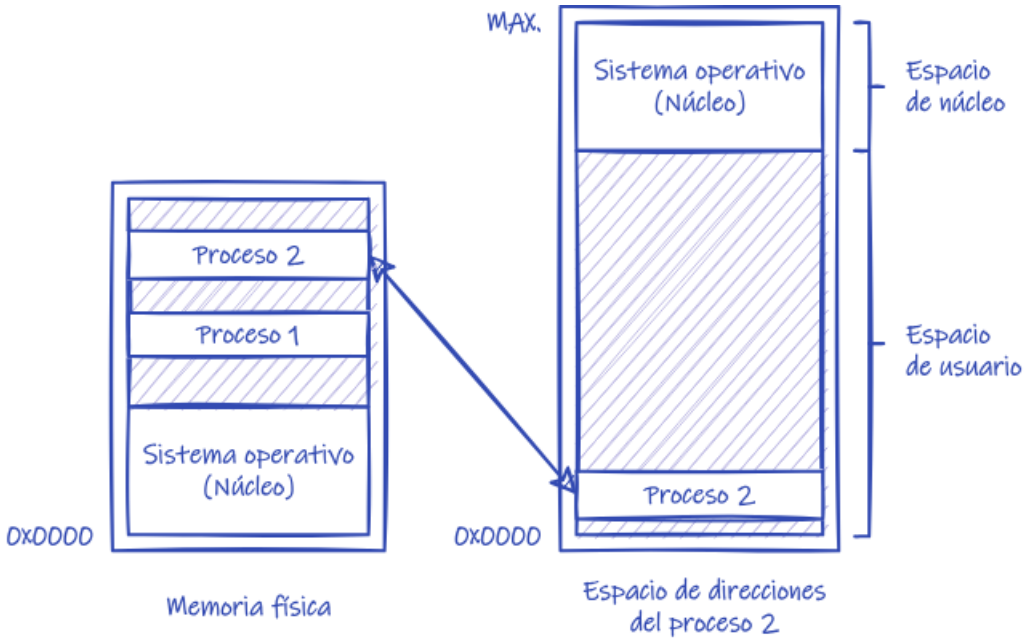
Ciclo de vida en Modo Dual de Operación:

- **Inicio del sistema**: La CPU arranca en **modo privilegiado** (bit de modo a 0), lo que permite cargar y ejecutar el núcleo del sistema operativo.
- **Cambio a modo usuario**: El núcleo cambia al **modo usuario** (bit de modo a 1) antes de ceder la CPU a un proceso de usuario, asegurando que estos procesos se ejecuten con menos privilegios.
- **Interrupciones y excepciones**: La CPU retorna al **modo privilegiado** cuando ocurre una interrupción o excepción, permitiendo que el sistema operativo maneje estas situaciones de manera segura.
- **Control del sistema operativo**: Los sistemas operativos son controlados mediante interrupciones. Al activar el modo privilegiado en cada interrupción, se garantiza que las tareas del sistema operativo siempre se ejecuten con los máximos privilegios.
- **Manejo de errores**: En modo dual, el hardware detecta errores de ejecución y los reporta al sistema operativo a través de excepciones. El sistema operativo es responsable de manejar estos errores, como finalizar un programa si intenta usar instrucciones ilegales o acceder a memoria no permitida.

Protección de la memoria

Memoria principal y su gestión en sistemas operativos:

- **División de memoria**: La memoria se divide en dos partes:
  - **Espacio del núcleo**: Alberga el núcleo del sistema operativo. Puede estar en la parte baja o alta de la memoria según la arquitectura de la CPU y la ubicación del vector de interrupciones
  - **Espacio de usuario**: Contiene los procesos de los usuarios.
- **Ubicación del núcleo según la arquitectura**:
  - En **x86**, el vector de interrupciones comienza en la dirección 0×0000000, por lo que el sistema operativo suele estar en la parte baja.
  - En **MIPS**, las interrupciones se gestionan desde la dirección 0×80000180, por lo que el sistema operativo suele estar en la parte alta de la memoria.
- **Protección de la memoria**:
  - **Acceso restringido**: Los procesos no tienen acceso completo a la memoria física. El sistema operativo proporciona a cada proceso un espacio de direcciones virtual, simulando que el proceso tiene toda la memoria para sí mismo.
  - **Espacio de direcciones virtual**: Es el conjunto de direcciones que puede generar la CPU para un proceso.
  - **Conversión de direcciones**: Durante la ejecución, la CPU convierte direcciones virtuales en físicas mediante la **Unidad de Gestión de Memoria** (MMU). Las direcciones físicas son las que realmente accede la memoria.



