4. ¿Qué es el Kernel y por qué se considera el núcleo fundamental de un sistema operativo?

Explique qué recursos administra y cómo se relaciona con los procesos.

El Kernel (núcleo) es la parte central y fundamental de un sistema operativo. Su función principal es actuar como intermediario entre el hardware y los programas, permitiendo que las aplicaciones utilicen los recursos del computador de manera segura y organizada.

Recursos que administra el Kernel

El Kernel se encarga principalmente de:

- **Procesador (CPU): asigna tiempo de procesamiento a los diferentes procesos.**
- **Memoria RAM: controla qué cantidad de memoria utiliza cada proceso y evita que un programa interfiera con la memoria de otro.**
- **Dispositivos de entrada y salida: administra elementos como teclado, mouse, pantalla, impresoras y discos.**
- **Almacenamiento: coordina el acceso a archivos y dispositivos de almacenamiento.**
- **Procesos: crea, ejecuta, pausa, organiza y finaliza los procesos que se encuentran activos.**

Relación entre el Kernel y los procesos

Un proceso es un programa que está siendo ejecutado. El Kernel controla su ciclo de vida y decide cuándo y durante cuánto tiempo puede utilizar el procesador. Para ello utiliza mecanismos de planificación (*scheduling*), permitiendo que varios procesos parezcan ejecutarse simultáneamente.

Por ejemplo, cuando un usuario abre un navegador, el sistema operativo crea uno o varios procesos. El Kernel les asigna memoria y tiempo de CPU, controla su acceso a archivos y dispositivos y supervisa su ejecución.

En conclusión: el Kernel es fundamental porque coordina la comunicación entre el hardware, los procesos y las aplicaciones, administrando los recursos del sistema y garantizando que funcionen de manera eficiente, ordenada y segura.

Complemento

Scheduling (planificación) es el mecanismo mediante el cual el **Kernel decide qué proceso debe utilizar el procesador (CPU), cuándo debe hacerlo y durante cuánto tiempo**.

¿Por qué es necesario?

En un computador pueden existir muchos procesos ejecutándose al mismo tiempo: un navegador, un reproductor de música, un antivirus, el sistema operativo, etc. Como la CPU debe repartir su capacidad entre ellos, el **scheduling organiza ese acceso**.

Ejemplo sencillo

Imagina que la CPU es **un profesor** y los procesos son **estudiantes esperando para hablar**:

1. El profesor selecciona un estudiante.
2. Le permite hablar durante un tiempo determinado.
3. Luego pasa a otro estudiante.
4. Continúa organizando los turnos para que todos puedan participar.

De manera similar:

Procesos → Scheduling → CPU

¿Qué decide el Scheduling?

Principalmente:

- **Qué proceso se ejecuta.**

- **Cuánto tiempo utiliza la CPU.**
- **Cuando se cambia de un proceso a otro.**
- **Qué procesos tienen mayor prioridad.**

Algunos algoritmos de Scheduling

Algoritmo	**¿Cómo funciona?**
FCFS	Atiende primero al proceso que llegó primero.
Round Robin	Cada proceso recibe un pequeño intervalo de tiempo.
Prioridades	Se ejecutan primero los procesos con mayor prioridad.
SJF	Prioriza los procesos que requieren menor tiempo de ejecución.

En pocas palabras: el **Scheduling es el sistema de turnos de la CPU**. El Kernel lo utiliza para administrar los procesos y aprovechar eficientemente el procesador.

5. ¿Qué es un proceso y cuál es la diferencia entre un programa y un programa en ejecución?

Investigue además qué información necesita el sistema operativo para controlar un proceso.

5. ¿Qué es un proceso?

Un **proceso** es un **programa que se encuentra en ejecución**. Cuando un programa está almacenado en el disco, es simplemente un conjunto de instrucciones; cuando el sistema operativo lo carga en la memoria y comienza su ejecución, se convierte en un **proceso**.

Diferencia entre programa y proceso

Concepto	Descripción	Ejemplo
Programa	Conjunto de instrucciones almacenadas que aún no se están ejecutando.	El archivo Chrome.exe almacenado en el disco.
Proceso	Programa que está siendo ejecutado y al que el sistema operativo le ha asignado recursos.	Google Chrome abierto y funcionando.

Ejemplo sencillo:
Un programa puede compararse con una **receta de cocina**. La receta contiene las instrucciones, pero no está realizando ninguna acción. Cuando alguien comienza a preparar la receta, tenemos una situación equivalente a un **proceso en ejecución**.

