APUNTES-TEMA-1-2-y-3-SISTEMAS-OP...

renzosantonim

Sistemas Operativos

2º Grado en Ingeniería Informática

Escuela Superior de Ingeniería y Tecnología
Universidad de La Laguna

APUNTES SISTEMAS OPERATIVOS

PRIMER SEGUIMIENTO TEMAS[1,2,3]

Renzo Santoni Moyano
Universidad de La Laguna

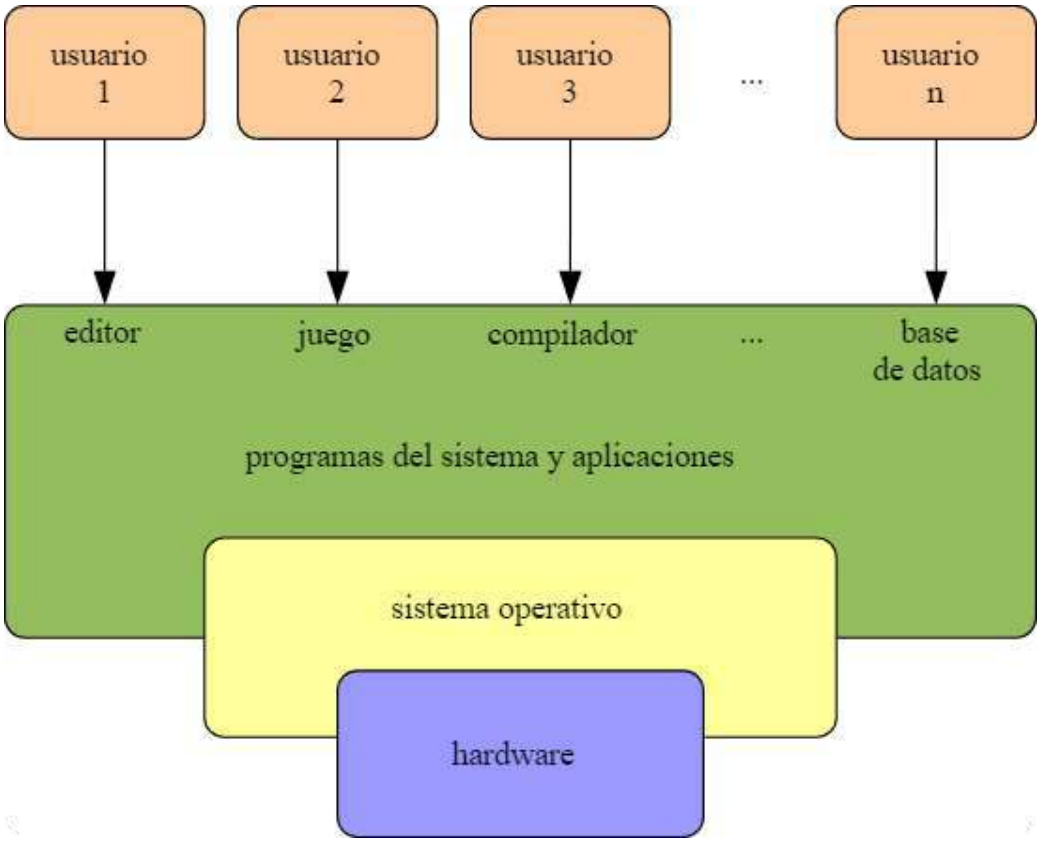
TEMA 1: INTRODUCCIÓN

1. ¿QUÉ ES UN SISTEMA OPERATIVO?

1.1 Definición de sistema operativo

El **sistema operativo** es el conjunto de programas informáticos que permite la administración eficaz de los recursos de una computadora / dispositivo electrónico.
Estos programas comienzan a trabajar apenas se enciende el equipo, ya que gestionan el hardware desde los niveles más básicos y permiten además la interacción con el usuario.

1.2 Funciones del sistema operativo



Un **sistema informático** puede ser dividido, *grosso modo*, en cuatro componentes: el hardware, los usuarios, los programas de aplicación y el sistema operativo.

- **Programas de aplicación**. El objetivo fundamental de un sistema informático es ejecutar programas para resolver los problemas informáticos de los usuarios. Con ese objetivo se construye su hardware y se desarrollan los programas de aplicación (hojas de cálculo, navegadores de Internet…) que usan los usuarios para resolver dichos problemas.

- **Hardware**. El hardware (CPU, memoria, dispositivos E/S…) proporcionan los recursos computacionales del sistema informático. Los programas de aplicación necesitan usar estos recursos para resolver los problemas de los usuarios.

- **Sistema operativo**. Las aplicaciones necesitan realizar operaciones comunes, como acceder a los dispositivos de E/S o reservar porciones de la memoria. Todas estas operaciones comunes están centralizadas en el sistema operativo.

El sistema operativo es el programa más íntimamente relacionado con el hardware, ya que gestiona los recursos hardware y software disponibles, los asigna a los diferentes programas, resuelve los conflictos en las peticiones y hace que el sistema opere eficientemente para resolver los problemas de los usuarios.
Además, es el encargado del control de la ejecución de los programas de los usuarios, por lo que tiene la tarea de prevenir errores y el uso inadecuado del ordenador.

El sistema operativo no hace un trabajo directamente útil para los usuarios, pero proporciona un entorno adecuado para que los programas de aplicación lo hagan. Es decir, los sistemas operativos existen porque es más sencillo crear sistemas informáticos útiles para los usuarios con ellos que sin ellos.

2. TIPOS DE SISTEMAS OPERATIVOS

2.1 Mainframes

Los ordenadores centrales o mainframes fueron utilizados en aplicaciones comerciales y científicas. Se caracterizan por su gran capacidad de memoria, almacenamiento secundario, gran cantidad de dispositivos de E/S, rapidez y fiabilidad. Hay varios tipos:

- **Sistemas de procesamientos por lotes**: Enormes máquinas operadas desde una consola y conectados a lectores de tarjetas perforadas. El trabajo era preparado por cada programador y entregado al operador del sistema, el cuál tenía acceso al sistema. Estos sistemas se convirtieron en sistemas de procesamiento por lotes pues su función era cargar y ejecutar sin interrupción un conjunto (lotes) de programas.
- **Sistemas multiprogramados**: La solución al inconveniente de los sistemas de procesamiento por lotes con la E/S fue que los programas no accedieran directamente al dispositivo de E/S, sino que, en su lugar, solicitaran la operación al monitor del sistema para que este la solicitara al hardware. Así el sistema operativo tiene la oportunidad de sustituir el programa en la CPU por otro, mientras la operación de E/S se completa. En los sistemas multiprogramados la ejecución de los trabajos funcionaban de la siguiente manera:
    1. En el disco magnético se almacenaba una cola donde se colocaban los trabajos.
    2. El SO cargaba varios trabajos en memoria en la cola del disco magnético.
    3. El SO cede la CPU a uno de los trabajos en memoria
    4. Cuando el trabajo de la CPU lo requería, le pedía al S.O usar la E/S. Cuando la E/S acababa, el programa que ocupaba la CPU no era interrumpido, sino que escogía un momento para ejecutarse en la CPU.
    5. Cuando un programa en la CPU terminaba, los recursos dejaban memoria libre y el SO escogía otro trabajo en la cola del disco magnético.
- **Sistemas de tiempo compartido:** Estos sistemas surgen debido a que un usuario introduce información de forma continua para luego detenerse durante largos periodos de tiempo, mientras que en un grupo de usuarios, las pausas de uno de ellos se pueden llenar con la actividad de los otros. Estos sistemas se caracterizan por:
    1. **Tener terminales**, es decir, hardware especializado para hacer de interfaz directa entre los usuarios y el sistema. Pudiendo haber múltiples usuarios al mismo tiempo.

2. **Usar la multiprogramación** para tener varios trabajos en la memoria principal al mismo tiempo.
3. **Repartir el tiempo de CPU entre usuarios**. El SO asignaba tiempo de CPU a cada usuario.

Estos sistemas significaron un salto importante en complejidad por:

- El sistema requería gestione de la memoria y procesos
- El sistema operativo debe utilizar técnicas de memoria virtual.
- Planificación de la CPU.
- Mecanismos de sincronización y comunicación.
- SO requería de un componente de gestión de discos.

2.2 Sistemas de escritorio:

Sobre los años 70 apareció la generación de **microcomputadoras**, que incluían periféricos de E/S, por lo que su uso fue más fácil, y de su mano llegaron los primeros **sistemas operativos de escritorio**.

Estos sistemas eran muy sencillos, carecían de permisos de usuario, protección, y no eran multiusuarios o multitarea. Con el paso de los años han adquirido características modernas hasta cierto punto que ya no son solo de escritorio (móviles, superordenadores).

Son muchos los ejemplos de sistemas de escritorio, desde el primero en 1977 CP/M hasta los actuales, como GNU/Linux, Windows, Apple macOS.

2.3 Sistemas de mano:

Hacemos referencia a tablets, smartphones, libros electrónicos con este término. Los desarrolladores deben enfrentarse a desafíos como la batería, el tamaño limitado. Se caracterizan por su facilidad de uso y rendimiento/tiempo de batería.

2.4 Sistema multiprocesador:

Es aquel ordenador donde hay procesadores interconectados que comparten el bus del sistema, el reloj, memoria y los periféricos. Ventajas de estos sistemas:

- **Aumentan la cantidad de trabajo realizado** dado a un mayor número de procesadores. No obstante esto puede ocasionar pérdida de rendimiento debido a la sincronización requerida para controlar el acceso a los recursos del hardware.
- Puede **costar menos** que múltiples sistemas monoprocesadores conectados para hacer un trabajo equivalente.
- **En caso de fallo** de un procesador, el sistema no se detendrá.

Google Gemini: Plan Pro a 0€ durante 1 año.

Tu ventaja por ser estudiante

Entra en wlh.es/estudiacongemini

Consigue la oferta

Sintetiza horas de investigación en minutos.

Oferta válida hasta el 9 de diciembre de 2025

Hay dos tipos de estos sistemas:

- **Sistemas de multiprocesamiento simétrico o SMP**: Todos los procesadores son iguales, comparten mismos recursos y cada uno ejecuta una copia del núcleo del sistema. El SO tiene que ser diseñado para repartir los recursos y trabajos entre los procesadores.
- **Sistemas de multiprocesamiento asimétrico o AMP**: Hay un procesador principal y varios secundarios a quienes el principal planifica y entrega las tareas.

2.5 Sistemas distribuidos:

Es un sistema usado en la actualidad para conectar ordenadores individuales mediante líneas de comunicación como Ethernet, líneas telefónicas o wifi. Se pueden clasificar en:

- **Sistemas cliente-servidor**: En estos sistemas existen ordenadores que actúan como servidores encargados de satisfacer las peticiones de los ordenadores clientes. Como ejemplo de este sistema son los servidores de bases de datos, que responden a las consultas SQL de los clientes o los servidores de archivos.
- **Sistemas de redes entre iguales**: también llamados P2P (peer-to-peer) clientes y servidores no se distinguen entre ellos. Todos los nodos son iguales y cada uno puede actuar como cliente y servidor. Ejemplos: BitTorrent y Bitcoin.
- **Sistemas operativos para sistemas distribuidos**: Se distinguen en:
    - **Sistemas operativos de red** que ofrecen a las aplicaciones que corren sobre ellos acceso a redes de ordenadores, por ejemplo algún mecanismo que permita a diferentes procesos de diferentes ordenadores enviar y recibir mensajes. Este tipo de sistemas son los más utilizados entre los distribuidos.
    - **Sistemas operativos distribuidos** que crean la ilusión de que está en un único ordenador, aunque en la realidad el SO controla todos los ordenadores de la red, provocando que el usuario no sepa donde se llevan a cabo los procesos o almacenamiento de sus archivos.

2.6 Sistemas en clúster:

Interconectan ordenadores individuales por medio de una red local y comparten almacenamiento. Se utilizan para:

- **Obtener servicios de alta disponibilidad** → Para ello un nodo clúster puede estar ejecutando un servicio mientras otro nodo lo monitoriza, así, en caso de fallo, el nodo que monitoriza podrá sustituirlo. Este sistema se puede multiplicar por dos o más nodos.

- **Computación de alto rendimiento o HPC** → Todos los nodos se utilizan para dar un mismo servicio, el que reparte el trabajo se llama nodo balanceador de carga. Usado para cálculos muy pesados, como simulaciones meteorológicas, servidores de internet con mucho tráfico, etc.

2.7 Sistemas de tiempo real:

Usados cuando hay requerimientos estrictos de tiempo en la ejecución de tareas o procesamiento de datos. Por ejemplo, se suelen usar en sistemas de control industrial, domótica, armamento, automoción (inyección de combustible, frenado, tracción) o dispositivos médicos. Se clasifican en:

- **Sistema de tiempo real estricto o hard real-time**: garantizan que las tareas están realizadas dentro de unos márgenes estrictos de tiempo. Suelen carecer de memoria virtual y de abstracciones que aíslen al desarrollador del funcionamiento del hardware, ya que produce impredecibilidad a retrasos en el funcionamiento de tareas delimitadas dentro de un rango de tiempo.
- **Sistema de tiempo real flexible o soft real-time**: en un sistema operativo hay tareas que tienen más importancia que otras, así que establecen prioridades. Usado para tareas como multimedia, realidad virtual, videojuegos y es compatible con los sistemas de escritorio. Estos sistemas se relacionan mucho con los sistemas empotrados
- **Sistemas empotrados**: Diseñados para tareas específicas, SSOO con características limitadas sin necesidad de interfaz usuario.

TEMA 2: ORGANIZACIÓN DE LOS SISTEMAS OPERATIVOS

1. COMPONENTES DEL SISTEMA

1.1 Gestión de procesos

Proceso → programa en ejecución.
El **proceso** es la unidad de trabajo en cualquier sistema operativo moderno. Es quién realiza las tareas que interesan a los usuarios. A cada proceso se le asigna un tiempo de CPU y el resto de recursos del sistema (memoria, archivos, dispositivos de E/S…).

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.

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

- Crear y terminar procesos.
- Suspender y reanudar procesos.
- Sincronización de procesos.
- Comunicación entre procesos.
- Tratamiento de interbloqueos.

1.2 Gestión de la memoria principal

La **memoria principal** es el único almacenamiento al que la CPU tiene acceso directo. Para que un programa pueda ser ejecutado, debe ser copiado en la memoria principal; y para que un proceso tenga acceso a datos almacenados en cualquier otro dispositivo de almacenamiento, primero deben ser copiados en la memoria principal.

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

- Controlar qué partes de la memoria están en uso y cuáles no.
- Decidir qué procesos (o partes de procesos) añadir o extraer de la memoria.
- Asignar y liberar espacio de la memoria principal según sea necesario.

1.3 Gestión del sistema de E/S

El sistema de E/S consta de:

- Un componente de **gestión de memoria especializado en E/S**, con soporte para servicios de *buffering, caching y spooling*. Estos servicios son utilizados por el resto del sistema de E/S.
- Una **interfaz E/S genérica** → Accede a cualquier dispositivo evitando las particularidades concretas del hardware instalado en cada ordenador.
- **Controladores de dispositivo** → Componente que realmente conoce las peculiaridades específicas del dispositivo. La interfaz de E/S genérica traslada las peticiones a estos controladores, que las convierten en acciones concretas sobre el hardware del dispositivo.

1.3.1 Buffering

El **buffering** es una estrategia que almacena los datos de manera temporal en una zona de la memoria, llamada **búfer**, de la siguiente manera:

1. El controlador indica a un dispositivo que escriba los bloques de datos solicitados en un búfer.
2. Cuando la escritura del búfer se ha completado, se transfiere su contenido al proceso que hizo la solicitud para que procese los datos. Mientras lo hace, el controlador indica al dispositivo que copie nuevos datos en el búfer.

1.3.2 Caching

En el **caching** el sistema mantiene en la memoria principal una copia de datos leídos o escritos recientemente en los dispositivos de E/S del sistema (discos duros, memorias USB). Esto mejora la eficiencia del sistema si se accede con frecuencia a los mismos datos, puesto que el acceso a la memoria principal es más rápido que a los dispositivos de E/S. La memoria principal es de tamaño limitado, por lo que solo se mantiene copia de los datos utilizados con *mayor frecuencia*.