- **Ventajas de la gestión de memoria**:
  - **Aislamiento de procesos**: Cada proceso cree tener toda la memoria, evitando que acceda a la de otros.
  - **Control de acceso**: Se definen permisos de lectura, escritura y ejecución para evitar accesos no autorizados. Los accesos indebidos, como ejecutar código en zonas no permitidas, generan excepciones.

El temporizador

El **temporizador** se configura por el sistema operativo durante el arranque del sistema para interrumpir a la CPU a intervalos regulares. Así, cuando el temporizador interrumpe, el control se transfiere automáticamente al núcleo del sistema. Entonces este puede:

- Conceder más tiempo al proceso en ejecución.
- Detenerlo y darle más tiempo de CPU en el futuro.
- Tratar la interrupción como un error y al terminar el programa.

El temporizador se utiliza para asegurar que ningún proceso acapara la CPU indefinidamente. Por ejemplo, un programa mal desarrollado que entra en un bucle infinito, del que no sale jamás.

Obviamente, las instrucciones que pueden modificar el contenido del temporizador son instrucciones privilegiadas.

Máquinas virtuales

Las máquinas virtuales permiten que los procesos accedan a recursos de hardware a través de una interfaz simulada, en lugar de interactuar directamente con el hardware real. Las instrucciones de E/S, que son privilegiadas, son interceptadas por el sistema operativo gracias al modo dual. En lugar de bloquearlas, las redirige hacia componentes de hardware virtual, simulando dispositivos como controladores de disco o tarjetas de sonido. Así, el proceso tiene la ilusión de estar accediendo directamente al hardware.

El software de gestión de las máquinas virtuales se encarga de implementar los componentes virtuales y gestionar las instrucciones privilegiadas que el proceso

intenta ejecutar. De esta forma, el proceso sigue funcionando sin ser consciente de que opera sobre una máquina virtual.

El modo dual es esencial para esta virtualizaci
on, ya que permite que el sistema operativo supervise estas operaciones, brindando soporte para máquinas virtuales en sistemas operativos modernos.

Paravirtualización

La paravirtualización es una técnica que mejora la eficiencia de las máquinas virtuales, ya que evita la necesidad de simular el hardware virtual. En lugar de usar controladores diseñados para hardware real, el sistema operativo de la máquina virtual usa controladores específicamente creados para trabajar en entornos virtuales. Estos controladores comunican las peticiones directamente al sistema operativo del anfitrión, utilizando una interfaz especial proporcionada por el software de gestión de máquinas virtuales. Esto reduce la sobrecarga de simular hardware, mejorando el rendimiento.

Arranque del sistema

Desde el momento en que el ordenador se pone en marcha hasta que el sistema operativo inicia su ejecución se realizan una serie de operaciones. Estos son los pasos más comunes en el arranque de un sistema:

1. Llega a la CPU una señal de **RESET** motivada por el encendido del sistema o por un reinicio.
2. La CPU inicializa el contador de programa a una dirección predefinida de la memoria. En esa dirección está el *bootstrap* inicial.

El *bootstrap* es el programa que se encarga en primera instancia del arranque. Debe estar almacenado en una memoria no volátil, porque la RAM está en un estado indeterminado en el momento de arranque. En los PC, el *bootstrap* forma parte del *firmware* de las placas madres.

> El término *firmware* viene de que por sus características se sitúa en algún lugar entre el hardware y el software. Concretamente es un componente software instalado en un dispositivo hardware para encargarse de su control a bajo nivel.

Tareas del bootstrap

1. **Diagnóstico de la máquina**: El *bootstrap* se detiene en este punto si el sistema no supera el diagnóstico.
2. **Inicializar el sistema**: Por ejemplo, configurar los registros de la CPU, inicializar los dispositivos y contenido de la memoria, etc.
3. **Iniciar el sistema operativo**.

El sistema operativo puede almacenarse en diferentes ubicaciones dependiendo del dispositivo. En consolas, móviles y dispositivos empotrados, se encuentra en memorias de solo lectura como ROM o Flash. Como estas memorias son más lentas que la RAM, el proceso de arranque (*bootstrap*) a menudo copia el sistema operativo a la RAM antes de iniciarlo. En sistemas más grandes, como los de propósito general, el sistema operativo se almacena en disco.

En sistemas antiguos, el *bootstrap* lee el gestor de arranque desde una posición fija del disco, lo copia en memoria y lo ejecuta. Esto ocurre en PCs antiguos que usan BIOS y particiones MBR.