¿Qué información necesita el sistema operativo para controlar un proceso?

El sistema operativo necesita mantener información sobre cada proceso. Esta información se almacena principalmente en una estructura denominada **PCB (Process Control Block o Bloque de Control de Procesos)**.

El PCB contiene información como:

- **Identificador del proceso (PID):** número único que permite identificarlo.
- **Estado del proceso:** indica si está listo, ejecutándose, bloqueado, esperando, etc.
- **Contador de programa (Program Counter):** indica cuál es la próxima instrucción que debe ejecutar el proceso.
- **Registros de la CPU:** contienen información necesaria para continuar la ejecución correctamente.
- **Información de memoria:** indica qué espacios de memoria utiliza el proceso.
- **Información de planificación (Scheduling):** prioridad del proceso y otros datos utilizados para decidir cuándo utilizará la CPU.
- **Recursos asignados:** archivos abiertos, dispositivos y otros recursos que está utilizando.
- **Información de entrada/salida:** operaciones de E/S que el proceso está realizando o esperando.

Idea fundamental

Podemos resumirlo así:

Programa → se carga en memoria → comienza a ejecutarse → se convierte en proceso

Y el **Kernel** utiliza el **PCB** para saber **quién es el proceso, en qué estado se encuentra, qué recursos utiliza y cómo debe continuar su ejecución**.

Por eso, el proceso no es solamente el programa: **incluye el programa en ejecución + su estado + los recursos y la información que necesita el sistema operativo para administrarlo**.

PCB significa **Process Control Block**, en español **Bloque de Control de Procesos**.

Es una **estructura de datos que utiliza el sistema operativo para almacenar toda la información necesaria para controlar y administrar un proceso**.

¿Para qué sirve?

Imaginemos que el sistema operativo tiene cientos de procesos. Necesita saber:

- ¿Cuál es cada proceso?
- ¿Está ejecutándose o esperando?
- ¿Qué parte del programa estaba ejecutando?
- ¿Qué memoria está utilizando?
- ¿Qué prioridad tiene?
- ¿Qué recursos tiene asignados?

Toda esta información se organiza en el **PCB**.

¿Qué contiene un PCB?

Entre la información más importante encontramos:

Información	¿Para qué sirve?
PID	Identifica de manera única al proceso.
Estado	Indica si está ejecutándose, listo, bloqueado, etc.
Program Counter	Indica cuál es la siguiente instrucción que debe ejecutar.

Información	¿Para qué sirve?
Registros de CPU	Guarda información necesaria para continuar la ejecución.
Información de memoria	Indica qué memoria utiliza el proceso.
Prioridad	Ayuda al *scheduling* a determinar cuándo debe ejecutarse.
Recursos	Registra archivos, dispositivos y otros recursos utilizados.

Ejemplo sencillo

Supongamos que abres **Microsoft Word**.

El sistema operativo crea un proceso para Word y genera un **PCB** asociado:

Word → Proceso → PCB

Si el sistema operativo debe detener temporalmente Word para darle la CPU a otro proceso, guarda en el PCB información sobre **dónde estaba Word y cuál era su estado**. Cuando vuelva a asignarle la CPU, puede utilizar esa información para **continuar la ejecución correctamente**.

En una frase

El PCB es como la "ficha de identificación y control" que el sistema operativo mantiene para cada proceso.

Sin esta información, el Kernel tendría grandes dificultades para **administrar, pausar, reanudar y finalizar los procesos**.

6. ¿Qué es un hilo y qué relación existe entre procesos e hilos?

Construya un ejemplo utilizando una aplicación cotidiana, como un navegador, videojuego o editor de texto.

Un **hilo**, también llamado **Thread**, es la **unidad más pequeña de ejecución dentro de un proceso**. Un proceso puede tener uno o varios hilos que realizan diferentes tareas de manera concurrente.

Podemos entender la relación así:

Proceso → contiene uno o varios → Hilos (Threads)

¿Cuál es la relación entre procesos e hilos?

Un **proceso** es un programa en ejecución que posee sus propios recursos, como memoria y archivos abiertos. Dentro de ese proceso pueden existir varios **hilos**, que comparten muchos de esos recursos y permiten realizar diferentes tareas.

Proceso	Hilo
Es un programa en ejecución.	Es una unidad de ejecución dentro de un proceso.
Tiene su propio espacio de memoria.	Comparte memoria y recursos con otros hilos del mismo proceso.
Es más pesado de crear y administrar.	Es más ligero y rápido de crear.
Puede contener varios hilos.	Pertenece a un proceso.