1.3.3 Spooling

El **spooling** se utiliza en dispositivos que no admiten el acceso simultáneo de varias aplicaciones a la vez, como las impresoras o unidades de cinta.

Cuando varias aplicaciones intentan enviar un trabajo a una impresora, el sistema operativo lo intercepta para copiar los datos enviados a un archivo independiente. Al finalizar el envío, el archivo se mete en una cola, de donde son extraídos los trabajos para su impresión uno a uno. Esto evita el acceso simultáneo al dispositivo por parte de varios procesos, mientras que estos pueden entregar el trabajo y continuar con sus tareas sin esperar a que la impresora esté disponible.

1.4 Gestión del almacenamiento secundario

Dentro de los dispositivos de E/S, los dedicados al almacenamiento secundario (discos duros, memorias USB…), merecen un tratamiento especial.
Los programas que se desean ejecutar deben estar en la memoria principal, pero esta es demasiado pequeña para alojar todos los datos y programas del sistema, y podrían perderse en caso de que ocurriera un fallo de alimentación. Por eso los ordenadores disponen de un almacenamiento secundario, para guardar datos de forma masiva y permanente.

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.

1.5 Gestión del sistema de archivos

Archivo / **Fichero** → colección de datos relacionados, identificados por un nombre, que es tratada por el sistema operativo como la unidad de información en el almacenamiento secundario.

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.
- Soporta 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.

1.6 Gestión de red

El **componente de red** se responsabiliza de la comunicación con otros sistemas interconectados mediante una red de ordenadores (ej: en Internet o en la red de área local de una oficina).

1.7 Protección y seguridad

Protección → Se centra en las amenazas internas en un sistema informático

- Mecanismos necesarios cuando un sistema informático tiene múltiples usuarios y permite la ejecución concurrente de varios procesos, pues así solo pueden utilizar los recursos aquellos procesos que hayan obtenido la autorización del sistema operativo. (ej: el temporizador, que garantiza que ningún proceso toma el control de la CPU por tiempo indefinido.)
- Mejora la fiabilidad, al permitir detectar los elementos del sistema que no operan correctamente. Un recurso desprotegido no puede defenderse contra el mal uso de un usuario no autorizado o incompetente.

Seguridad → Se centra en las amenazas externas a un sistema informático.

- Un sistema puede tener la protección adecuada pero estar expuesto a fallos y permitir accesos inapropiados. Por eso es necesario disponer de mecanismos de seguridad encargados de defender el sistema frente a ataques internos y externos. Eso incluye a virus y gusanos, ataques de denegación de servicio, robo de identidad y uso no autorizado del sistema, etc.
-

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

2.1 Servicios que garantizan el funcionamiento eficiente del sistema

- **Asignación de recursos.** Cuando hay múltiples usuarios o múltiples trabajos ejecutándose los recursos deben ser asignados a cada uno de ellos (ej: la CPU, asignada por el planificador de la CPU del gestor de procesos; la memoria principal, asignada por el gestor de memoria). Esta asignación debe hacerse con el fin de garantizar la máxima eficacia del sistema.

- **Monitorización.** Seguimiento de los recursos que los usuarios usan y en qué cantidad. Útil para facturar a los usuarios por el uso de los recursos(ej: facturar por el tiempo de CPU) para configurar el sistema mejorando el rendimiento o para limitar cuánto de cada recurso puede usar cada usuario como máximo.

- **Protección y seguridad.** La protección implica asegurar que el acceso a los recursos del sistema está controlado. Por ejemplo, que la información almacenada en un sistema multiusuario solo pueda ser accedida por su propietario o que un proceso no pueda interferir con otro o con el sistema operativo. La seguridad del sistema respecto a los agentes exteriores también es importante. Empieza obligando a los usuarios a autenticarse en él para obtener acceso a los recursos del mismo, pero incluye defender de intentos de acceso inválidos a través de la red.

2.2 Servicios útiles para el usuario

- **Interfaz de usuario.** Los S.O diseñados para que los usuarios interactúen con ellos deben proporcionar una interfaz de usuario adecuada, que puede tener diferentes formas según el propósito del sistema.

- **Operaciones de E/S.** Un programa puede necesitar realizar operaciones de E/S que involucren archivos o dispositivos de E/S. Por eficiencia y protección del usuario, normalmente los procesos no tienen acceso directo a los dispositivos, por lo que el S.O debe proporcionar medios para solicitar estas operaciones a los componentes correspondientes del sistema operativo.

- **Manipulación de sistemas de archivos.** Los programas necesitan leer y escribir archivos y directorios, crearlos y borrarlos por nombre, buscar un archivo dado y listar información acerca del mismo.

- **Comunicaciones.** Los procesos necesitan poder intercambiar información entre ellos, tanto si se ejecutan en el mismo ordenador, como en diferentes equipos unidos por una red.

- **Detección de errores.** El sistema operativo necesita tener conocimiento de los posibles errores y tomar la acción apropiada para asegurar una computación consistente y segura.
  (ej: errores del hardware(fallos de energía o errores en la memoria); en la E/S(errores de paridad o falta de papel en la impresora) y en los programas de usuario(desbordamientos aritméticos o accesos ilegales a la memoria))

2.3 Interfaz de usuario

La interfaz de usuario es un servicio fundamental para todos los sistemas, diseñados para la interacción directa con los usuarios. Existen varios tipos de interfaces:

- **Interfaz de línea de comandos / intérprete de comandos.** Permite que los usuarios introduzcan directamente los comandos que el sistema operativo debe ejecutar. En algunos sistemas este tipo de interfaz se incluye dentro del núcleo, pero generalmente se trata de un programa especial denominado shell que se ejecuta cuando un usuario inicia una sesión.

- **Interfaz de proceso por lotes.** Los comandos y directivas para controlar dichos comandos se listan en archivos que posteriormente pueden ser ejecutados. Este tipo de interfaz es la utilizada en sistemas no interactivos, como los antiguos sistemas de procesamiento por lotes y los sistemas multiprogramados. También suele estar disponible en los sistemas de tiempo compartido y en los sistemas de escritorio modernos, junto con algún otro tipo de interfaz de usuario.
  (ej: la shell de los sistemas UNIX permite indicar comandos uno a uno de forma interactiva, pero también permite usar scripts, un archivo con una lista de órdenes para que se ejecuten automáticamente de principio a fin.)

- **Interfaz gráfica de usuario(GUI).** Permite a los usuarios utilizar un sistema de ventanas y menús controlables mediante el ratón.
-

3. INTERFAZ DE PROGRAMACIÓN DE APLICACIONES

3.1 Interfaces de programación de aplicaciones

- **Windows API** → Interfaz de programación de aplicaciones de Windows Provee un conjunto amplio de servicios: E/S a archivos y dispositivos, gestión de procesos, hilos y memoria, manejo de errores, registro de Windows, interfaz a dispositivos gráficos, gestión de ventanas, comunicaciones en red, etc.

- **POSIX** → (Portable Operating System Interface for Unix) es una familia de estándares que definen una interfaz de programación de aplicaciones para S.O. Esto permite que un mismo programa pueda ser ejecutado en distintos sistemas operativos, siempre que sean compatibles con POSIX. El estándar POSIX define una API común para todos los UNIX y sistemas estilo UNIX modernos.

3.2 Llamadas al sistema

Para que un proceso pueda invocar los servicios del S.O que necesita, hace falta un procedimiento denominado **llamada al sistema.**

· **Invocar llamadas al sistema**

Llamada al sistema → instrucción específica en lenguaje ensamblador que genera una excepción (interrupción lanzada por la CPU al detectar instrucciones especiales o un error al ejecutar una instrucción).

Ej: (MIPS e Intel x86) *syscall* → lanza una excepción y la CPU salta a una rutina en el código del núcleo del sistema, deteniendo así la ejecución del proceso que la invocó.

Al realizar una llamada, el sistema debe saber qué operación le está pidiendo el proceso. Para ello se pone un número identificativo de la llamada en un registro concreto de la CPU.

nº para identificar cada llamada al sistema → depende del S.O

registro donde se guarda, la instrucción utilizada y detalles sobre cómo realizar la llamada → depende también de la arquitectura de la CPU

· **Paso de argumentos**

Una llamada al sistema suele requerir más información que la identidad de la llamada(Si se quiere abrir un archivo, requiere del nombre del archivo, y si se abre para leer o escribir).

Hay 3 métodos para pasarle parámetros adicionales al identificador de la llamada:

- **Registros de la CPU:** Carga los parámetros en los registros de la CPU antes de realizar la llamada (+*eficiente* pero limita el nº de parámetros al nº de registros)
- **Tabla en memoria:** Copia los parámetros en una tabla en la memoria principal y guarda la dirección la tabla en un registro específico de la CPU antes de realizar la llamada (no limita el nº de parámetros)
- **Pila del proceso:** se insertan los parámetros en la pila del proceso y el sistema operativo los recupera de allí durante la llamada al sistema (tampoco limita el nº de parámetros)

3.3 Librería del sistema

Como las llamadas al sistema se hacen en lenguaje ensamblador, no son cómodas de usar. Por ello, los programas llaman a las funciones de la librería del sistema, que son las encargadas de hacer las llamadas al sistema.

Librería del sistema → Colección de clases o funciones que ofrecen los servicios del sistema operativo a los programas, apoyándose en las llamadas al sistema (es parte del S.O).

- Algunas funciones son traducciones literales de llamadas al sistema (ej: write() / open()) mientras que otras son más complejas, trabajan más o muestran conceptos más abstractos que las llamadas al sistema.
- Constituye la interfaz de programación de aplicaciones del sistema operativo. Es la forma recomendada de solicitar servicios al sistema operativo. Invocar directamente las llamadas al sistema debe ser el último recurso.
- La invocación de las funciones de la librería del sistema se realiza como si fueran cualquier otra función del programa.
- Suele estar implementada en C, lo que permite que tanto los programas en **C** como en **C++** la puedan utilizar directamente.

3.4 Librería estándar

Lenguajes distintos de **C** y **C++** pueden tener difícil usar las funciones de la librería del sistema. Por eso, junto al compilador de cada lenguaje de programación suele ir una **librería estándar** que ofrece clases o funciones con las que los programas pueden acceder a los servicios del S.O y realizar las tareas de forma más sencilla. Generalmente no forman parte del sistema operativo, sino de las herramientas de desarrollo de cada lenguaje de programación, y constituyen la interfaz de programación de aplicaciones del lenguaje al que acompañan.
La librería estándar utiliza la librería del sistema para acceder a los servicios del S.O, que a su vez realiza las llamadas al sistema necesarias.

4. OPERACIÓN DEL SISTEMA OPERATIVO

El S.O y los procesos de usuarios comparten los recursos del sistema, por lo que es necesario que un error en un programa solo afecte al proceso que lo ejecuta → Se deben establecer mecanismos de protección frente a los errores en los programas que se ejecutan en el sistema.





4.1 Software controlado mediante interrupciones

Los sistemas operativos modernos pertenecen a un tipo de software controlado mediante interrupciones → Los sucesos que requieren la atención del sistema se indican mediante una interrupción:

- Cuando un proceso comete un error o un programa solicita un servicio al S.O a través de una llamada al sistema lo que se genera es una excepción. Esto despierta al S.O para que haga lo que sea más conveniente.
- Cuando un proceso necesita un servicio lanza una llamada al sistema (ejecuta una instrucción que lanza una excepción). Esta excepción despierta al S.O para que atienda la petición.
- Cuando los dispositivos de E/S requieren la atención del S.O (ej: se ha completado una transferencia de datos) se genera una interrupción que despierta al sistema operativo.

El S.O configura la CPU durante el arranque para que si ocurre cualquier interrupción o excepción la ejecución, salte a rutinas en el código del núcleo, para darles el tratamiento adecuado. Si no se realiza ninguna de las anteriores, el S.O permanece inactivo esperando a que algo ocurra.

4.2 Operación en modo dual

Para proteger el sistema de programas con errores es necesario distinguir entre la ejecución del código del S.O y del código de los programas de usuario, de tal forma que el código de los programas de usuario esté más limitado que el del S.O.

El método más utilizado es usar un soporte en la CPU que permita diferenciar entre varios modos de ejecución y restringir la utilización de las instrucciones peligrosas(**instrucciones privilegiadas**) para que solo puedan ser utilizadas en el modo en el que se ejecuta el código del S.O.

Modos de operación:

- Modo usuario → ejecuta el código de los procesos de los usuarios
  Si se intenta ejecutar una instrucción privilegiada, se genera una excepción en vez de ejecutar la instrucción.
- Modo privilegiado/supervisor/kernel → ejecuta el código de las tareas del S.O
  Las instrucciones privilegiadas sólo pueden ser ejecutadas en este modo.

Las instrucciones privilegiadas suelen ser:

- Conmutar al modo usuario desde el modo privilegiado.
- Acceder a dispositivos de E/S.
- Gestión de las interrupciones (desactivarlas, activarlas y configurarlas).

Ejecución de instrucciones:

1. Al encender el ordenador, la CPU se inicia en el modo privilegiado → se carga el núcleo del S.O e inicia su ejecución.
2. Se debe cambiar al modo usuario antes de ceder la CPU a un proceso de usuario → asegura que el código de los procesos de usuario siempre se ejecuten en modo usuario, con menos privilegios.
3. La CPU vuelve al modo privilegiado cuando ocurre una interrupción o una excepción.

4.3 Protección de la memoria

La memoria principal se divide en 2 partes:

- **Espacio del núcleo** → alberga el núcleo del sistema operativo
  El S.O puede estar localizado en la parte baja o alta de la memoria. El determinante en la elección es la localización del vector de interrupciones (tabla en la memoria que define las direcciones a las que saltará la CPU en caso de interrupción o excepción).
  En la mayor parte de las arquitecturas este reside en la parte baja de la memoria, por lo que el S.O también se aloja en la parte baja.

- **Espacio de usuario** → alberga los procesos de usuario
  Los procesos no tienen acceso libre a toda memoria física, para proteger a los procesos en ejecución y al S.O de posibles errores en cualquiera de ellos.
  El S.O proporciona a cada proceso un **espacio de direcciones virtual** (conjunto de todas las direcciones que puede generar la CPU para un proceso).
  En los accesos a la memoria principal durante la ejecución del proceso, estas direcciones virtuales son convertidas en direcciones físicas, antes de ser enviadas a la memoria principal.
  Direcciones físicas → direcciones reales que ve la memoria.
  Espacio de direcciones físico → conjunto de direcciones todas las direcciones físicas.

Ventajas:

- Permite el aislamiento de los procesos → cada uno obtiene "toda la memoria" para sí mismo → evita que un proceso acceda a la memoria de otro proceso.
- Evita que el código ejecutado en modo usuario acceda a zonas de memoria con instrucciones privilegiadas.

4.4 El temporizador

El temporizador se utiliza para asegurar que ningún proceso acapara la CPU indefinidamente (ej: un programa que entra en un bucle infinito). Este es configurado para interrumpir a la CPU en intervalos regulares, transfiriendo el control al núcleo del sistema, con lo que se consigue:

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

Las únicas instrucciones que pueden modificar el contenido del temporizador son las instrucciones privilegiadas.

4.5 Máquinas virtuales

Una máquina virtual también es un proceso en un sistema operativo de una máquina real. Se utilizan las mismas técnicas para crear la ilusión de que se ejecuta en su propia máquina. Sin embargo, en lugar de llamadas al sistema, el software que gestiona la máquina virtual ofrece una interfaz de hardware virtual. Es decir:

1. El sistema operativo de la máquina virtual intenta acceder al hardware, ya que presupone que se ejecuta en una máquina real.
2. El sistema operativo anfitrión intercepta estos intentos y, en lugar de detener el proceso, comunica el suceso al software de gestión de la máquina virtual.
3. El software de gestión de la máquina virtual identifica a qué dispositivo y que intenta hacer el sistema operativo de la máquina virtual en él y lo transforma en peticiones al sistema operativo anfitrión.