> También se llama MBR a ese bloque 0 del disco donde está el gestor de arranque. De hecho MBR son las siglas de **Master Boot Record** o registro de arranque principal.

El código inicial del arranque (*bootstrap*), que se encuentra en el bloque 0 del disco, generalmente tiene la tarea de cargar el resto del gestor de arranque, ya que el código debe caber en solo 512 bytes. En sistemas antiguos, como los que utilizan BIOS y MBR, este proceso era más limitado.

En PCs modernos que usan UEFI y particiones GPT, la UEFI puede leer directamente el sistema de archivos para localizar y cargar el gestor de arranque completo. Este gestor es el encargado de buscar el núcleo del sistema operativo en el sistema de archivos, cargarlo e iniciar su ejecución.

Desde este punto, cada sistema operativo continúa el proceso de arranque de manera diferente. Por ejemplo, en sistema UNIX, el proceso continúa en modo texto.

Arranque de sistemas UNIX

Cuando el núcleo de un sistema UNIX se inicia, realiza las siguientes tareas claves:

1. Configura el entorno para ejecutar procesos, incluyendo la configuración de interrupciones, modos de operación (privilegiado y usuario), gestión de la memoria, inicialización de dispositivos, montaje del sistema de archivos raíz y creación del proceso inactivo.
2. Crea el proceso **init** (PID 1), el cual es el primer proceso. En sistemas GNU/Linux modernos, el proceso init más común es **systemd**.
3. El planificador de la CPU toma el control y selecciona init para ejecutarse, ya que es el único proceso activo.
4. El proceso **init** ejecuta scripts que configuran servicios del sistema (demonios), como gestión de dispositivos, registro de eventos, particiones, entre otros.
5. Finalmente, **init** configura el entorno de usuario, inicia procesos **login** para terminales y los supervisa para reiniciarlos si es necesario.

Este proceso asegura que el sistema esté listo para ejecutar procesos de usuario y servicios.

> Aunque, por lo general, un sistema de escritorio tiene una única pareja de teclado y monitor, por lo tanto, una única terminal real; el sistema suele estar configurado para crear varios terminales virtuales entre los que el usuario puede conmutar usando las combinaciones de teclas adecuadas.

Los procesos **login** se encargan de autenticar a los usuarios y de iniciar y configurar su sesión:

1. Muestran una pantalla de inicio de sesión donde se solicita el nombre del usuario y su contraseña.
2. Autentican al usuario comprobando las credenciales proporcionadas.
3. Si la autenticación es positiva, el proceso **login** cambia su identidad actual —generalmente de **root** o administrador— por la del usuario autenticado, configura la sesión y sustituye su programa actual por el del *intérprete de comandos* configurado para ese usuario.

El *intérprete de comandos* completa la configuración del entorno usando los archivos de configuración, muestra el prompt y espera el primer comando del usuario.

---

8. Sistemas operativos por su estructura

Estructura sencilla

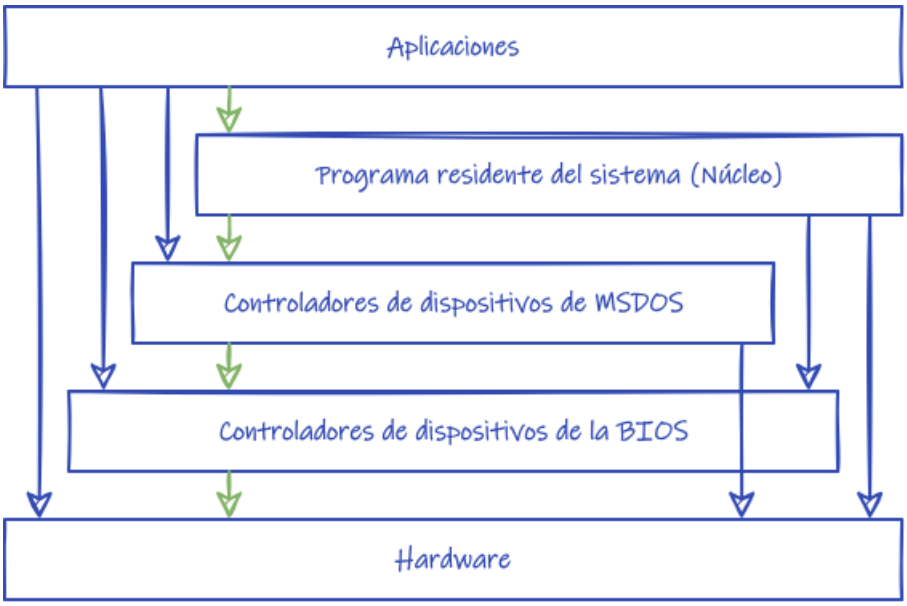
Los sistemas con **estructura sencilla** se caracterizan por:

- No tener una estructura bien definida. Los componentes no están bien separados y las interfaces entre ellos no están bien definidas.
- Son sistemas monolíticos, dado que gran parte de la funcionalidad del sistema se implementa en el núcleo.