Ejemplo: un navegador web

Supongamos que abrimos **Google Chrome** y tenemos varias actividades al mismo tiempo:

Proceso del navegador

⬇️

├── Hilo para cargar una página web
├── Hilo para ejecutar JavaScript
├── Hilo para reproducir un video
├── Hilo para responder al teclado y mouse
└── Hilo para realizar tareas de red

Aunque todas estas tareas pertenecen al navegador, pueden ejecutarse mediante diferentes hilos.

Por ejemplo, mientras **un hilo descarga información de Internet**, otro puede encargarse de **responder a los movimientos del mouse**, mientras otro **procesa contenido de la página**.

Una analogía sencilla

Imaginemos un **restaurante**:

- **Proceso = restaurante:** tiene sus propios recursos, espacio y materiales.
- **Hilos = trabajadores:** cada uno realiza una tarea diferente.
- **Recursos compartidos = cocina, utensilios e ingredientes.**

El restaurante puede funcionar con varios trabajadores realizando tareas simultáneamente.

En resumen

Un proceso es un programa en ejecución, mientras que un hilo es una unidad de ejecución dentro de ese proceso. Un proceso puede tener varios hilos trabajando de manera concurrente y compartiendo recursos.

Esto permite que aplicaciones como los navegadores, videojuegos y editores de texto sean **más rápidas, fluidas y capaces de realizar varias tareas al mismo tiempo**.

Concurrente significa que **varias tareas pueden avanzar durante el mismo período de tiempo**, compartiendo los recursos disponibles del sistema.

En sistemas operativos, la **concurrencia** permite que un programa gestione varias tareas aparentemente al mismo tiempo.

7. ¿Qué es una llamada al sistema y para qué sirve?

Investigue y explique las llamadas:

- fork

- kill
- signal
- wait
- create
- fopen
- fclose

¿Qué es una llamada al sistema (*System Call*)?

Una **llamada al sistema** (*System Call*) es un mecanismo mediante el cual un **programa solicita un servicio al sistema operativo**.

Las aplicaciones no pueden acceder directamente a todos los recursos del computador. Por ejemplo, un programa no debería manipular directamente la memoria, crear procesos o acceder al hardware sin control. Por eso, utiliza llamadas al sistema para solicitar al **Kernel** que realice determinadas operaciones.

¿Cómo funciona?

De forma simplificada:

Aplicación → Llamada al sistema → Kernel → Recurso del sistema

Por ejemplo, si un programa necesita abrir un archivo:

Programa → fopen() → Sistema operativo → Archivo

El Kernel verifica la solicitud, realiza la operación y devuelve un resultado al programa.

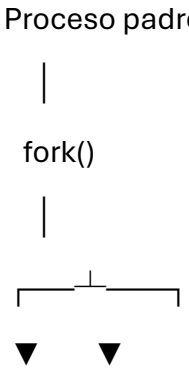
Principales llamadas solicitadas

fork()

Se utiliza principalmente en sistemas **Unix/Linux** para **crear un nuevo proceso** a partir de un proceso existente.

El proceso que realiza fork() se conoce como **proceso padre**, y el nuevo proceso se denomina **proceso hijo**.

Ejemplo conceptual:



Una característica importante es que, después de fork(), **ambos procesos continúan ejecutándose desde el punto posterior a la llamada**.

---

kill()

Permite **enviar una señal a un proceso** identificado normalmente mediante su PID.

Aunque su nombre puede generar confusión, kill() **no necesariamente termina el proceso**. Dependiendo de la señal enviada, puede solicitar diferentes acciones.

Por ejemplo:

kill(PID, SIGTERM)

Puede solicitar al proceso que termine de manera controlada.

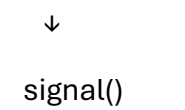
---

signal()

Se utiliza para **establecer qué debe hacer un programa cuando recibe una determinada señal**.

Por ejemplo, un programa puede indicar que quiere ejecutar una función específica cuando recibe una señal determinada.

Conceptualmente:



↓

Función definida por el programa

Esto permite que los programas respondan ante determinados eventos generados por el sistema operativo u otros procesos.

---

wait()

Permite que un **proceso padre espere a que uno de sus procesos hijos termine**.

Por ejemplo:

Proceso padre

│

├── crea hijo

│

├── wait()

│ ↓

│ espera

│

└── continúa cuando el hijo termina

Es especialmente importante para evitar que los procesos hijos terminados queden como **procesos zombis**.

---

create

Aquí es importante hacer una precisión: **create no es una única llamada al sistema estándar de Unix/Linux con un significado universal**.