Problema → La máquina virtual es menos eficiente que la física, ya que necesita que se simule el hardware virtual mediante software, para luego traducir las peticiones realizadas en peticiones al sistema operativo del sistema anfitrión.
Solución → Paravirtualización → Se instalan en la máquina virtual unos controladores de dispositivos que trasladan las peticiones que llegan del S.O en la máquina virtual directamente al S.O anfitrión, sin necesidad de utilizar el hardware simulado.

4.6 Arranque del sistema

Desde que el sistema se pone en marcha hasta que el S.O inicia su ejecución, ocurren una serie de pasos:

1. Se envía una señal de **reset** a la CPU motivada por el encendido/reinicio del sistema.
2. La CPU inicializa el contador de programa a una dirección predefinida de la memoria. En esa dirección está el **bootstrap** inicial.

Bootstrap → es el programa que se encarga en primera instancia del arranque. Es almacenado en una memoria no volátil(ROM o Flash), ya que la RAM está en un estado indeterminado en el momento del arranque. El bootstrap forma parte del firmware de las placas madres. Sus tareas son:

1. **Diagnóstico de la máquina** → El bootstrap se detiene en este punto si el sistema no supera el diagnóstico.
2. **Inicializar el sistema** → Configurar los registros de la CPU, inicializar los dispositivos y contenido de la memoria, etc.
3. **Iniciar el S.O**.

5. SISTEMAS OPERATIVOS POR SU ESTRUCTURA

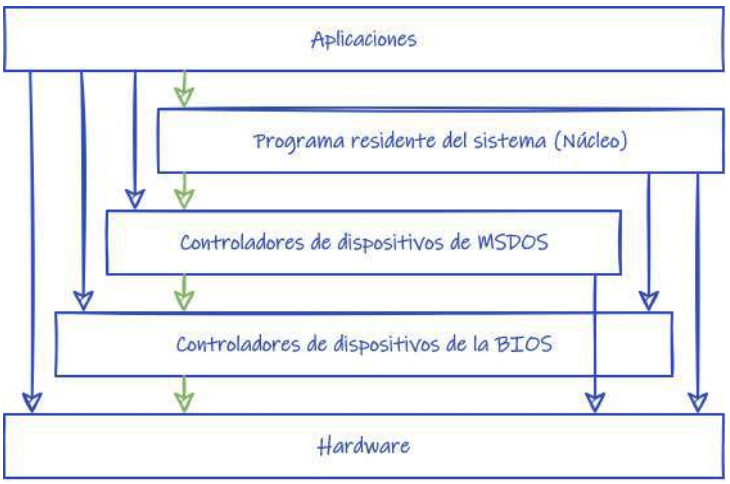
5.1 Estructura sencilla

Los S.O de 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.

Ejemplos:

- **MSDOS**
  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.

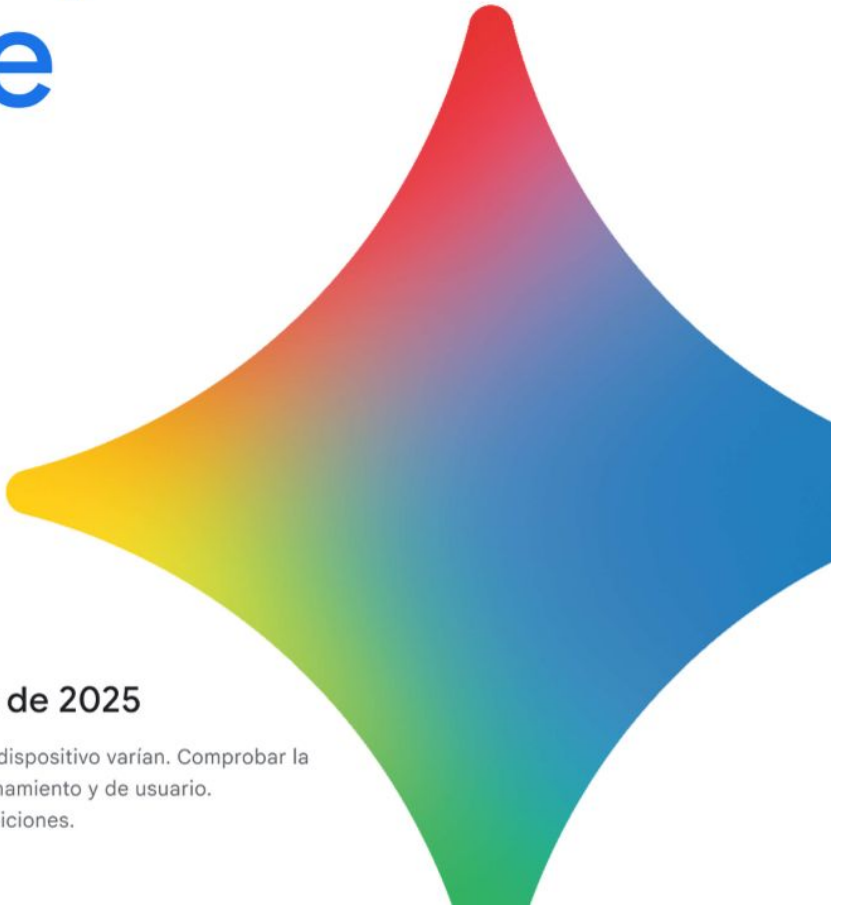


Google Gemini: Plan Pro a 0€ durante 1 año.

Tu ventaja por ser estudiante

Entra en wlh.es/estudiacongemini

Consigue la oferta



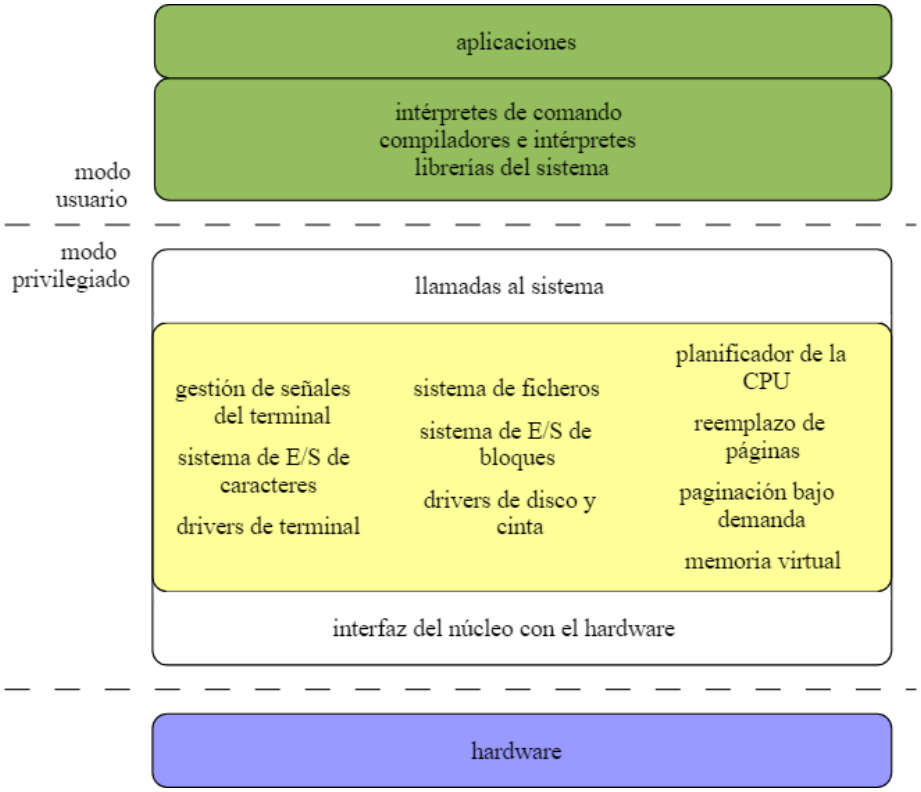
Oferta válida hasta el 9 de diciembre de 2025

Después, 21,99€/mes. 18+. Los resultados/la compatibilidad del dispositivo varían. Comprobar la exactitud de las respuestas. Se aplican restricciones de almacenamiento y de usuario. Se requiere una cuenta de Google. Consulta los términos y condiciones.

Como el Intel 8086 para el que fue escrito MS-DOS no proporcionaba un modo dual de operación, los diseñadores del sistema no tuvieron más opción que dejar accesible el hardware a los programas de usuario.

- **UNIX**

  Otro ejemplo es el de UNIX original, donde sí había una separación clara entre procesos de usuario y código del sistema, pero juntaba un montón de 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 gran cantidad de funcionalidades, que es difícil de implementar y mantener si no se compartimenta adecuadamente.

5.2 Estructura en capas

Características:

- **Funcionalidad dividida en capas** → Una capa solo utiliza funciones y servicios de la capa inmediatamente inferior a través de una interfaz bien definida.
- Cada capa **oculta los detalles de su implementación** a la capa superior (ej: las estructuras de datos internas o el hardware de la capa inferior que utiliza).
- Las capas hacen que el **código** esté **mejor compartimentado** → Mayor escalabilidad que los S.O de estructura sencilla.
  Ej: al corregir un bug o añadir una nueva funcionalidad solo hay que preocuparse de su efecto en la capa a la que afecta y no en todo el código del núcleo (siempre que no se altere la interfaz de la capa con el exterior).
- **Menos eficientes** que la de los sistemas de **estructura sencilla** → Los argumentos son transformados en cada capa y los datos necesarios deben de ser transferidos al invocar operaciones en la capa inferior → cada una añade cierto nivel de sobrecarga al funcionamiento del sistema.
- También **monolíticos** → Gran parte de la funcionalidad del sistema se implementa en el núcleo, aunque esté compartimentado en capas.

Dificultades con el diseño:

- Puede ocurrir que 2 componentes que se encuentren uno encima del otro, necesiten el uno al otro y viceversa, lo que no es posible debido a que deberían turnarse la capa superior para poder funcionar. Esto se soluciona colocando ambos componentes en la misma capa.

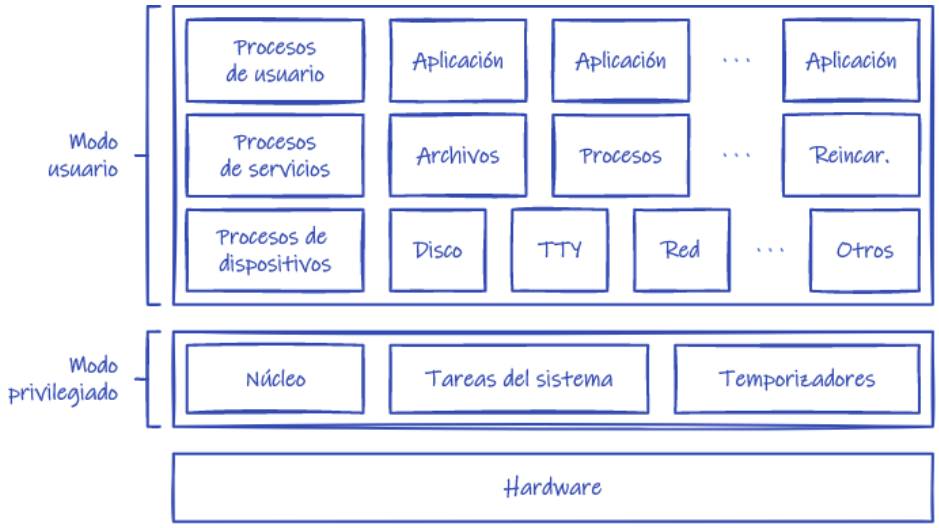
Al final, la solución de compromiso es tender hacia sistemas con muy pocas capas donde cada una tiene mucha funcionalidad, pero esto limita mucho las ventajas de esta técnica, ya que no permite compartimentar el núcleo tanto como sería deseable.

5.3 Microkernel

Características:

- Elimina todos los componentes no esenciales del núcleo y los implementa como procesos de usuario.
- Proporciona funciones mínimas de gestión de procesos y de memoria y algún mecanismo de comunicación entre procesos. Algunos microkernel reales también 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.

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.



Beneficios:

- **Facilidad al 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.
- **Facilidad para llevar el sistema a otras plataformas** → Como el núcleo es muy pequeño, resulta muy sencillo de portar a otras plataformas.
- **Más seguridad y fiabilidad** → Como la mayoría de los servicios se ejecutan como procesos de usuario separados, si un servicio falla no puede afectar a otros ni puede ser utilizado para ganar acceso a otros servicios o al núcleo.

Desventajas:

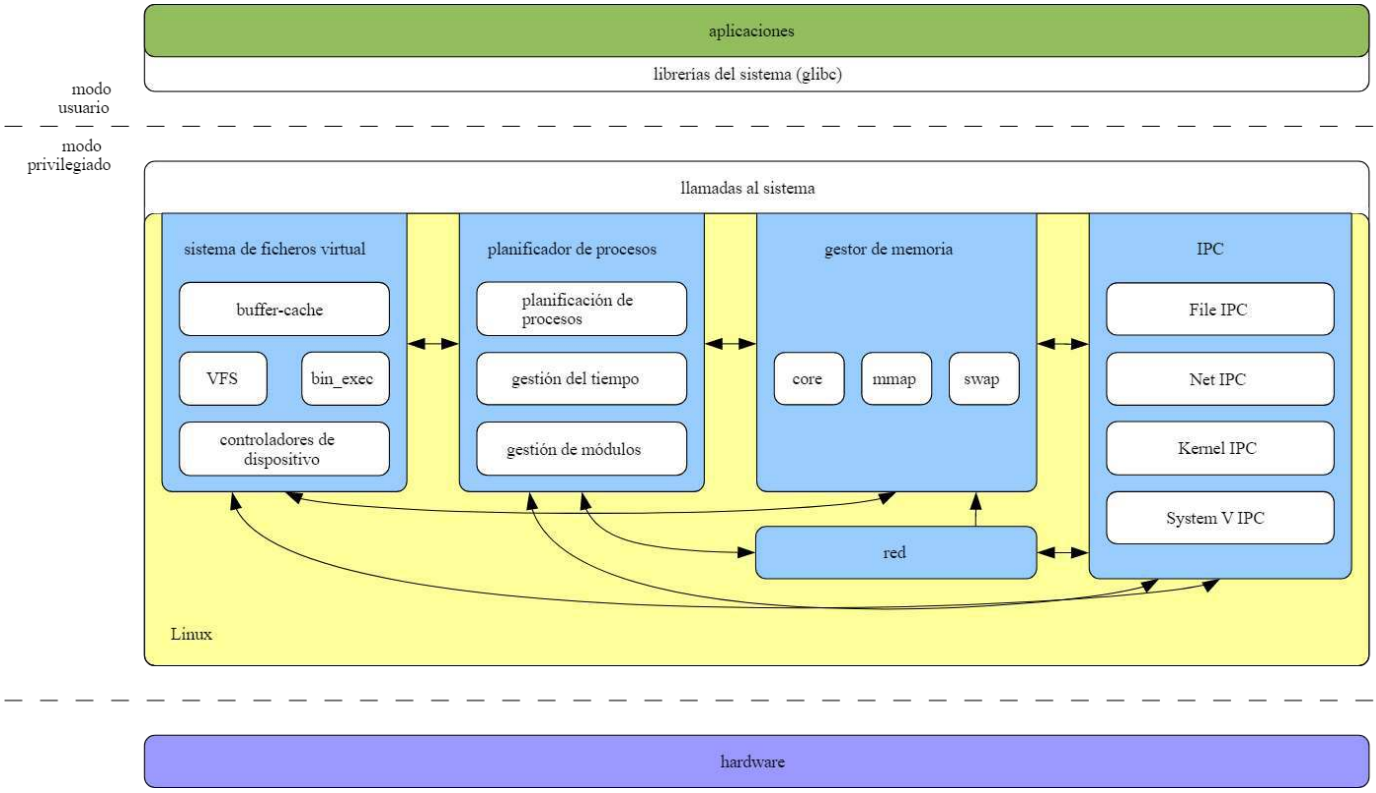
- **Menor rendimiento** → El mecanismo de comunicación añade una sobrecarga al sistema, como ocurre en la estructura por capas.