En la actualidad, este tipo de estructuras se usa en sistemas que deben ejecutarse en hardware muy limitado, como en sensores conectados, termostatos, sistemas de control o electrodomésticos.

MS-DOS

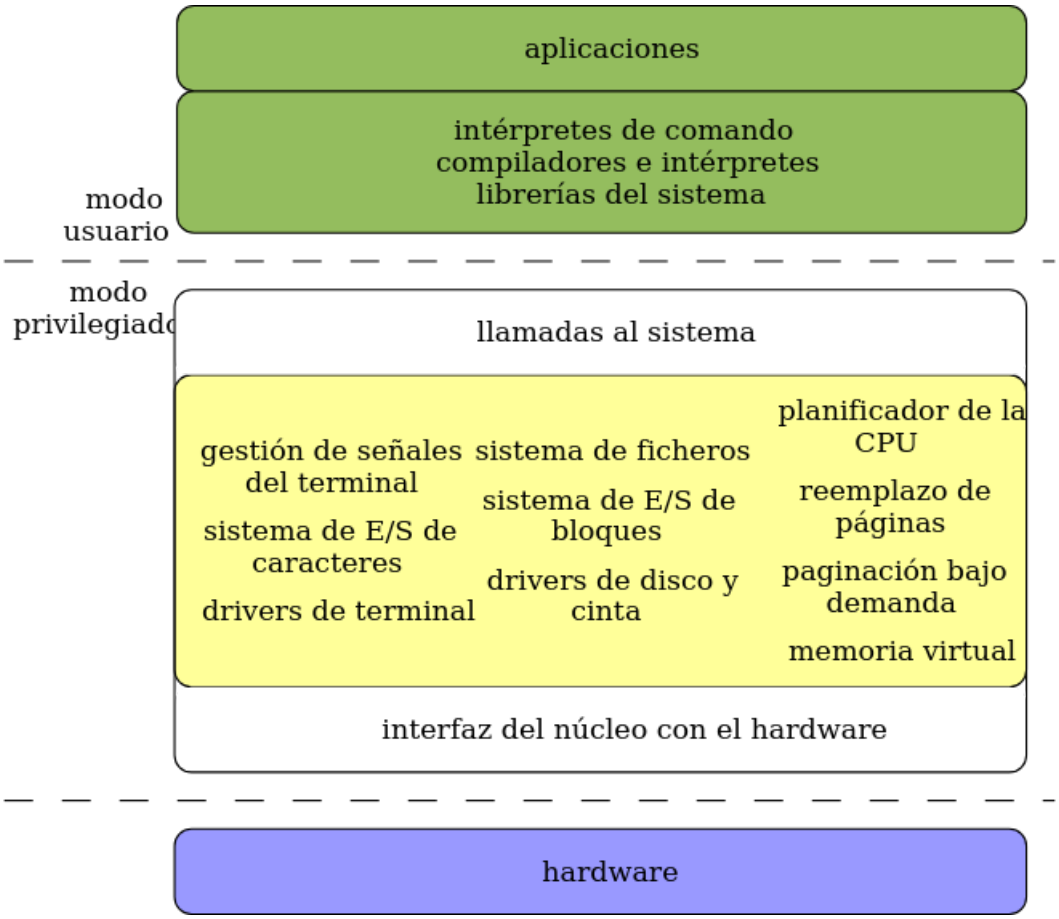
Por ejemplo, en el **MS-DOS** los programas de aplicación podían acceder directamente a toda la memoria y a cualquier dispositivo. Disponiendo de esa libertad, un programa erróneo cualquiera podía corromper el sistema completo.



Como el **Intel 8086** para el que fue escrito **MS-DOS** no proporcionaba un modo dual de operación, los diseñadores del sistema no podían evitar que los programas de usuario accedieran directamente al hardware ni tenían forma de proteger las distintas partes del sistema operativo.

UNIX

Otro ejemplo es el de UNIX original, donde sí había una separación clara entre procesos de usuario y código del sistema, pero juntaba mucha funcionalidad en el núcleo del sistema.



El núcleo proporciona la planificación de CPU, la gestión de la memoria, el soporte de los sistemas de archivos y muchas otras funcionalidades del sistema operativo. En general, se trata de una enorme cantidad de funcionalidad que es difícil de implementar y mantener, si no se compartimenta adecuadamente.