Dependiendo del sistema operativo o API, pueden existir funciones denominadas create para crear diferentes recursos.

Por ejemplo, en **Windows** existen funciones como:

- CreateProcess() → crea un proceso.

- CreateFile() → crea o abre un archivo.
- CreateThread() → crea un hilo.

Por lo tanto, si en el contexto de la asignatura se habla simplemente de **create**, probablemente se está utilizando como concepto general de **creación de recursos**, especialmente procesos, archivos o hilos.

---

fopen()

fopen() se utiliza en el lenguaje **C** para **abrir un archivo**.

Por ejemplo:

FILE *archivo;

archivo = fopen("datos.txt", "r");

La "r" indica que se quiere abrir el archivo en **modo lectura**.

Otros modos comunes son:

Modo	Función
"r"	Lectura
"w"	Escritura
"a"	Agregar información al final

Aunque fopen() pertenece a la **biblioteca estándar de C**, internamente termina utilizando mecanismos del sistema operativo para acceder al archivo.

---

fclose()

fclose() se utiliza para **cerrar un archivo que previamente fue abierto con fopen()**.

Ejemplo:

FILE *archivo;

archivo = fopen("datos.txt", "r");

/* operaciones con el archivo */

fclose(archivo);

Cerrar el archivo permite liberar recursos asociados con él y garantizar que las operaciones pendientes se completen correctamente.

Resumen

Llamada	Función principal
fork()	Crea un proceso hijo.
kill()	Envía una señal a un proceso.
signal()	Define cómo responder ante determinadas señales.
wait()	Permite que un proceso espere a que termine un hijo.
create	Concepto general de creación; depende del sistema/API.
fopen()	Abre un archivo en C.
fclose()	Cierra un archivo abierto en C.

Idea fundamental

Las llamadas al sistema son el **puente entre las aplicaciones y el Kernel**:

Programa → System Call → Kernel → Recurso

Por ejemplo, cuando un programa necesita **crear un proceso, comunicarse con otro proceso, esperar a un proceso hijo o trabajar con un archivo**, utiliza mecanismos que permiten solicitar estos servicios al sistema operativo.

8. ¿Qué es un Shell o intérprete de comandos y qué importancia tiene para la interacción con el sistema operativo?

Compare comandos equivalentes de Windows y Linux, tomando como referencia dir, ls, cls, cat y chmod.

¿Qué es un Shell o intérprete de comandos?

Un **Shell** o **intérprete de comandos** es un programa que permite al usuario **comunicarse con el sistema operativo mediante comandos escritos**.

El Shell recibe una instrucción del usuario, la interpreta y solicita al sistema operativo que realice la acción correspondiente.

La relación puede representarse así:

Usuario → Shell → Kernel → Hardware

Por ejemplo, cuando escribimos ls en Linux, el Shell interpreta el comando y solicita al sistema operativo que muestre los archivos y carpetas del directorio actual.

¿Qué importancia tiene?

El Shell permite realizar muchas tareas sin utilizar una interfaz gráfica, como:

- Crear, eliminar y copiar archivos.
- Crear y administrar carpetas.
- Ejecutar programas.
- Administrar procesos.
- Configurar permisos.
- Automatizar tareas mediante scripts.
- Administrar sistemas de manera remota.

En **Windows** encontramos principalmente **CMD** y **PowerShell**, mientras que en **Linux** son comunes Shells como **Bash**, Zsh y otros.

Comparación de comandos Windows y Linux

Acción	Windows (CMD)	Linux (Bash)	Función
Listar archivos	dir	ls	Muestra archivos y carpetas.
Limpiar pantalla	cls	clear	Limpia el contenido visible de la terminal.
Mostrar contenido de un archivo	type archivo.txt	cat archivo.txt	Muestra el contenido de un archivo de texto.
Cambiar permisos	icacls	chmod	Modifica permisos de archivos y carpetas.

dir vs. ls

En **Windows CMD**:

dir

Muestra los archivos y carpetas del directorio actual.

En **Linux**:

ls

Realiza una función equivalente.

Por ejemplo:

ls -l

muestra información más detallada sobre los archivos.

---

cls vs. clear

En Windows CMD:

cls

Limpia la pantalla de la consola.

En Linux:

clear

cumple una función equivalente.

---

cat y su equivalente en Windows

En Linux:

cat archivo.txt

muestra el contenido del archivo.

En Windows CMD, el comando equivalente es:

type archivo.txt

Por ejemplo:

type notas.txt

muestra el contenido de notas.txt.