5.4 Estructura modular

Características:

- **Divide el núcleo en módulos**. Cada núcleo implementa funciones y servicios concretos y se comunican entre sí a través de una interfaz bien definida.
- **Cada módulo oculta** al resto los **detalles** de su **implementación**.
- **Todos** los módulos **pueden llamar** a funciones de la interfaz de **cualquier otro módulo**(en la estructura en capas 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 esté compartimentado en módulos.

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.



En este aspecto 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 su funcionalidad se ofrece desde el modo privilegiado. Un error en cualquier componente puede comprometer o hacer caer el sistema.

TEMA 3: GESTIÓN DE PROCESOS

1. PROCESOS

Tipos:

- **Procesos del sistema** → Ejecutan el código del S.O contenido en los programas del sistema, que sirven para hacer tareas del sistema operativo que es mejor mantener fuera del núcleo.
- **Procesos de usuario** → Ejecutan el código contenido en los programas de aplicación.

1.1 El proceso

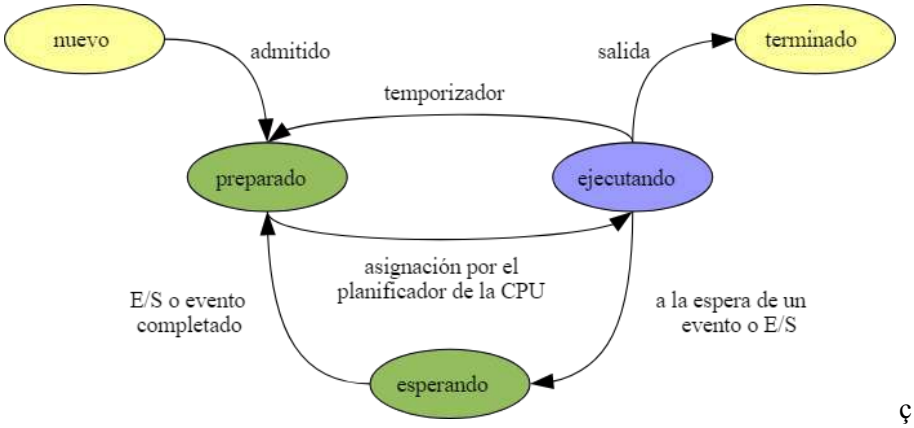
Como ya hemos visto, un proceso es un programa en ejecución, pero no solo están compuestos por el código del programa, sino que también son importantes otros elementos:

- **Segmento de código** → Contiene las instrucciones ejecutables del programa. También es conocido como segmento text ó .text.
- **Segmento de datos** → Contiene las variables globales y estáticas del programa que se inicializan con un valor predefinido. También es conocido como segmento .data.
- **Segmento BSS** → block started by symbol → contiene las variables globales y estáticas del programa inicializadas a 0 o sin inicialización explícita. Como contiene variables globales sin valor inicial, en el ejecutable solo se guarda la longitud que debe tener este segmento en la memoria. También es conocido como segmento .bss.
- **Pila** → Contiene datos temporales, como parámetros y direcciones de retorno de las funciones y variables locales.
- **Montón** → Contiene el espacio de la memoria que se asigna dinámicamente durante la ejecución del proceso. También es conocido como heap.
- **Información sobre el estado actual de ejecución** → Como el contador de programa, los valores de los registros de la CPU, el estado del proceso, etc.

Los segmentos de código, datos y BSS por lo general son secciones dentro del archivo ejecutable que contiene el programa. El resto de elementos los crea el sistema operativo al cargar el programa y crear el proceso.

1.2 Estados de los procesos

Cada proceso tiene un estado que cambia a lo largo de su ejecución y que está definido, parcialmente, por la actividad que realiza actualmente el propio proceso.



ç

Los estados por los que puede variar un proceso suelen ser:

- **Nuevo** → Proceso en proceso de creación. La creación de un proceso no es algo instantáneo, ya que necesita de varias operaciones que pueden tardar tiempo en realizarse, como: reservar memoria libre, cargar el programa en la memoria, inicializar estructuras de datos y configurar el entorno de ejecución.
- **Ejecutando** → El proceso está siendo ejecutado en la CPU. Tiene que haber sido escogido por el planificador de la CPU de entre todos los procesos en estado preparado. Solo hay un proceso en este estado por CPU en el sistema.
- **Esperando** → El proceso está esperando por algún evento. Ej: que termine una operación de E/S solicitada previamente o que otro proceso termine su ejecución. Múltiples procesos pueden estar en este estado de espera.
- **Preparado** → El proceso está esperando a poder usar la CPU. Múltiples procesos pueden estar en este estado.
- **Terminado** → El proceso ha finalizado su ejecución y espera a que el sistema operativo recupere los recursos que le fueron asignados. Como en el caso del estado nuevo, terminar un proceso tampoco es algo instantáneo.

1.3 Bloque de control de proceso (PCB)

Es una estructura de datos que representa a cada proceso en el S.O y que guarda información sobre su estado de actividad actual.
Sistema → Un PCB por proceso. Sirve de almacén para cualquier información que puede variar de un proceso a otro:

- **Estado del proceso**. El estado actual del proceso (nuevo, esperando, etc).

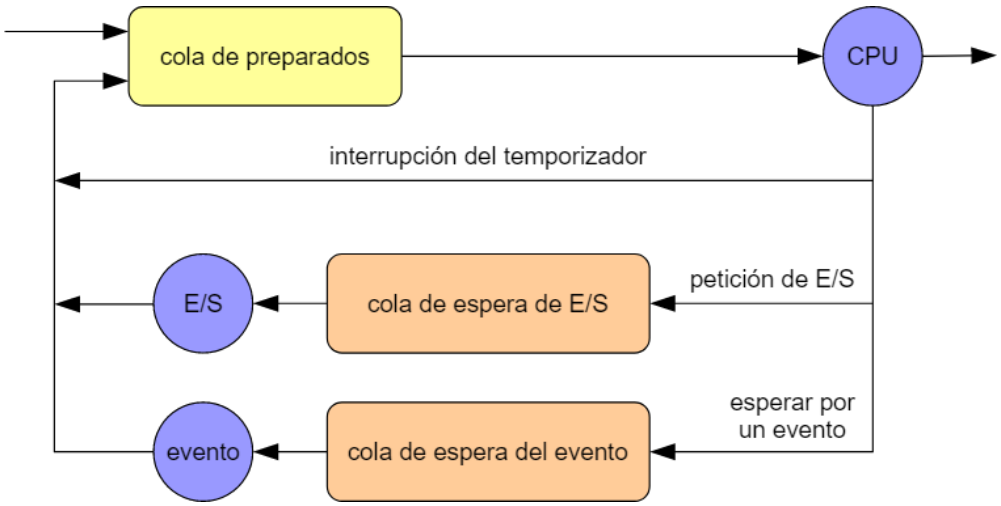
- **Contador de programa / PC** → Dirección de la próxima instrucción del proceso que debe ser ejecutada por la CPU. Durante el estado ejecutando el PC está en el registro correspondiente de la CPU. Su valor se guarda en el PCB al salir el proceso de la CPU para que comience a ejecutarse en ella otro proceso.
- **Registros de la CPU** → Como en el caso del PC, durante el estado ejecutando los valores están en los registros de la CPU, pero se guardan en el PCB cuando el proceso sale de la CPU para que se ejecute otro proceso.
- **Información de planificación de la CPU** → Incluye la info. requerida por el planificador de la CPU (La prioridad del proceso, punteros a las colas de planificación donde está el proceso, etc)
- **Información de gestión de la memoria** → Incluye la info. requerida para la gestión de la memoria. Ej, al usar:
  Asignación contigua de memoria → valores de los registros base y límite que definen el área de la memoria física que ocupa el proceso en el caso de se use.
  Paginación → la dirección a la tabla de páginas
- **Información de registro** → Cantidad de CPU usada, límites de tiempo en el uso de la CPU, estadísticas de la cuenta del usuario a la que pertenece el proceso, estadísticas de la ejecución del proceso, etc.
- **Información de estado de la E/S** → Lista de dispositivos de E/S reservados por el proceso, la lista de archivos abiertos, etc.

1.4 Colas de planificación

Hay diferentes colas de planificación para los procesos en distintos estados:

- **Cola de trabajo** → Contiene todos los trabajos del sistema, de manera que cuando entran en el sistema van a esta cola, a la espera de ser escogidos para ser cargados en la memoria y ejecutados (no existe en los sistemas modernos, solo en los multiprogramados).
- **Cola de preparados** → Contiene los procesos en estado preparado.(procesos cargados en la memoria principal que esperan para usar la CPU). La cola de preparados es generalmente una lista enlazada de PCB, donde cada uno incluye un puntero al PCB del siguiente proceso en la cola.
- **Colas de espera** → Contienen los procesos en estado esperando (que esperan por un evento concreto, como por ejemplo la finalización de una petición de E/S). Estas colas también suelen ser implementadas como listas enlazadas de PCB y suele haber una por evento, de manera que cuando ocurre algún evento todos los procesos en la cola asociada pasan automáticamente al estado preparado y a la cola de preparados.
- **Colas de dispositivo** → Son un caso particular de cola de espera. Cada dispositivo de E/S tiene asociada una cola de dispositivo que contiene los procesos que están esperando por ese dispositivo en particular.

Representación de la planificación de procesos a través de un diagrama de colas:



Analizándolo podemos tener una idea clara del flujo típico de los procesos dentro del sistema:

1. Un nuevo proceso llega al sistema. Una vez pasa del estado nuevo a preparado es colocado en la cola de preparados. Allí espera hasta que es seleccionado por el planificado de la CPU para su ejecución y se le asigna la CPU. Mientras se ejecuta pueden ocurrir varias cosas.
    - El proceso solicita una operación de E/S por lo que abandona la CPU y es colocado en la cola de dispositivo correspondiente en estado esperando. No debemos olvidar que aunque en nuestro diagrama no exista más que una de estas colas, en un sistema operativo real suele haber una para cada dispositivo.
    - El proceso puede querer esperar por un evento. Por ejemplo, puede crear otro proceso y esperar a que termine. En ese caso el proceso hijo es creado, mientras el proceso padre abandona la CPU y es colocado en una cola de espera en estado esperando hasta que el proceso hijo termine. La terminación del proceso hijo es el evento que espera el proceso padre para salir de la cola de espera y entrar en la cola de preparados para continuar su ejecución en la CPU cuando sea posible.
    - El proceso puede ser sacado forzosamente de la CPU, como resultado de la interrupción del temporizador, que permite determinar cuando un proceso lleva demasiado tiempo ejecutándose, así que es colocado en la cola de preparados en estado preparado.

2. Cuando las esperas concluyen, los procesos vuelven a la cola de preparado, pasando del estado de espera al de preparado.

3. Los procesos repiten este ciclo hasta que terminan. En ese momento son eliminados de todas las colas mientras el PCB y los recursos asignados son recuperados por parte del sistema operativo para poder usarlos con otros procesos.

1.5 Planificación de procesos

Durante su ejecución, los procesos se mueven entre las diversas colas de planificación a criterio del S.O. Este proceso de selección debe ser realizado por el planificador adecuado:

- **Planificador de largo plazo / planificador de trabajos** → Selecciona los trabajos desde la cola de trabajos en el almacenamiento secundario y los carga en memoria. Se usaba en los sistemas multiprogramados. Los sistemas de tiempo compartido posteriores y los sistemas modernos, carecen de planificador de trabajos, porque los programas se cargan directamente en memoria para ser ejecutados, cuando el usuario lo solicita.
- **Planificador de corto plazo o planificador de CPU** → Selecciona uno de los procesos en la cola de preparados y lo asigna a la CPU. Este planificador es invocado cuando un proceso en ejecución abandona la CPU, dejándola disponible para otro proceso.
- **Planificador de medio plazo** → Era utilizado en algunos sistemas para sacar procesos de la memoria cuando escasea y reintroducirlos posteriormente cuando vuelve a haber suficiente memoria libre. A este esquema se le denomina intercambio —o *swapping*. Útil en sistemas antiguos donde un proceso tenía que estar cargado completamente en la memoria para poder ejecutarse. En los sistemas actuales no es utilizado.

1.6 Cambio de contexto

El cambio de contexto es la tarea de asignar la CPU a un proceso distinto al que la tiene asignada en el momento actual. Esto implica salvar el estado del viejo proceso en su PCB y cargar en la CPU el estado del nuevo. Entre la información que debe ser preservada en el PCB se incluyen:

- El **contador de programa**.
- Los **registros de la CPU**.
- El **estado del proceso**.
- La **información de gestión de la memoria**. Por ejemplo, la información necesaria para configurar el espacio de direcciones del proceso.

El cambio de contexto es sobrecarga pura → poco eficiente (Algunas CPU disponen de instrucciones especiales para salvar y cargar todos los registros +eficientemente).

1.7 Operaciones sobre los procesos

1.7.1 Creación de procesos

Un proceso (padre) puede crear múltiples procesos (hijos) utilizando una llamada al sistema específica para la creación de procesos. Cada proceso creado se identifica de manera unívoca mediante un identificador de proceso o PID (suele ser un nº entero).

Árbol de procesos → Como cada nuevo proceso puede a su vez crear otros procesos, al final se acaba obteniendo un árbol de procesos.
Se conoce como init al proceso padre raíz de todos los procesos de usuario. Su PID siempre es 1, por lo tanto, es el responsable de crear todos los procesos que son necesarios para el funcionamiento del sistema.

Cómo obtienen los procesos hilos los recursos que necesitan → Existen dos alternativas:

- Cada proceso hijo puede solicitar y obtener los recursos directamente del S.O(Alternativa más común en Windows, Android, Linux, macOS…)
- Los procesos hijos solo pueden aspirar a obtener un subconjunto de los recursos de su padre. Usado en sistemas robustos, ya que evita sobrecarga.

Mecanismos para pasar parámetros de inicialización a los procesos hijo:

- **Argumentos de línea de comandos** → Un proceso puede hacer uso de argumentos para indicar a procesos hijos opciones y argumentos en la línea de comandos.
- **Variables de entorno** → Variables dinámicas que pueden crear, leer y modificar durante la ejecución de un proceso.
- **Herencia de recursos** → Los procesos hijos pueden heredar cierto tipo de recursos del padre, lo que puede alterar el comportamiento del hijo. En los sistemas POSIX todos los archivos abiertos por un proceso son heredados en el mismo estado por sus hijos. Todo proceso, en este sistema, tiene por defecto tres archivos abiertos que corresponden a tres dispositivos de E/S:
    - **Entrada estándar**: Los procesos leen la entrada del teclado de la terminal
    - **Salida estándar:** El proceso escribe para mostrar texto en la pantalla de la terminal.
    - **Salida de error:** Usada para mostrar errores en la pantalla de la terminal.

Debido a la herencia de los archivos abiertos del proceso padre, todo proceso hijo tiene acceso a estos tres mismos dispositivos, y a su vez los hijos de los hijos, etc.

Qué ocurre con la ejecución del padre → Suelen contemplar dos posibilidades:

- El padre continúa ejecutándose al mismo tiempo que el hijo.
- El padre queda detenido hasta que algún hijo o todos acaben.

Cómo se construye el espacio de direcciones de los procesos hijos →

- Este espacio del proceso hijo sea un duplicado del que tiene el padre. -Este espacio del hijo se cree desde cero y cargue en él un nuevo programa.

1.7.2 Terminación de procesos

Un proceso termina cuando se lo indica el SO con la llamada al sistema **exit**, devolviendo un valor. El proceso padre puede esperar a que el hijo termine y recuperar ese valor a través de la llamada al sistema **wait**. Los **motivos para terminar** un procesos hijo pueden ser:

- El hijo ha excedido el uso de algunos de los recursos reservados.
- La tarea asignada al hijo ya no es necesaria.
- El padre termina, y el SO está diseñado para no permitir que el hijo pueda seguir ejecutándose si no tiene padre.

1.7.3 Ejemplo de operaciones con procesos

- Windows API: Se ofrece la función CreateProcess(). Recibe muchos argumentos pues permite configurar bastantes aspectos de la creación de procesos. Siempre necesita la ruta del ejecutable.
- POSIX API: Crean una llamada fork(), que se encarga de crear el proceso como copia del proceso padre. Como se trata de una copia, las nuevas variables o modificaciones no serán visibles para el otro. Pero como el proceso hijo hereda el acceso a todo tipo de recursos abiertos del padre, es sencillo crear un canal de comunicación entre ambos procesos.