Tanto MS-DOS como UNIX eran originalmente sistemas pequeños y simples, limitados por las funcionalidades del hardware de su época, que fueron creciendo más allá de las previsiones originales. Lo cierto es que con mejor soporte del hardware se puede dividir el sistema operativo en piezas más pequeñas y apropiadas que las del MS-DOS y el UNIX original.

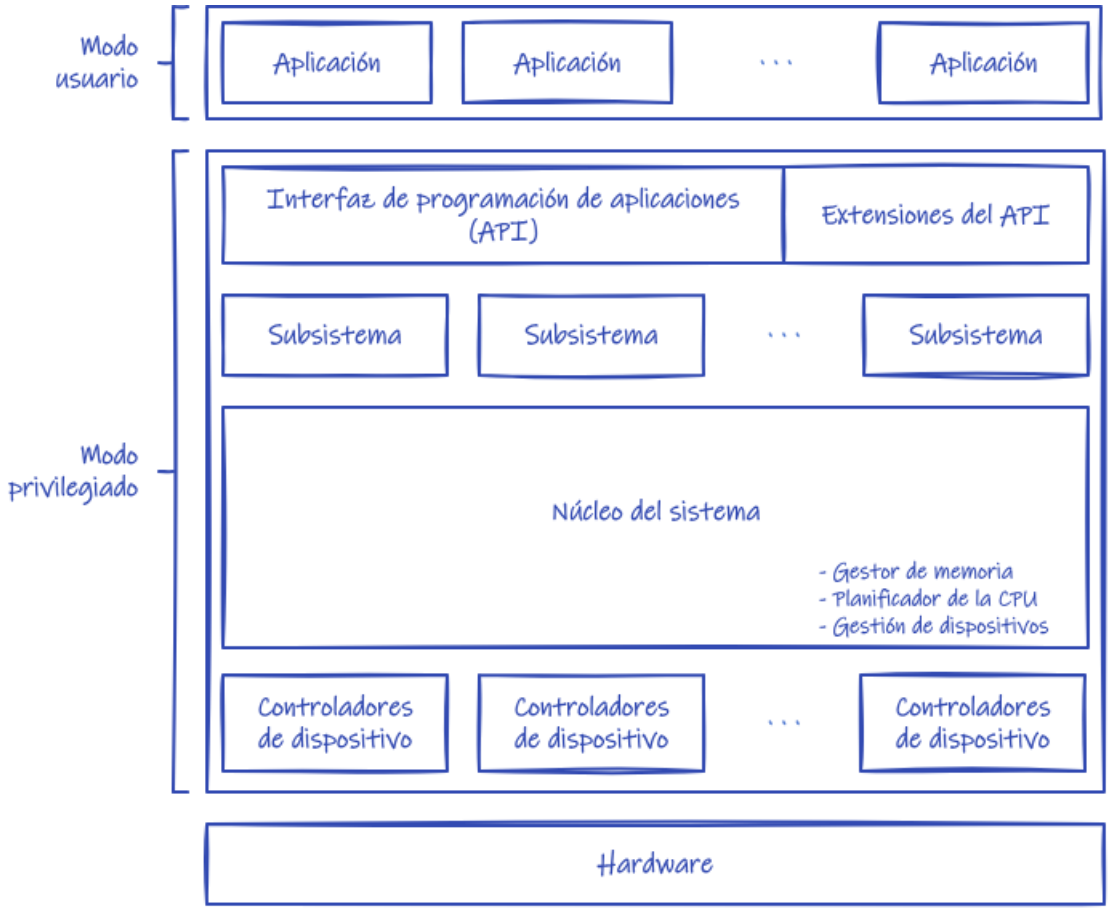
Estructura en capas

Los sistemas con **estructura en capas** se caracterizan por:

- La funcionalidad se divide en capas, donde cada capa solo utiliza las funciones de la capa inferior a través de una interfaz definida.
- Cada capa oculta los detalles de su implementación a la capa superior, similar a la programación orientada a objetos.
- Escalan mejor que los sistemas simples, ya que al hacer cambios, solo es necesario revisar la capa afectada, siempre que no se altere la interfaz.
- Son menos eficientes debido a la sobrecarga que cada capa añade al transferir datos y transformar argumentos entre capas.
- A pesar de ser por capas, siguen siendo **sistemas monolíticos**, ya que gran parte de la funcionalidad está implementada en el núcleo compartimentado.

Sin embargo, hoy en día se prefieren **estructuras modulares**, que ofrecen las mismas ventajas sin las complicaciones de diseño que tienen los sistemas en capas.

Un ejemplo de este tipo de sistemas operativos es el **OS/2**,



Dificultades con el diseño

Diseñar un sistema con **estructura en capas** requiere una planificación cuidadosa, ya que cada capa solo puede utilizar los servicios de las capas inferiores. Un ejemplo de este desafío es la relación entre el **planificador de la CPU** y la **gestión del almacenamiento secundario**. El planificador necesita acceso a los datos en memoria, algunos de los cuales pueden almacenarse en disco. Por lo tanto, la gestión del almacenamiento debe estar en una capa inferior. Sin embargo, el planificador también necesita asignar la CPU a otro proceso cuando el actual realiza una operación de **E/S**, lo que implica que la gestión del almacenamiento debería estar en una capa superior.

La solución a esta **dependencia circular** es colocar ambos componentes en la misma capa. Estas dependencias entre componentes no son infrecuentes y complican el diseño por capas, debido a la interdependencia inherente de los componentes del sistema operativo.

> Al final, la solución de compromiso es tender hacia sistemas con muy pocas capas, donde cada una tiene mucha funcionalidad. Esto limita mucho las ventajas de esta técnica, porque no permite compartimentar el núcleo tanto como sería deseable.

Microkernel

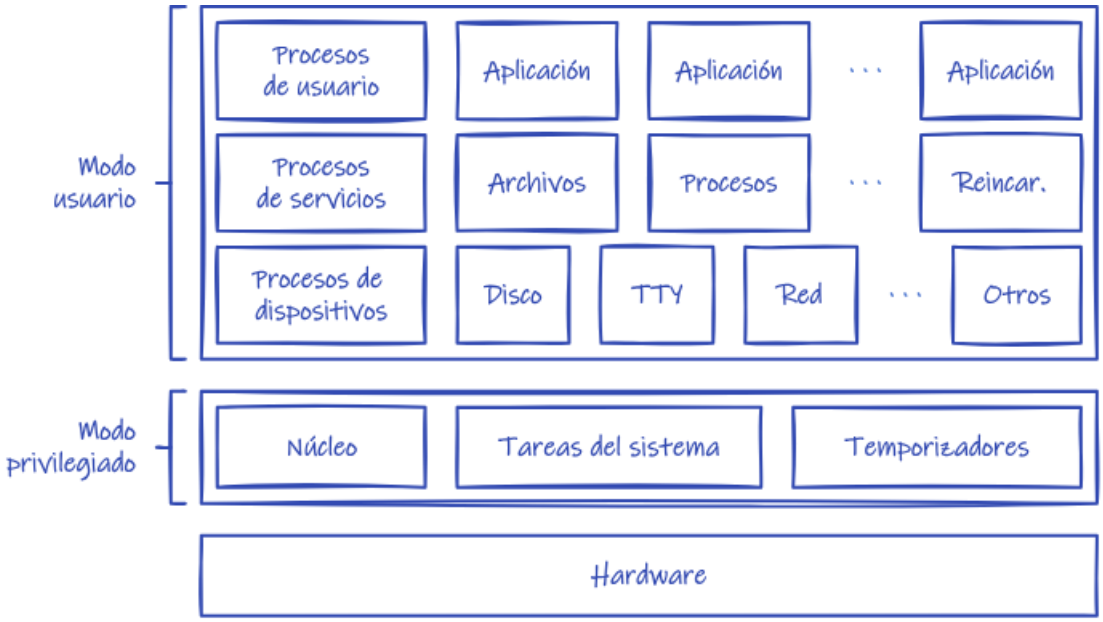
Los sistemas con **estructura microkernel** se caracterizan por:

- Eliminar todos los componentes no esenciales del núcleo e implementarlos como procesos de usuario.
- Un núcleo **microkernel** proporciona funciones mínimas de gestión de procesos y de memoria y algún mecanismo de comunicación entre procesos. Algunos **microkernel** reales incluyen en el núcleo algunas funcionalidades adicionales.

- El mecanismo de comunicación permite a los procesos de los usuarios solicitar servicios a los componentes del sistema. También sirve para que los componentes del sistema se comuniquen entre sí y se pidan servicio.

Este tipo de estructura se utilizan principalmente en sistemas empotrados, como ocurre con los de estructura sencilla. Sin embargo, la **estructura microkernel** necesita un hardware algo más potente, que admita modo dual de operación; siendo especialmente interesante en sistemas críticos o cuando hay especial preocupación por la seguridad. Algunos ejemplos son los sistemas de automoción o los equipos médicos.