1.8 Procesos cooperativos

Podemos clasificar los procesos en dos grupos:

- **Procesos independientes** → No afectan ni pueden ser afectados por otros procesos del sistema. Es cualquier proceso que no comparte datos con otros procesos.
- **Procesos cooperativos** → Pueden afectar y ser afectados por otros procesos del sistema. Es cualquier proceso que comparte datos con otros procesos.

Motivaciones para la colaboración entre procesos:

- **Compartición de información**→ Entorno proporcionado por el S.O que permite el acceso de recursos.
- **Velocidad de cómputo**→ Para que una tarea se ejecute más rápido se puede dividir en subtareas que se ejecuten en paralelo.
- **Modularidad**→ Divide funciones del programa en procesos separados que se comunican entre sí .
- **Conveniencia**→ Incluso un usuario individual puede querer hacer varias tareas al mismo tiempo.

Métodos de comunicación entre procesos:

- **Memoria compartida** → Los procesos utilizan regiones compartidas de la memoria principal para compartir información.
- **Paso de mensajes** → Los procesos utilizan funciones del S.O para enviarse mensajes entre ellos, compartiendo información y sincronizando acciones.

2. COMUNICACIÓN MEDIANTE PASO DE MENSAJES

Paso de mensajes → Mecanismo que permite a los procesos compartir información y sincronizar sus acciones sin necesidad de compartir recursos. Útil en entornos distribuidos. Este sistema debe ser proporcionado por el S.O, que se encarga de la sincronización y dar formato de los datos del mensaje.

2.1 Tamaño del mensaje

- **Mensajes de tamaño fijo**:
  Implementación del S.O muy sencilla → Uso de la interfaz por parte de las aplicaciones es compleja. Con estos mensajes no se preserva la separación entre mensajes al recibirlos, a esto se le denomina comunicación orientada a flujos.
- **Mensajes de tamaño variable:**
  Implementación del S.O más compleja → Uso de la interfaz es más sencillo.

2.2 Referenciación

Los procesos que quieran comunicarse deben tener una forma de señalarse el uno al otro. Para ello, el diseñador puede elegir entre paso de mensajes con comunicación directa o indirecta:

- **Directa** → Cada proceso debe nombrar explícitamente al proceso destinatario.
  - **Direccionamiento simétrico**: Tanto el proceso que envía como el que recibe tiene que identificar al otro para comunicarse

- **Direccionamiento asimétrico**: Solo el proceso que envía es el que identifica al otro.
- **Enlace de comunicaciones**: Se establece entre cada par de procesos que quieren comunicarse. Su enlace se asocia solo a dos procesos.

- **Indirecta** → Los mensajes son enviados a buzones, maillox o puertos, que son objetos donde los procesos pueden dejar y recoger mensajes.
  - **Enlace de comunicaciones**: Se establece entre dos procesos si comparten el mismo puerto. Un enlace puede estar asociado a más de dos procesos, puesto que muchos procesos pueden compartir el mismo puerto.

2.3 Buffering

Los mensajes intercambiados por enlace de comunicación se almacenan en una cola temporal. Hay tres formas de implementar dicha cola:

- **Capacidad cero o sin buffering** → La cola tiene una capacidad máxima de 0 mensajes, por lo que no puede haber ningún mensaje esperando en el enlace.
- **Buffering automático** → Existen dos opciones:
  - **Capacidad limitada** → La cola tiene capacidad máxima de N mensajes, por lo que si la cola se llena, el proceso transmisor se bloquea.
  - **Capacidad ilimitada** → La cola es de longitud potencialmente infinita.

2.4 Operaciones síncronas y asíncronas

En función de cómo implementar las llamadas de **send()** y **receive()** (comunicación entre procesos), si se pueden bloquear o no (send→cola transmisión llena; receive→cola recepción vacía), existen dos tipos de mensajes, **síncrono** (de bloqueo) y **asíncrono** (sin bloqueos).

- **Cuando el envío es asíncrono** → El proceso transmisor nunca se bloquea. Si se llama a send() cuando la cola de mensajes está llena, es común que retorne un código que indica que el proceso debe volver a ser enviado más tarde.
- **Cuando el envío es síncrono** → El proceso transmisor se bloquea cuando no hay espacio en la cola de mensajes hasta que pueda depositar mensajes en la misma.
- **Cuando la recepción es asíncrona** → El receptor nunca se bloquea. En caso de que la cola de mensajes esté vacía, el S.O envía indica al proceso intentarlo más tarde a través de un código de retorno o devolviendo un mensaje vacío.
- **Cuando la recepción es con bloqueo** → El receptor se bloquea cuando no hay mensajes en la cola hasta que llegue alguno.

2.5 Ejemplos de sistemas de paso de mensajes

2.5.1 Colas de mensajes POSIX

Son un caso de comunicación indirecta, con tamaño de mensaje variable, buffering con capacidad limitada y que soporta operaciones asíncronas. Las colas de mensajes son útiles para enviar mensajes de pequeño tamaño entre procesos que se ejecutan en el mismo sistema.

2.5.2 Señales en S.O POSIX

Es la forma más sencilla de comunicar dos procesos del mismo sistema en POSIX mediante envío de señal el uno al otro. Los procesos que quieran mandar esta señal utilizan la llamada al sistema kill(), requiere el identificador (mecanismo de comunicación directa) del proceso de destino y el número que identifica la señal.

2.5.3 Tuberías

Son un mecanismo de paso de mensajes de comunicación indirecta, orientada a flujos, capacidad limitada y comunicación síncrona (aunque puede soportar asíncrona). Usan funciones como read(), write(), close() para manipular archivos.

2.5.4 Sockets

Son un mecanismo de paso de mensajes de comunicación indirecta, que admite tanto comunicación orientada a flujos como mensajes de tamaño variable y buffering de capacidad limitada. También tanto comunicación síncrona como asíncrona, aunque el comportamiento real final de la interfaz depende de la tecnología de red utilizada.

3. MEMORIA COMPARTIDA

Es una estrategia para comunicar procesos donde uno de ellos gana acceso a regiones de la memoria del otro, algo que el S.O intentará evitar (Es necesario que los dos procesos eliminen dicha restricción). Dos procesos que comparten memoria pueden intercambiar información leyendo o escribiendo datos. Sin embargo, hay que tener en cuenta que:

- La estructura de los datos y su localización determinan los procesos en comunicación, y no el sistema operativo (a diferencia del paso de mensajes).
- Los procesos son responsables de sincronizarse para no escribir y leer en el mismo sitio de la memoria al mismo tiempo (puede generar inconsistencias).

Ventajas de la memoria compartida frente a otros mecanismos de comunicación:

- **Eficiencia**: Se trata de un mecanismo tremendamente rápido.
- **Conveniencia**: Es un sistema sencillo y fácil de usar (solo requiere leer y escribir de la memoria).

La memoria compartida puede ser anónima o con nombre:

3.1 Memoria compartida anónima

Solo existe para el proceso que crea y para sus procesos hijos, que heredan el acceso. Por tanto, es una forma eficiente de comunicar procesos padre e hijo.
En los sistemas POSIX, las funciones y operadores de reserva de memoria como malloc() y new, utilizan internamente la llamada al sistema mmap(). Esta función se puede llamar de la siguiente manera para reservar length bytes de memoria.

3.2 Memoria compartida con nombre

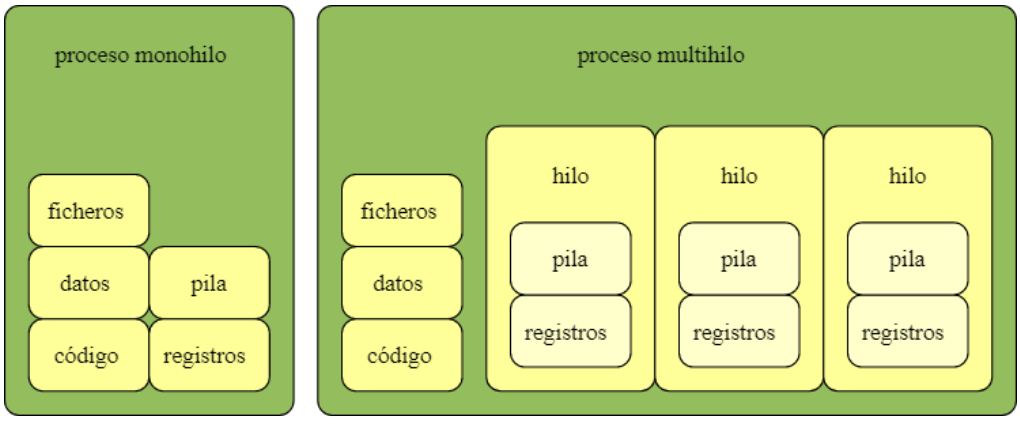
Es pública para el resto del sistema, por lo que cualquier proceso con permisos puede acceder a ella para comunicarse con otros procesos. Los objetos de memoria compartida con nombre tienen que ser creados antes de comenzar a utilizarlos(POSIX→shm_open()). Un objeto recién creado tiene tamaño 0, para redimensionarlo, se utiliza ftruncate(), que lo que necesita es el descriptor del objeto y el nuevo tamaño.

4. HILOS

Si queremos realizar varias tareas al mismo tiempo, estamos obligados a crear varios procesos y seleccionar un mecanismo de comunicación. Para ello, se ha extendido el concepto de proceso y permite que cada uno tenga múltiples secuencias de instrucciones. A cada una de estas secuencias se las conoce como **hilos**.

4.1 Introducción

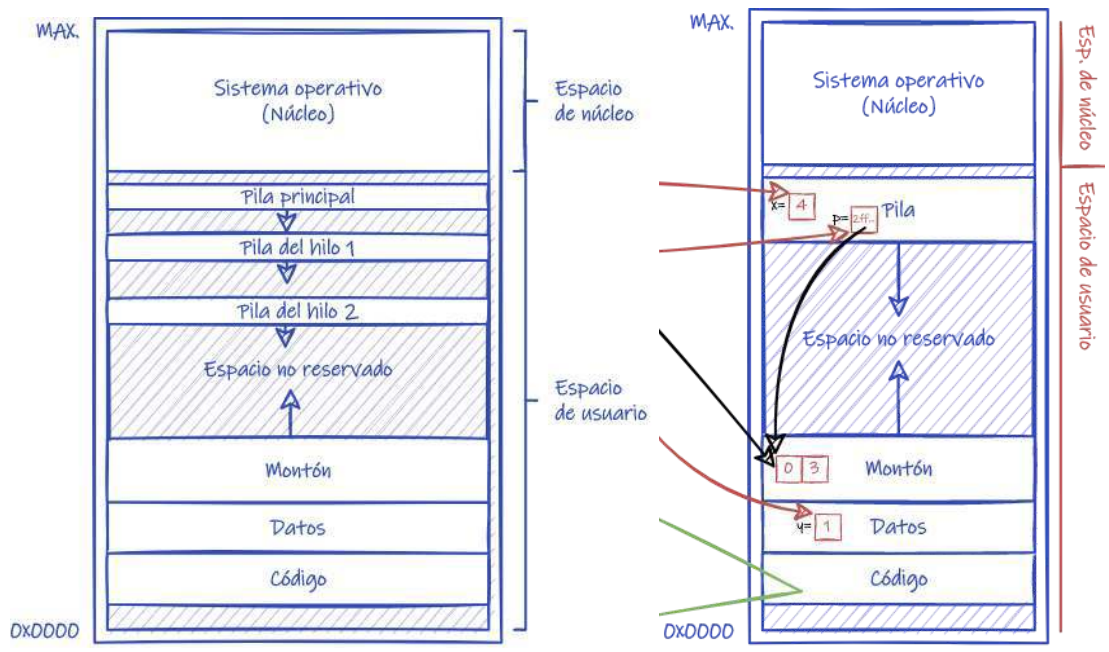
En los **S.O multihilo**, el hilo es la unidad básica de uso de la CPU, y no los procesos por separado, como hemos visto hasta ahora.



Cada hilo tiene una serie de recursos propios dentro del proceso:

- **Identificador del hilo**: Es único para cada hilo y sirve para identificarlos.
- **Contador de programa**: Es un registro de la CPU que indica la dirección de la próxima instrucción del hilo que debe ser ejecutada por la CPU.
- **Los registros de la CPU**: cuyos valores son diferentes para cada hilo, puesto que pueden ejecutar diferentes partes del mismo.
- **La pila**: Contiene datos temporales como argumentos y direcciones de retorno de las funciones y variables locales.
- **Código del programa**: El programa es el mismo para todos los hilos.
- **Los segmentos BSS y de datos y el montón**: Las secciones de datos diferentes de la pila son accesibles a todos los hilos.
- **Otros recursos del proceso:** archivos, sockets, tuberías…

Podemos ver como cambia la disposición de elementos de un proceso en la memoria cuando es multihilo:



MULTIHILO NO MULTIHILO

4.2 Beneficios

La programación multihilo aporta muchos beneficios:

4.2.1 Tiempo de respuesta

Una aplicación multihilo interactiva puede continuar ejecutando tareas aunque uno o varios hilos de la misma estén bloqueados o realizando operaciones muy lentamente, mejorando así el tiempo de respuesta al usuario.

Ej: un navegador web multihilo puede gestionar la interacción del usuario a través de un hilo, mientras el contenido solicitado se descarga en otro. Para hacer lo mismo en un navegador monohilo habría que utilizar comunicaciones asíncronas, de lo contrario, mientras el proceso está en estado esperando, a la espera de que lleguen los datos a través de la red, no puede atender las acciones del usuario.

4.2.2 Compartición de recursos

Al utilizar hilos, las tareas ejecutadas en ellos comparten los recursos automáticamente, sin que tengamos que hacer nada. Además, no solo comparten la memoria, sino también otros muchos recursos del proceso. Por lo que los hilos son una forma más conveniente de tener procesos que realizan diferentes actividades al mismo tiempo (en los S.O monohilo se crean varios procesos y se comunican mediante memoria compartida, lo que los hace menos eficientes).

4.2.3 Economía

Puesto que los hilos comparten los recursos de los procesos a los que pertenecen, son mucho más económicos de crear. También es más económico el cambio de contexto entre ellos, ya que hay que guardar y recuperar menos información al cambiar entre dos hilos de un mismo proceso.

4.2.4 Aprovechamiento de las arquitecturas multiprocesador

En los sistemas multiprocesador diferentes hilos pueden ejecutarse en paralelo en distintos procesadores. Por el contrario, un proceso monohilo solo se puede ejecutar en una CPU a la vez, independientemente de cuantas CPU estén disponibles para ejecutarlo.

4.3 Soporte multihilo

Las librerías de hilos proporcionan al programador la interfaz de programación para crear y gestionar los hilos de su proceso. Hay dos formas de implementar una librería de hilos:

4.3.1 Librería de hilos en el espacio de usuario

La librería se implementa en el espacio de usuario, junto al código y los datos del proceso, sin requerir ningún soporte especial por parte del núcleo.
Estos hilos no existen para el núcleo del S.O, solo para el proceso que los creó. Se los denomina hilos del nivel de usuario.

Como el código y los datos de la librería residen en el espacio de usuario, invocar una función de la misma se reduce a una simple llamada a una función, evitando el coste de hacer llamadas al sistema.

4.3.2 Librería de hilos en el núcleo

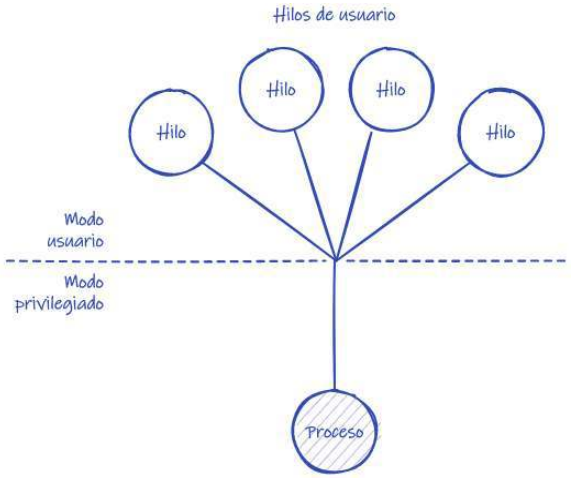
Si la librería de hilos se implementa en el núcleo, es el núcleo del sistema el que se encarga de darles soporte. Se los denomina hilos del nivel de núcleo.
Cada hilo tiene una estructura llamada bloque de control del hilo o **TCB (Thread Control Block)** que representa a cada hilo en el S.O y guarda información sobre su estado de actividad actual.