Dado que los componentes del sistema están aislados unos de otros, el mecanismo de comunicación entre procesos es la única forma que tienen los procesos de los usuarios y los componentes, de solicitarles un servicio.



Esquema de la estructura microkernel de MINIX 3.

Generalmente esta comunicación se implementa mediante paso de mensajes

Entre los beneficios de estos sistemas operativos se incluyen:

- **Facilidad a la hora de añadir nuevas funcionalidades**: Los nuevos servicios son añadidos como aplicaciones de nivel de usuario, por lo que no es necesario hacer modificaciones en el núcleo. Desarrollar en el modo privilegiado siempre es más peligroso que en el modo usuario, porque los errores pueden ser **catastróficos**: bloqueo o caída de sistema, corrupción de datos, etc.
- **Facilidad a la hora de llevar el sistema a otras plataformas**: Puesto que el núcleo es muy pequeño, resulta muy sencillo de portar a otras plataformas.
- **Más seguridad y fiabilidad**: Puesto que la mayor parte de los servicios se ejecutan como procesos separados de usuario, un servicio que falla no puede afectar a otros ni puede ser utilizado para ganar acceso a otros servicios o al núcleo. Además se pueden implementar estrategias para mejorar la tolerancia a fallos, como reiniciar un servicio que ha fallado, como si fuera un programa cualquiera.

Rendimiento

El mayor inconveniente es el pobre rendimiento que puede tener, causado por la sobrecarga que añade el mecanismo de comunicación.

Por ejemplo, **Microsoft Windows NT*** nació con una estructura de **microkernel** en capas donde una parte importante de los servicios eran proporcionados por unos procesos de usuario llamado subsistemas.

El sistema operativo podía podía mostrar diferentes personalidades o *entornos operativos*, a través del uso de subsistemas ambientales, que también se ejecutaban como procesos de usuario. Las aplicaciones de **Microsoft Windows NT** se comunicaban con estos subsistemas utilizando un mecanismo de comunicación denominado **LPC** (Local Intel-Process Communication).

Con esta estructura, la pérdida de rendimiento respeto a **Microsoft Windows 95** era tan importante, que los diseñadores se vieron obligados a mover más servicios al espacio del núcleo en la versión 4.0. El resultado es que los Windows sucesores a **Windows NT 4.0** tienen una arquitectura más monolítica que microkernel, ya que aunque muchos servicios siguen siendo proporcionados por procesos de usuario, esto solo ocurre con aquellos donde el rendimiento no es un factor crítico.

> Microsoft Windows XP tiene 280 llamadas al sistema a las que hay que sumar las más de 650 llamadas del subsistema gráfico, que también se aloja en el núcleo desde Microsoft Windows NT 4.0. Mientras que Microsoft Windows NT 3.51 tenía algo menos de 200 llamadas al sistema.

Sin embargo, varios sistemas operativos siguen utilizando núcleos **microkernel**. Ambos son sistemas operativos de tiempo real, que basan en la estructura de **microkernel** su estabilidad como sistema para tareas críticas.

En la figura del esquema MINIX 3 se observa el pequeño tamaño del núcleo, causando que la mayor parte de la funcionalidad reside en los procesos de servicios de controladores de dispositivo.

MINIX 3 es un sistema compatible POSIX. Así que soporta las llamadas al sistema definidas por este estándar, pero estas se convierten en mensajes enviados al servidor correspondiente con la petición, y no en llamadas directas al núcleo. Para que un servidor pueda atender una petición, quizás tenga que enviar peticiones a otros servidores o controladores de dispositivo. Incluso pueden tener que hacer llamadas al núcleo, para solicitar alguna operación privilegiada que no se pude implementar en el modo usuario. Por ejemplo, operaciones de E/S o el acceso a tablas del núcleo.

> Es este trasiego de mensajes con peticiones y respuestas para resolver una petición de un proceso de usuario, lo que teóricamente justifica el menor rendimiento de los sistemas **microkernel**.