Como el código y los datos de la librería residen en el núcleo, invocar una función requiere frecuentemente hacer una llamada al sistema. Obviamente, la librería del sistema ofrece funciones para no tener que hacer la llamada al sistema directamente.

4.4 Modelos multihilo

4.4.1 Muchos a uno

Los hilos que ven el proceso se mapean en una única entidad planificable en la CPU del núcleo. Este es el modelo utilizado cuando el núcleo no soporta múltiples hilos de ejecución. La única entidad planificable de la CPU que conoce el núcleo es el proceso. La librería de hilos se implementa en el espacio de usuario.



Características del modelo:

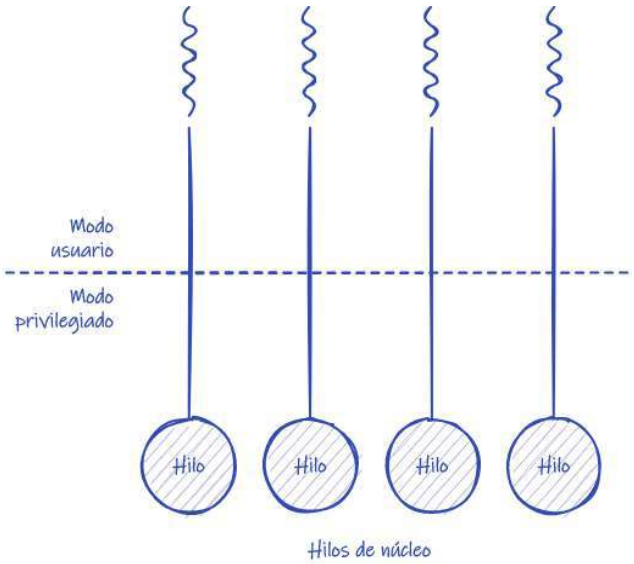
- Gestión de hilos se hace con una librería de espacio de usuario, por lo que los hilos se pueden **crear de forma rápida** y con **poco coste**.
- Si uno de los hilos solicita al S.O una operación que deba ser bloqueada a la espera (operaciones E/S..) todo el proceso es bloqueado.
- Como un solo hijo puede ser asignado al proceso, los hilos del mismo proceso no se pueden ejecutar en paralelo en sistemas multiprocesador.

Implementaciones del modelo:

A este modelo de hilos frecuentemente se lo llama Green Threads. Otras implementaciones de este modelo son las fibras de Windows API, Python, GNU.. Debido a su bajo coste, son muy útiles en sistemas multihilos, por su **bajo coste de recursos y alta eficiencia**.

4.4.2 Uno a uno

Cada hilo que ve el proceso se mapea en una entidad planificable en la CPU diferente del núcleo. Este modelo es utilizado cuando el núcleo del S.O soporta hilos de ejecución. En este caso, la librería de hilos se implementa en el núcleo, por lo que las entidades que planifican el núcleo en la CPU son los hilos de núcleo y los procesos pueden gestionar estos hilos mediante llamadas al sistema.



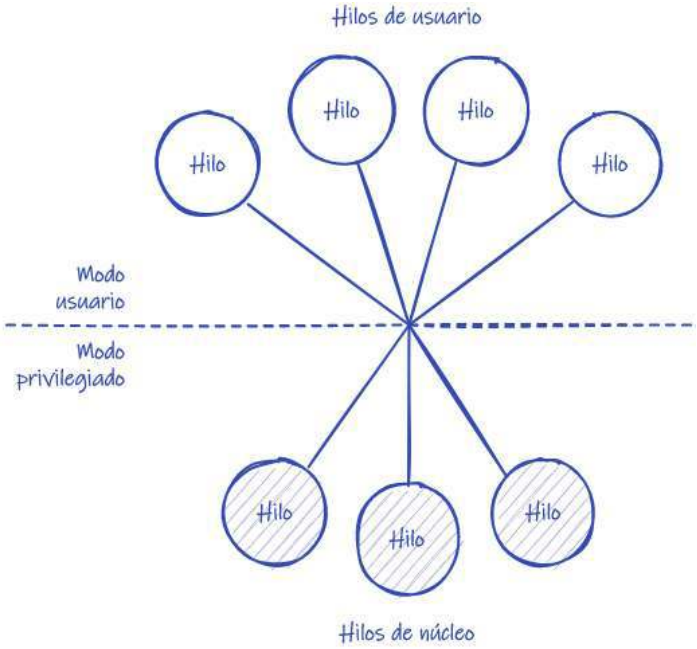
Características del modelo:

- Permite a otros hilos del mismo proceso ejecutarse aún cuando uno de ellos haga una llamada al sistema que debe bloquearse.
- Permite paralelismo en sistemas multiprocesador, ya que diferentes hilos pueden ser planificados por el núcleo en distintos procesadores.

- Crear un hilo para un proceso implica crear ciertas estructuras de datos en el núcleo. Muchos sistemas restringen la cantidad de hilos de núcleo por la cantidad de memoria disponible.
- La gestión de los hilos se hace con una librería en el espacio de núcleo, lo que requiere que el proceso haga llamadas al sistema para gestionarlos. Esto siempre es **más lento** que invocar simplemente una función, como ocurre en el modelo **muchos a uno**.

4.4.3 Muchos a muchos

En teoría debería ser posible aprovechar lo mejor de los dos modelos anteriores con una librería de hilos en el núcleo, para crear hilos de núcleo, y otra en el espacio de usuario, para crear hilos de usuario. Así los desarrolladores pueden utilizar la librería de hilos en el espacio de usuario para crear tantos hilos como quieran y que se ejecuten sobre los hilos de núcleo. El planificador de la librería de hilos se encarga de determinar qué hilo de usuario es asignado a qué hilo del núcleo.



Características del modelo:

- Permite paralelismo en sistemas multiprocesador ya que diferentes hilos de núcleo pueden ser planificados en distintos procesadores.
- Permite a otro hilo de usuario del mismo proceso ejecutarse en los otros hilos del núcleo cuando un hilo hace una llamada al sistema para bloquearse.

Activación del planificador:

Tanto en **modelo de muchos** a muchos como en el de **dos niveles** es necesaria la coordinación entre el núcleo y la librería de hilos del espacio de usuario para garantizar la máxima eficiencia.

Uno de los esquemas de comunicación se denomina **activación del planificador** y consiste en que el núcleo informa a la librería de hilos en espacio de usuario de que una llamada al sistema va a bloquear un hilo de proceso. Antes de eso, el núcleo crea un nuevo hilo de núcleo en el proceso, y se lo pasa a la librería de hilos en la notificación. Así, el planificador de la librería puede asignarle alguno de los otros hilos de usuario, evitando el bloqueo completo del proceso y ajustando el número de hilos de núcleo dinámicamente.

4.4.4 Dos niveles

Variación del modelo muchos a muchos donde, además de funcionar como antes explicado, se permite que un hilo de usuario quede ligado indefinidamente a un único hilo de núcleo, como en el modelo uno a uno.

4.5 Operaciones sobre los hilos

Como los procesos, los hilos pueden ser creados y eliminados dinámicamente.

- **Creación de hilos**: En un S.O con librería de hilos implementada en el núcleo, todo proceso se crea con un hilo principal, con el que comienza a ejecutarse el programa. El hilo principal puede crear otros hilos y estos a su vez, más hilos. No bastante, no existe una relación padres e hijos ni se crea árbol de hilos.
- **Cancelación de hilos**: Es la operación de terminar un hilo antes de que termine su trabajo. Si el usuario pulsa cancelar en una descarga, es necesario que todos los hilos que intervienen en la descarga paren. Esto puede ocurrir de dos maneras:
  - 1) En la **cancelación asíncrona** el hilo termina inmediatamente. Puede causar problemas al no liberarse los recursos reservados del proceso por parte del hilo. Además, si el hilo que termina estaba modificando datos que compartía con otros hilos, estos podrían quedar a medias.
  - 2) En la **cancelación en diferido**, el hilo comprueba periódicamente cuándo se debe terminar. La diferencia es que ahora el desarrollador conoce de antemano los puntos donde podría terminar el hilo.

5. SINCRONIZACIÓN

Ya vimos que varios procesos pueden compartir regiones de memoria (mem. compartida) y vimos que en los procesos multihilo todos los hilos comparten el espacio de direcciones del proceso al que pertenecen. Ambas posibilidades producen algunos riesgos, puesto que el acceso simultáneo a los datos compartidos pueden ocasionar inconsistencias. Es por ello que la sincronización juega un papel muy importante.

5.1 El problema de las secciones críticas

Condición de carrera → situación en la que varios procesos o hilos pueden acceder y manipular los mismos datos al mismo tiempo (de forma concurrente) y donde el resultado de la ejecución depende del orden particular en el que tienen lugar dichos accesos.

5.1.1 Problema del producto-consumidor

Supongamos que dos hilos o procesos comparten una región de la memoria que contiene un vector de elementos y un contador con el número de elementos del vector.
El primer hilo realiza varias tareas, en ocasiones añade un elemento al vector e incrementa el contador, es decir, primer hilo = productor.
El segundo hilo realiza varias tareas, pero en ocasiones debe tomar un elemento del vector compartido y decrementa el contador, es decir, segundo hilo = consumidor.
A la ejecución del código, la variable count llega a un resultado erróneo debido a que las sentencias no se ejecutan secuencialmente, sin mezclar operaciones(se ejecuta ++count y --count a la vez, lo que hace que una interrumpa a la otra).

5.1.2 Manipular estructuras de datos

El problema comentado también aparece en bloques de código destinados a tareas complejas. Por ejemplo, supongamos que vector ahora es una lista enlazada, de tal forma que extraer un elemento sería así:

```
--count;
item = vector.extract(count);
```

y el método `extract()` tendría que dar los siguientes pasos:

1. Iterar sobre la lista para buscar el nodo en la posición `count`.
2. Al encontrarlo, preservar en variables locales el puntero a ese nodo y al previo.
3. Cambiar en el nodo previo el puntero al siguiente nodo, para que apunte al nodo tras el que queremos extraer. En este momento el nodo a extraer ya no pertenece a la lista enlazada.
4. Extrae el `item` del campo que lo contiene en el nodo.
5. Destruir el nodo.
6. Salir del método retornando el elemento.

5.1.3 Exclusión mutua

Para evitar estas situaciones, debemos asegurarnos que **solo un hilo** en cada momento puede manipular recursos y variables compartidas. Es decir, buscar algún tipo de mecanismo para que cuando se ejecute ++count en un hilo, no se ejecute --count en otro hilo y viceversa.

Para resolver esto, debemos buscar las **secciones críticas** (donde se accede a variables, recursos compartidos). El acceso a secciones críticas debe ser controlado, para que cuando un hilo se esté ejecutando en una sección de este tipo, ningún otro pueda hacerlo en la suya correspondiente para manipular los mismos recursos. En estos casos se dice que existe **exclusión mutua** entre las secciones críticas.

5.1.4 Eventos

Las condiciones de carrera son el principal problema del código anterior del productor y consumidor, pero no el único, ya que también se usan bucles para que el hilo espere, si el vector está lleno o vacío → **espera ocupada / activa** → gasta tiempo de CPU inútilmente. En su lugar, se recomienda usar mecanismos de sincronización ofrecidos por el S.O → un hilo notifica de eventos a otro, para que se mantenga en estado esperando, dejando la CPU a los hilos que la necesitan .

5.2 Sincronización por hardware

Las soluciones ofrecidas por el S.O para resolver los problemas anteriores, suelen tener que apoyarse en características del hardware:

5.2.1 Bloque de las interrupciones

El problema de sección crítica puede ser resuelto en un sistema monoprocesador. El núcleo del S.O es un software controlado mediante interrupciones, por lo que basta con que los hilos bloqueen las interrupciones mientras se está dentro de la sección crítica. Así el S.O no puede tomar el control y asignar otro hilo a la CPU → impide que se ejecute otra secuencia de instrucciones que podría modificar los datos compartidos → no es práctico en sistemas multiprocesador.

5.2.2 Instrucciones atómicas

Todas las CPU modernas tienen instrucciones para modificar el contenido de una variable o intercambiar contenidos de dos variables de forma **atómica**, es decir, las operaciones se ejecutan como una unidad ininterrumpible. No importa que varias CPU ejecuten estas instrucciones simultáneamente → el hardware se encargará de que sean ejecutadas secuencialmente en algún orden arbitrario.
La importancia de estas instrucciones radica en que pueden ser utilizadas por el S.O para ofrecer soluciones sencillas a problemas de la sección crítica. Por ejemplo, semáforos o mutex.

5.3 Semáforos

Es un recurso que utiliza internamente instrucciones y otras características de la CPU para resolver problemas de secciones críticas. Los semáforos son un tipo de objeto del SO que controla el acceso a una sección crítica por medio de dos primitivas: acquire y release (wait y signal).

5.3.1 Tipos de semáforos

- **Anónimos** → Solo existen en el espacio de direcciones del proceso que los crea, por lo que están disponibles para sincronizar hilos del mismo proceso. Para sincronizar procesos diferentes o hilos en diferentes procesos, con Windows se pueden heredar de padres a hijos. En sistemas POSIX es necesario crear el semáforo en una región de memoria compartida.
- **Con nombre** → Son públicos al resto del sistema, por lo que cualquier proceso con permisos puede abrirlos para utilizarlos.

5.3.2 Ejemplos de uso de semáforos

En un programa para calcular el factorial, se usa un semáforo para que el proceso hijo indique al proceso padre que ha terminado de calcular el factorial y que el resultado está en memoria.

Para solucionar el problema productor-consumidor consideramos un semáforo para que haya exclusión mutua entre ambos al insertar y extraer elementos del vector. Y necesitamos una forma de que el productor espere cuando el vector esté lleno y que el consumidor haga lo mismo cuando el vector esté vacío. Una solución sería usar dos semáforos, uno para contar el número de elementos de un vector y otro para contar el número de huecos libres.

5.4 Mutex

Los semáforos inicializados a 1 se denominan **mutex** o **semáforos binarios**. Por tanto, si un lenguaje/sistema solo soporta semáforos, es directo implementar mutex, a la inversa igual. Por defecto solo se pueden utilizar para sincronizar hilos del mismo proceso, pero tienen un atributo para permitir la sincronización entre procesos diferentes.

5.4.1 Ejemplos de uso de mutex

En un problema donde se calculaba el factorial de un número, dos hilos retornaban el resultado. Ahora, cada hilo mete su resultado en un vector compartido, y el hilo principal recorre el vector multiplicando cada valor. Como ahora ambos hilos acceden a una estructura de datos compartida, esta debe ir protegida por un mutex antes de entrar a la sección crítica y liberarlo antes de salir de esta.

5.5. Variables de condición

En la solución del problema productor-consumidor usamos semáforos para implementar las esperas del productor y el consumidor cuando el vector está lleno o vacío. Los mutex no se pueden usar de la misma manera para señalar eventos, en su lugar necesitamos otro tipo de objeto llamado **variable de condición**, que soportan tres primitivas principales:

- **wait (Mutex)** → Es llamada por un hilo que desea esperar a que ocurra el evento que representa la variable de condición. El hilo debe haber adquirido antes el mutex y liberarlo en el momento de poner al hilo en estado esperando. Varios hilos pueden llamar a wait sobre la misma variable de condición, a la espera de que alguno use notify.
- **notify** → Es llamada por un hilo que quiere notificar el suceso de un evento a los hilos que esperan en la variable de condición. Uno de los hilos es despertado, adquiere el mutex que liberó al llamar a wait y, finalmente, retorna de wait para seguir ejecutándose.
- **notify all** → Es llamada por un hilo que quiere notificar el suceso de un evento a los hilos que esperan en la variable de condición. Todos los hilos son despertados e intentan adquirir el mutex que liberaron al llamar a wait. Cuando lo consiguen, retornan de wait para seguir ejecutándose. Obviamente, si todos hicieron wait sobre el mismo mutex, irán retornando de wait de uno en uno, porque solo un hilo puede tener el mutex al mismo tiempo.

5.5.1 Ejemplos del uso de variables de condición

Intentamos resolver el problema de productor-consumidor sin usar semáforos. Consideramos que:

- Necesitamos exclusión mutua entre ambos hilos al insertar y extraer elementos del vector para evitar condiciones de carrera, por tanto usamos **mutex** para proteger la sección crítica.
- Necesitamos una forma de que el productor espere cuando el vector está lleno y que el consumidor haga lo mismo cuando el vector está vacío. Para señalar estos eventos necesitamos **dos variables de condición**, una para indicar cuándo el vector está lleno y otro para indicar cuándo está vacío.

5.6 Funciones reentrantes y seguras en hilos

A la hora de utilizar una librería en un programa multihilo es necesario que tengamos en cuenta los conceptos reentrante y seguridad de hilos.

5.6.1 Funciones reentrantes

Función que puede ser interrumpida en medio de su ejecución y mientras espera, volver a ser llamada con total seguridad. Las funciones recursivas deben ser reentrantes. Una función es reentrante si:

- No modifica variables estáticas o globales. Si lo hiciera solo puede hacerlo mediante operaciones leer-modificar-escribir.
- No modifica su propio código y no ha llamado a otras funciones que no sean reentrantes.

5.6.2 Funciones de seguridad en hilos

Una función es segura en hilos si al manipular estructuras compartidas de datos lo hace de tal manera que se garantiza la ejecución segura de la misma por múltiples hilos al mismo tiempo. Estamos hablando de un problema de sección crítica, por lo que las funciones lo resuelven sincronizando el acceso a estos datos mediante el uso de semáforos o mutex.

A la hora de usar una función o librería que va a ser llamada desde múltiples hilos, primero debemos consultar la documentación para averiguar si es segura en hilos. Si no lo fuera, tendríamos que buscar funciones alternativas o recordar proteger las llamadas a las funciones no seguras con mecanismos de sincronización, para asegurar que solo son invocadas desde un hilo al mismo tiempo.

6. PLANIFICACIÓN DE LA CPU

El planificador de la CPU o planificador de corto plazo tiene la misión de seleccionar de la cola de preparados el siguiente proceso o hilo de núcleo a ejecutar. En dicha cola suelen estar los PCB de todos los procesos o hilos de núcleo que esperan una oportunidad para usar la CPU. Sea cual sea el algoritmo de planificación utilizado, este debe ser rápido.

6.1 Planificación expropiativa

El planificador debe ser invocado necesariamente en los siguientes casos:

- Cuando un proceso pase de ejecutando a esperando (por solicitar operación E/S, esperar a que un hijo termine, o un semáforo).
- Cuando un proceso termina.

Cuando el planificador de la CPU es invocado sólo en los casos anteriores, decimos que tenemos un S.O con planificación **cooperativa o no expropiativa**.

En la **planificación cooperativa,** cuando la CPU es asignada a un proceso, este la acapara hasta terminar o hasta pasar al estado de esperando. Esta planificación no requiere hardware especial.

Sin embargo, las decisiones de planificación también pueden ser tomadas en otros dos casos:

- Cuando ocurre una interrupción del temporizador (detecta si un proceso lleva mucho tiempo ejecutándose).
- Cuando el proceso pasa de esperando a preparando (por ej. termina la operación E/S que estaba esperando un proceso).

Cuando el planificador es invocado en los cuatro casos decimos que tenemos planificación **expropiativa o apropiativa**.

La **planificación expropiativa** sí requiere un soporte adecuado por parte del hardware, por lo que se utiliza en los S.O modernos. La utilización de un planificador expropiativo introduce algunas dificultades adicionales:

- El S.O debe proporcionar mecanismos de sincronización para coordinar el acceso a datos compartidos que podrían estar siendo modificados por el proceso que abandona la CPU.
- Dentro del núcleo se manipulan datos importantes compartidos por todo el sistema, que deben permanecer consistentes en todo momento.

Para resolver estas cuestiones, la solución más sencilla es impedir la expropiación dentro del núcleo. Es decir, el cambio de contexto no ocurre inmediatamente, sino que se retrasa hasta que la llamada del sistema se completa o se bloquea poniendo al proceso en el estado de esperando.

6.2 El asignador

Es el componente que da el control de la CPU al proceso seleccionado por el planificador de corto plazo. Esta tarea implica:

- Cambiar el contexto
- Cambiar al modo usuario
- Saltar al punto adecuado del programa para continuar la ejecución del proceso.

Puesto que el asignador es invocado para cada intercambio de procesos en la CPU, es necesario que el tiempo que tarda en detener un proceso e iniciar otro sea lo más corto posible. Al tiempo que transcurre desde que un proceso es escogido para ser planificado en la CPU hasta que es asignado a la misma se lo denomina **latencia de asignación**.

6.3 Criterios de planificación

Los diferentes algoritmos de planificación de la CPU tienen diversas propiedades que pueden favorecer a una clase de procesos respecto a otra. Hay muchos criterios para comparar los algoritmos de planificación de CPU. Los más comunes son:

6.3.1 Criterios a maximizar

Los algoritmos de planificación son mejores cuanto mayor es su valor para los siguientes criterios:

- **Uso de CPU** → Un buen planificador debería mantener la CPU lo más ocupada posible.

$$\text{Uso de CPU} = 100\frac{\text{Tiempo que la CPU permanece ocupada}}{\text{Tiempo durante el que se toma la medida}}\%$$

- **Tasa de procesamiento** → Cuando la CPU está ocupada es porque el trabajo se está haciendo. La tasa de procesamiento es una buena medida del volumen del trabajo (tareas o procesos terminados por unidad de tiempo).

$$\text{Tasa de procesamiento} = \frac{\text{Numero de procesos terminados}}{\text{Tiempo durante el que se toma la medida}}\text{procesos/s}$$

6.3.2 Criterios a minimizar

Los algoritmos de planificación son mejores cuanto menor es su valor para los siguientes criterios:

- **Tiempo de ejecución** → Intervalo de tiempo que transcurre desde que el proceso es cargado hasta que termina. Incluye el tiempo de espera debido a las operaciones de E/S → suele estar limitado por la velocidad de los dispositivos E/S.
- **Tiempo de espera** → Suma de tiempos que el proceso permanece a la espera en la cola de preparados. NO incluye el tiempo de espera debido a las operaciones de E/S.
- **Tiempo de respuesta** → Intervalo de tiempo que transcurre desde que se lanza un evento (una tecla, llega paquete por la interfaz de red) hasta que se produce la primera respuesta del proceso. Mide el tiempo que se tarda en responder y no el tiempo de E/S.

6.3.3 Elección del criterio adecuado

En función del tipo de sistema que se vaya a ejecutar puede ser conveniente medir la eficiencia del sistema usando un criterio u otro. Esto beneficiará a unos algoritmos de planificación frente a otros.

Podemos encontrar dos clases de trabajos para los que puede ser necesario evaluar la eficiencia del sistema de manera diferente, los trabajos interactivos y no interactivos:

- **Sistemas interactivos** → Los procesos pasan la mayor parte del tiempo esperando algún tipo de entrada por parte de los usuarios. El tiempo de ejecución no suele ser el mejor criterio para determinar la bondad de un algoritmo de planificación, ya que viene determinado en gran medida por la velocidad de la entrada de los usuarios. Por el contrario, se espera que el

sistema reaccione lo antes posible a las órdenes recibidas, lo que hace que el **tiempo de respuesta** sea un criterio más adecuado para evaluar al planificador de la CPU.

- **Sistemas no interactivos** → Lo que menos importa es el tiempo de respuesta y lo importante es completar cada tarea en el menor tiempo posible, por ello es aconsejable usar criterios tales como el **tiempo de ejecución** o la **tasa de procesamiento** (Ej: superordenadores de cálculos complejos).

Promedio o varianza del criterio → Estos criterios varían de un proceso a otro, por lo que se busca optimizar los valores promedios en el sistema, pero no se debe olvidar que hay casos en los que es más conveniente optimizar el máximo y mínimo antes que el promedio. Como hemos visto, por ejemplo, en los sistemas interactivos es más importante minimizar la varianza en el tiempo de respuesta que el tiempo de respuesta promedio.

6.4 Ciclo de ráfagas de CPU y de E/S

La ejecución de un hilo o proceso consiste en ciclos de CPU y esperas de E/S, de forma que alternan entre estos dos estados.
La ejecución empieza con una ráfaga de CPU, seguida por una ráfaga de E/S, que a su vez es seguida por otra de CPU y así sucesivamente. Finalmente, la última ráfaga de CPU finaliza con una llamada al sistema para terminar la ejecución del proceso.

Esta gráfica suele variar entre tipos de tareas y sistemas informáticos distintos. Significa que los procesos se pueden clasificar entre aquellos que representan un gran número de ráfagas de CPU cortas o aquellos con un pequeño número de ráfagas de CPU largas.

→ Decimos que un proceso es **limitado por la E/S** cuando representa muchas ráfagas de CPU cortas, pues pasa la mayor parte del tiempo esperando por la E/S.
→ Decimos que un proceso está **limitado por la CPU** cuando presenta pocas rágas de CPU largas, pues hace un uso intensivo de la misma y apenas pasa tiempo esperando por la E/S.

Esta distinción entre tipos de procesos puede ser importante en la selección de un algoritmo de planificación de CPU adecuado, puesto que, por lo general el algoritmo escogido debe planificar antes a los procesos limitados por la E/S, evitando así que los procesos limitados por la CPU la acaparen.

Planificar primero a los procesos limitados por la E/S tiene efectos positivos:

- Los procesos interactivos son generalmente procesos limitados por la E/S, por lo que planificarlos primero hace que mejore el tiempo de respuesta.
- El tiempo de espera promedio se reduce cuando se planifican primero los procesos con ráfagas de CPU cortas. Estos procesos son precisamente limitados por la E/S.

6.5 Algoritmos de planificación de la CPU

6.5.1 Planificación FCFS

First Come, First Served. El primero que llega, el primero servido, la cola es FIFO:

- Los procesos que llegan se colocan al final de la cola que le corresponde.
- El proceso asignado a la CPU se coge siempre del principio de la cola seleccionada.
- Es un algoritmo cooperativo, puesto que un proceso mantiene la CPU hasta que decide liberarla voluntariamente.

Ejemplo: Supongamos que 4 procesos llegan a la cola de preparados en los tiempos indicados en la tabla. Además, aunque es difícil tener un conocimiento a priori del tiempo de la ráfaga de CPU de cada proceso, vamos a suponer que también son conocidos:

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)
P1	0	24
P2	1	3
P3	2	3
P4	3	2

En la siguiente figura podemos ver el diagrama de Gantt de la planificación considerando que se utiliza el algoritmo **FCFS**:

Utilizando la ilustración, calculamos el tiempo de espera y ejecución promedio:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	24	(0-0)=	0	(24-0)=	24
P2	27	(24-1)=	23	(27-1)=	26
P3	30	(27-2)=	25	(30-2)=	28
P4	32	(30-3)=	27	(32-3)=	29
Tiempos promedio (ms)			**18.75**		**26.75**

→ El resultado cambia si los procesos llegan en otro orden. (Ej: P1 llega el último)

Podemos destacar que el algoritmo FCFS no garantiza tiempos de espera ni de ejecución mínimos, ya que, si en vez del P1, se hubiese ejecutado en P4, estos tiempos serían menores. Esto se le denomina efecto convoy. Durante el tiempo que P1 utiliza

toda la CPU, los otros procesos terminan sus operaciones de E/S y pasan a la cola de esperandos, mientras que los dispositivos de E/S están desocupados.
Cuando acaba P1, el resto de P se ejecutan rápidamente. Pasan a la cola de preparados donde los P tienen que esperar al P1 para usar la CPU. Es decir, que el orden de llegada influye en el tiempo que los procesos esperan para realizar su trabajo, y que es mejor que los procesos cortos se ejecuten primero.

6.5.2 Planificación SJF

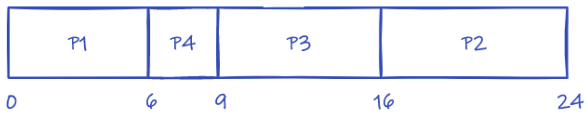
Shortest-Job first. Primero el más corto. Es decir:

- Se asocia a cada proceso la longitud de tiempo de su siguiente ráfaga de CPU.
- Cuando la CPU esté disponible, se ejecuta el proceso de menor ráfaga de CPU.
- Si dos procesos tienen ráfagas de una misma longitud, se utiliza el algoritmo FCFS (el que lleve más tiempo en la cola de preparados).
- Es un algoritmo cooperativo, puesto que un proceso mantiene la CPU hasta que decide liberarla voluntariamente.

Ejemplo:

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)
P1	0	6
P2	1	8
P3	2	7
P4	3	3

Considerando que se utiliza el algoritmo **SJF** obtendremos el siguiente diagrama de Gantt:



Con los tiempos de espera y ejecución promedio correspondientes:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	6	(0-0)=	0	(6-0)=	6
P2	24	(16-1)=	15	(24-1)=	23
P3	16	(9-2)=	7	(16-2)=	14
P4	9	(6-3)=	3	(9-3)=	6
Tiempos promedio (ms)			**6.25**		**12.25**

Si hubiéramos utilizado FCFS, el tiempo promedio hubiera sido de 14.75 → El algoritmo SJF es óptimo en el sentido de que el tiempo de espera promedio es mínimo, porque reduce más el tiempo de espera de los procesos cortos y aumenta el de los procesos largos. Además, así se evita el **efecto convoy**.

6.5.3 Planificación SRTF

SRTF: Shortest-remaining-time first. La forma expropiativa del SJF. La diferencia está en lo que ocurre cuando un nuevo proceso llega a la cola de preparados. En SRTF se compara el tiempo de la siguiente ráfaga de CPU del nuevo proceso, con el tiempo de ráfaga que le queda al proceso en ejecución. Si la primera magnitud es inferior, el proceso que tiene la CPU es expropiado y sustituido por el nuevo proceso.

Ejemplo:

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

El resultado es el que se muestra en el siguiente diagrama de Gantt:

Los tiempos de espera y ejecución promedio correspondientes serían:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	17	(0-0) + (10-1)=	9	(17-0)=	17
P2	5	(1-1)=	0	(5-1)=	4
P3	26	(17-2)=	15	(26-2)=	24
P4	10	(5-3)=	2	(10-3)=	7
Tiempos promedio (ms)			**6.50**		**13.00**

Los algoritmos expropiativos suelen ofrecer mejores tiempos de respuesta, puesto que un proceso que llega a la cola de preparados puede ser asignado a la CPU sin esperar a que el proceso que se ejecuta en ella actualmente termine su ráfaga de CPU.

6.5.4 Planificación con prioridades

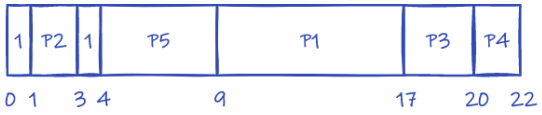
Se asocia una prioridad a cada proceso, de tal forma que el de mayor prioridad es asignado a la CPU. En caso de igual prioridad se utiliza FCFS. Las prioridades se suelen indicar con números enteros en un rango fijo, ej: [0-7]. Los números más grandes representan mayor prioridad. Este algoritmo puede ser expropiativo o cooperativo:

- **En caso de expropiativo**: cuando un proceso llega a la cola de preparados su prioridad es comparada con la del proceso en ejecución. Se expropia de la CPU si la prioridad del nuevo proceso es superior a la del ejecutando.
- **En el caso de cooperativo**: no se toma ninguna decisión cuando llega un proceso a la cola de preparados, solo cuando el que tiene asignada la CPU la abandona.