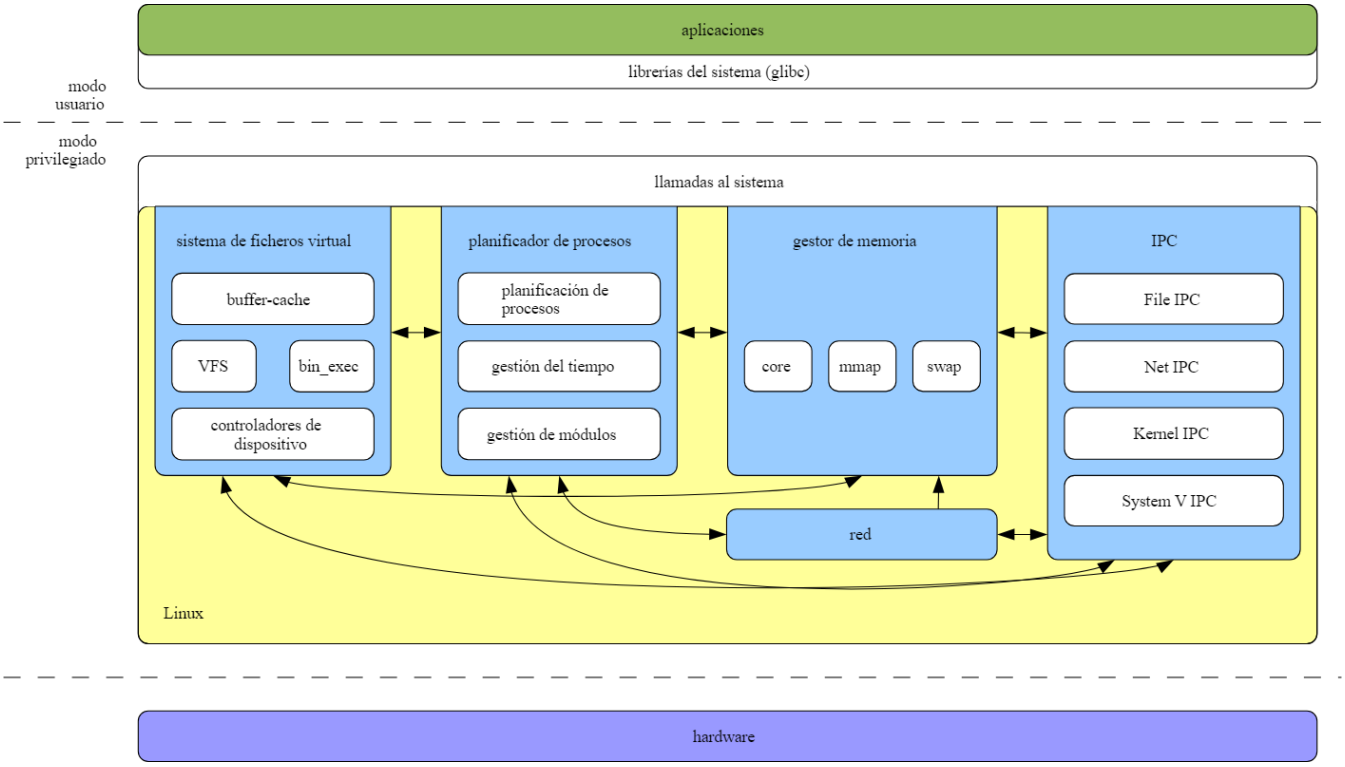
Estructura modular

Los sistemas con **estructura modular** se caracterizan por:

- Dividir el núcleo en módulos, cada uno de los cuales implemente funciones y servicios concretos y se comunican entre sí a través de una interfaz bien definida.
- Como en la OOP, cada módulo oculta al resto los detalles de su implementación.
- Todos los módulos pueden llamar a funciones de la interfaz de cualquier otro módulo, a diferencia de los sistemas operativos con **estructura en capas**, donde una capa solo podía usar a la inmediatamente inferior.
- También son sistemas **monolíticos**, dado que gran parte de la funcionalidad del sistema se implementa en el núcleo, aunque ahora el núcleo esté compartimentado en módulos.

La mayor parte de los sistemas operativos de propósito general, que se instalan tanto en sistema de escritorio como en servidores, son de este tipo. También en los *smartphones*, *smart TV*, y, en general, en muchos sistemas empotrados.

Estos núcleos suelen disponer de un pequeño conjunto de componentes fundamentales que se cargan durante el arranque. Posteriormente pueden cargar módulos adicionales, tanto durante la inicialización del sistema como en tiempo de ejecución.



Se asemejan a los núcleos **microkernel**, ya que el módulo principal solo tiene funciones básicas. Sin embargo, los núcleos modulares:

- **Son más eficientes** al no necesitar un mecanismo de comunicación, puesto que los componentes se cargan en la memoria destinada al núcleo, por lo que pueden llamarse directamente.
- **Son menos seguros y fiables**, puesto que gran parte de funcionalidad se ofrece desde el modo privilegiado. Un error en cualquier componente puede comprometer o hacer caer el sistema.
  Este tipo de estructura es la utilizada en los UNIX modernos. También se puede considerar que los sistemas Windows actuales, tienen estructura modular.