Ejemplo:

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)	Prioridad
P1	0	10	3
P2	1	2	1
P3	2	3	4
P4	3	2	5
P5	4	5	2

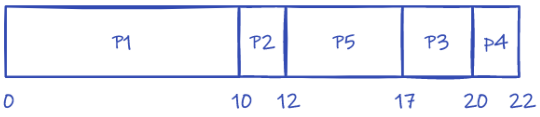
En las condiciones anteriores, si utilizamos el algoritmo de planificación por prioridades expropiativo, obtendremos el diagrama de Gantt de la siguiente figura:



Con los tiempos de espera y ejecución promedio correspondientes:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	4	(0-0) + (3-1) + (9-4)=	7	(17-0)=	17
P2	3	(1-1)=	0	(3-1)=	2
P3	20	(17-2)=	15	(20-2)=	18
P4	22	(20-3)=	17	(22-3)=	19
P5	9	(4-4)=	0	(9-4)=	5
Tiempos promedio (ms)			**7.80**		**12.20**

Mientras que si utilizamos el algoritmo de planificación por prioridades cooperativo, obtendremos el diagrama de Gantt de la siguiente figura:



Con los **tiempos de espera y ejecución promedio** correspondientes:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	10	(0-0)=	0	(10-0)=	10
P2	12	(10-1)=	9	(12-1)=	11
P3	20	(17-2)=	15	(20-2)=	18
P4	22	(20-3)=	17	(22-3)=	19
P5	17	(12-4)=	8	(17-4)=	13
Tiempos promedio (ms)			**9.80**		**14.20**

Hay dos maneras de asignar las prioridades:

- Internamente: Se utiliza una cualidad medible del proceso para calcular su prioridad. Por ej, límites de tiempo, necesidades de memoria, número de archivos abiertos…
- Externamente: las prioridades son fijadas por criterios externos al SO. Por ej, importancia del proceso para los usuarios, dinero para el uso del sistema…

Muerte por inanición

El mayor problema de este tipo de planificación es el bloqueo indefinido o muerte por inanición. Si hay muchos procesos de alta prioridad demandando CPU, el algoritmo puede dejar algunos procesos de menor prioridad esperando indefinidamente. Como solución, se aplica el **mecanismo de envejecimiento**, que consiste en aumentar gradualmente la prioridad de los procesos que esperan. De esta manera los procesos de baja prioridad tarde o temprano tendrán una oportunidad para ejecutarse. Una vez se les asigna la CPU, se restablece su prioridad al valor original.

6.5.5 Planificación RR

Round-Robin. Es similar al **FCFS** pero expropiativo con procesos cuando llevan cierta cantidad de tiempo ejecutándose en la CPU. Este algoritmo requiere:

- Definir una **ventana de tiempo** o **cuanto** (generalmente entre 10-100ms)
- Definir **cola de preparados** como una cola circular, donde el planificador asigna a la CPU a cada proceso en intervalos de tiempo de hasta un cuanto, como máximo.

Cuando un proceso está en la CPU pueden darse diversos casos:

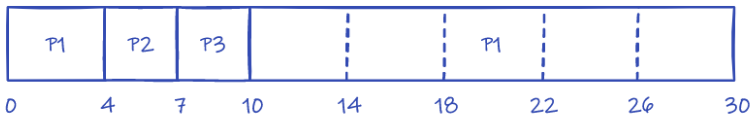
- La ráfaga de CPU es menor que un cuanto → El proceso liberará la CPU voluntariamente al terminar la ráfaga.
- La ráfaga de CPU es mayor que un cuanto → El temporizador interrumpirá el proceso al terminar el cuanto e informará al S.O. Este hará el cambio de contexto para asignar la CPU al siguiente proceso y el que abandona la CPU es insertado al final de la cola de preparados.

Este algoritmo es expropiativo, pues los procesos son expropiados por la interrupción del temporizador.

Ejemplo: Ilustremos el algoritmo en un ejemplo con cuanto de 4 ms

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)
P1	0	24
P2	1	3
P3	2	3

El resultado es el que se muestra en el siguiente diagrama de Gantt:



Y los **tiempos de espera y ejecución promedio** correspondientes serían:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	30	(0-0) + (10-4)=	6	(30-0)=	24
P2	7	(4-1)=	3	(7-1)=	6
P3	10	(7-2)=	5	(10-2)=	8
Tiempos promedio (ms)			**4.67**		**12.67**

Rendimiento

Cuando se utiliza la planificación RR, el **tamaño del cuanto es un factor clave** en la eficiencia del planificador:

- Cuando se **reduce el cuanto**, el tiempo de respuesta y espera promedio tienden a mejorar. Sin embargo, el número de contexto será mayor. Interesa que el cuanto sea mucho mayor que el tiempo del cambio de contexto.
- Cuando se **incremente el cuanto**, el tiempo de espera promedio también incrementa. Si el cuanto es muy grande, RR se convierte en FCFS, que suele tener grandes tiempos de espera promedio.

El tiempo de ejecución promedio generalmente mejora cuantos más procesos terminan su próxima ráfaga de CPU dentro de su cuanto. Por lo tanto, nos interesa un cuanto grande para que más procesos terminen su ráfaga dentro del mismo.

Reparto equitativo de la CPU

Uno de los inconvenientes de RR es que no garantiza un reparto equitativo del tiempo de la CPU entre los procesos limitados por la E/S y limitados por la CPU (aunque es mejor que FCFS). Esto es debido a que los primeros utilizan el procesador durante periodos cortos de tiempo, para bloquearse posteriormente a la espera de que se realice la operación de E/S que han solicitado. Cuando la espera termina, vuelven a la cola de preparados donde aguardan a que se les asigne la CPU → Los procesos limitados por la CPU hacen un mayor uso de la misma, mientras que los limitados por la E/S pueden tener que esperar durante bastante tiempo.

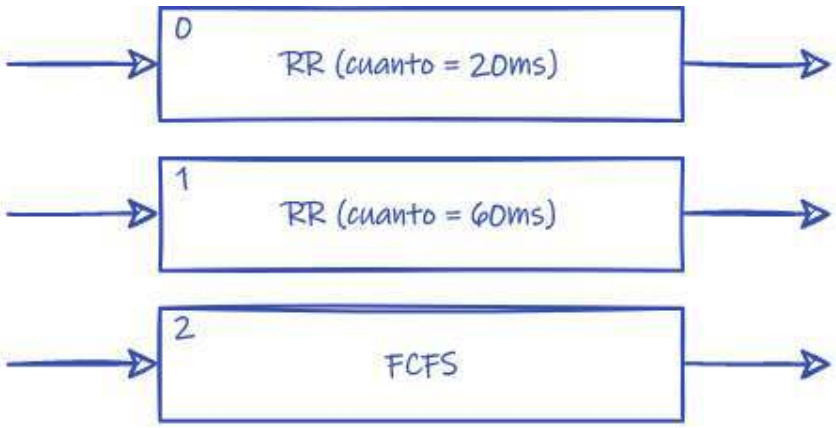
Para evitar esto se suele optar por un planificador de colas multinivel o por la planificación equitativa.

6.5.6 Planificación equitativa

No la pongo porque no aparecen ejercicios de esta.

6.5.7 Planificación con colas multinivel

Se recurre a esta planificación cuando se quiere combinar características de varios algoritmos. Se divide la cola de preparados en colas separadas, donde los procesos asignados a cada cola pueden tener un algoritmo de planificación distinto. La asignación de un proceso a una cola se hace en relación a alguna característica del proceso (si es interactivo o no, su prioridad o su tamaño en memoria, etc).



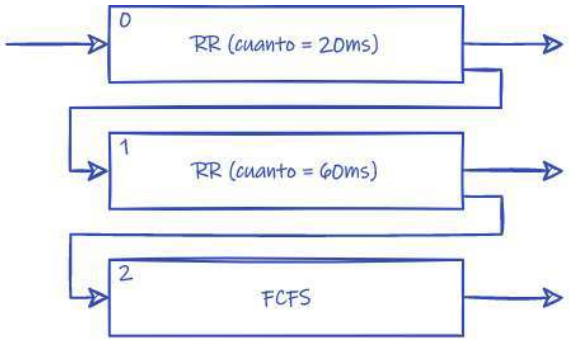
Cómo seleccionar la cola que debe escoger al siguiente proceso a ejecutar:

- **Usar planificador con prioridades** → Cada cola tiene una prioridad y así el planificador sólo tiene que escoger la cola de prioridad más alta que no esté vacía.
- **Usar cuantos sobre las colas** → A cada cola se le asigna una porción del tiempo de la CPU que debe repartirse entre los distintos procesos de la misma.

La planificación de colas multinivel con un planificador con prioridades para escoger la cola adecuada, es con diferencia la opción más común en los sistemas operativos modernos. Sin embargo, en este tipo de colas multinivel la asignación de los procesos a las colas es **permanente.**

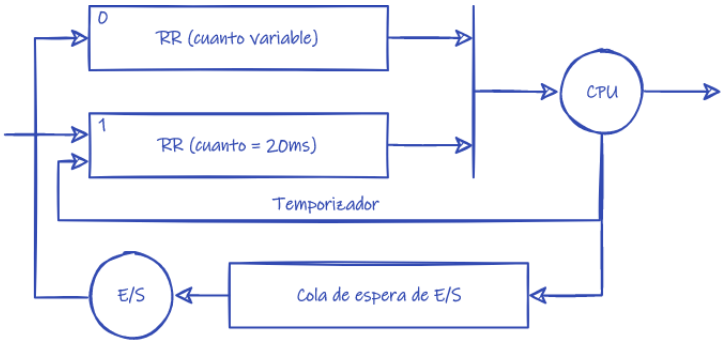
6.5.8 Planificación con colas multinivel realimentadas

Para aumentar la flexibilidad de la planificación con colas multinivel se puede permitir a los procesos pasar de una cola a otra. Así se pueden clasificar en colas distintas, procesos con diferentes tiempos de ráfaga de CPU.
Ej: situar los procesos interactivos o limitados por la E/S en las colas de más alta prioridad → mejora los tiempos de espera y de respuesta y evita el efecto convoy.



Ejemplo: Supongamos un planificador de colas multinivel donde cada cola tiene una prioridad, así que se usa la planificación con prioridades para seleccionar la cola. En las colas se usa el algoritmo RR para seleccionar el siguiente proceso, siendo mayor el cuanto de la cola cuanto menos prioritaria es la cola (imagen arriba).
Los procesos que llegan nuevos o desde el estado esperando lo hacen con la prioridad más alta (hemos decidido que sea 0) así que se insertan en la cola correspondiente. Mientras que los procesos expropiados por vencimiento del cuanto pierden un punto de prioridad, siendo insertados en una cola de prioridad menor.
El planificador de colas multinivel realimentadas también se puede utilizar para pasar a colas superiores los procesos que han esperado mucho tiempo en colas inferiores, evitando la muerte por inanición, que puede afectar a los sistemas de planificación de colas multinivel con prioridad fija.

El algoritmo **RR virtual** es un caso de planificador de **colas multinivel realimentadas** que resuelve los problemas del RR, en cuanto al reparto de la CPU entre procesos limitados por la E/S y limitados por la CPU.



6.6 Planificación de tiempo real

Ya hemos visto la importancia de los sistemas de tiempo real. Ahora veremos las funcionalidades necesarias para soportar la ejecución de procesos en tiempo real:

6.6.1 Tiempo real estricto

Estos sistemas son necesarios para realizar tareas críticas que deben ser completadas dentro de unos márgenes de tiempo preestablecidos. Generalmente las tareas son entregadas al S.O con una declaración de las restricciones de tiempo y la cantidad de tiempo que necesitan para ejecutarse. El planificador sólo admitirá las tareas si puede garantizar el cumplimiento de las restricciones de tiempo, rechazando en caso contrario.

6.6.2 Tiempo real flexible

La ejecución de los procesos aquí es menos restrictiva. Solo requiere que los procesos críticos reciban mayor prioridad que los que no lo son. Puede generar excesos en la cantidad de recursos asignados a los procesos de tiempo real, así como inanición. La mayoría de los S.O soportan este tiempo real flexible debido a que soportan multimedia, juegos, etc.

6.6.3 Implementación del soporte de tiempo real

Requiere:

- **SO con planificación con prioridades** → Los procesos de tiempo real deben tener la mayor prioridad y ser fija.
- **Baja latencia de asignación** → Cuanto menor sea la latencia, más rápido comenzará a ejecutarse el proceso de tiempo real después de ser seleccionado por el planificador de la CPU.

6.6.3 Reducir la latencia de asignación

Muchos SO tienen núcleo no expropiable. Estos núcleos no pueden realizar cambios de contexto mientras ejecutan código del núcleo, por lo que se ven obligados a que la operación termine antes de asignar la CPU a otro proceso. Esto aumenta la latencia de asignación. Se han desarrollado diversas ideas para que el código del núcleo sea expropiable:

- **Puntos de expropiación:** Una solución es introducir estos puntos en diversos lugares seguros del código. En ellos se comprueba si algún proceso de prioridad más alta está en la cola de preparados. En caso de ser así, se expropia la CPU al proceso actual y se le asigna al proceso de más alta prioridad.

- **Núcleo expropiable:** Puesto que en este caso la ejecución de cualquier operación en el núcleo puede ser interrumpida en cualquier momento por procesos de mayor prioridad que el que actualmente tiene asignada la CPU, es necesario proteger las estructuras de datos del núcleo con mecanismos de sincronización. Esto hace que el diseño de un núcleo de estas características sea mucho más complejo.
- **Inversión de prioridad:** Al hecho de que un proceso de alta prioridad tenga que esperar por uno de baja se le conoce como inversión de prioridad. Para resolverlo se utiliza un protocolo de herencia de la prioridad, donde un proceso de baja prioridad hereda la prioridad del proceso de más alta prioridad que espera por un recurso al que el primero está accediendo. En el momento en que el proceso de baja prioridad libere el acceso a dicho recurso, su prioridad retornará a su valor original

6.7 Planificación en sistemas multiprocesador

Para tratar el problema de la planificación en los sistemas multiprocesador nos limitaremos al caso de los **sistemas homogéneos**. En dichos sistemas los procesadores son **idénticos**, por lo que, en cualquiera de ellos, se puede ejecutar cualquier proceso. Según el tipo de procesamiento, existen diversas posibilidades a la hora de enfrentar el problema de la planificación en un sistema multiprocesador:

6.7.1 Multiprocesamiento asimétrico

Todas las decisiones de planificación, procesamiento de E/S, etc son gestionadas por el núcleo del sistema ejecutándose en un único procesador: el **servicio o maestro**. Este esquema es sencillo, puesto que evita la necesidad de compartir estructuras de datos entre el código que se ejecuta en los diferentes procesadores.

6.7.2 Multiprocesamiento simétrico (SMP)

Cada procesador ejecuta su propia copia del núcleo del S.O y se autoplanifica mediante su propio planificador de CPU. Encontramos varias alternativas:

- **Con una cola de preparados común** → requiere el uso de mecanismos de sincronización para controlar el acceso concurrente de los núcleos a las colas. En caso contrario, varios procesadores podrían escoger y ejecutar el mismo proceso a la vez. Sin embargo, esto presenta inconvenientes:
    - La posibilidad de que un proceso se pueda ejecutar en cualquier CPU es negativa desde el punto de vista de que dejan de ser útiles las cachés de los procesadores, penalizando notablemente el rendimiento del sistema.
    - Los mecanismos de sincronización requeridos para controlar el acceso a la cola de preparados pueden mantener a los procesadores mucho tiempo desocupados.

- **Con una cola para cada procesador** → No utiliza mecanismos de sincronización, por lo que se eliminan los tiempos de espera para acceder a la cola de preparados y escoger un nuevo proceso.
  El mayor inconveniente es que puede generar desequilibrios entre los procesadores, pues un proceso puede acabar desocupado mientras otro está muy ocupado. Por ello, el sistema dispone de algunos mecanismos de balanceo de carga:
  - En la **migración comandada**, una tarea específica estima la carga de trabajo de cada CPU y en caso de encontrar algún desequilibrio mueve algunos procesos de la cola de preparados de unos procesadores a la de otros.
  - En la **migración solicitada**, un procesador inactivo extrae de la cola de preparados de un procesador ocupado alguna tarea que esté esperando.