---

chmod y su equivalente en Windows

chmod es un comando característico de sistemas **Unix/Linux** y permite modificar los **permisos de acceso** de archivos y directorios.

Por ejemplo:

chmod 755 script.sh

permite establecer determinados permisos de lectura, escritura y ejecución.

En Windows CMD, una herramienta equivalente para administrar permisos es:

icacls archivo.txt

Por ejemplo:

icacls archivo.txt

permite consultar las listas de control de acceso (**ACL**) del archivo.

Una aclaración importante

Los comandos **no siempre tienen una equivalencia exacta** entre Windows y Linux. Esto se debe a que utilizan **modelos de permisos, estructuras de archivos y herramientas administrativas diferentes**.

Por ejemplo:

Linux:

chmod 755 archivo.sh

utiliza permisos tradicionales de Unix basados en **usuario, grupo y otros**.

Windows:

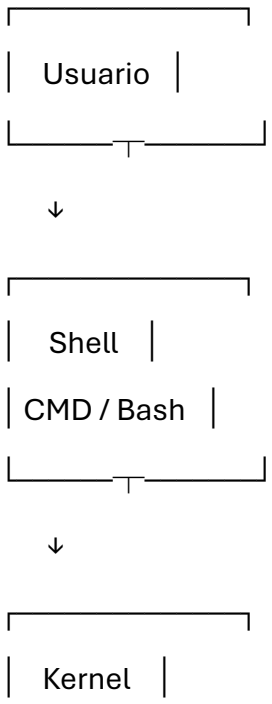
icacls archivo.txt

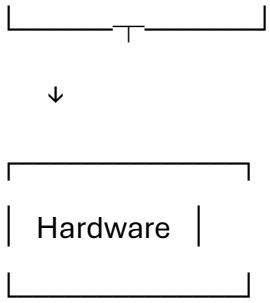
trabaja principalmente con **ACL (Access Control Lists)** y permisos asociados a usuarios y grupos.

Para recordar

El **Shell es el intermediario que permite al usuario dar instrucciones al sistema operativo mediante comandos**.

Una forma sencilla de visualizarlo es:





En conclusión: el Shell facilita la interacción con el sistema operativo y proporciona una herramienta poderosa para **administrar archivos, procesos, permisos y recursos**, tanto en Windows como en Linux.

14. ¿Qué es la planificación de procesos y cuáles son los principales algoritmos ?

Investigue y explique:

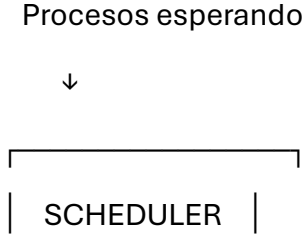
- Round Robin.
- Planificación por prioridades.
- Colas múltiples.
- Primero el trabajo más corto.
- Planificación en dos niveles.

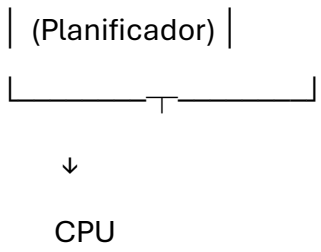
¿Qué es la planificación de procesos?

La **planificación de procesos** (*Process Scheduling*) es el mecanismo mediante el cual el **sistema operativo decide qué proceso utilizará la CPU, cuándo lo hará y durante cuánto tiempo**.

Como normalmente existen varios procesos que necesitan utilizar el procesador, el **Kernel**, mediante el planificador (*Scheduler*), organiza los turnos de ejecución.

Por ejemplo:





El objetivo es utilizar la CPU de manera eficiente y conseguir que los programas tengan **buen tiempo de respuesta, poca espera y un uso adecuado de los recursos**.

Principales algoritmos de planificación

Round Robin (RR)

Round Robin significa aproximadamente **"por turnos"**.

Cada proceso recibe una cantidad fija de tiempo de CPU llamada **quantum o intervalo de tiempo**. Cuando termina ese intervalo, el proceso es interrumpido y la CPU pasa al siguiente proceso.

Ejemplo:

Supongamos tres procesos:

P1 → P2 → P3 → P1 → P2 → P3 → ...

Si el *quantum* es de 2 segundos:

P1 | P2 | P3 | P1 | P2 | P3

2s 2s 2s 2s 2s 2s

Ventaja: proporciona una distribución relativamente justa del procesador.

Desventaja: si el *quantum* es demasiado pequeño, aumenta el número de cambios de contexto.

Planificación por prioridades

En este algoritmo, cada proceso tiene asignada una **prioridad**.

El sistema operativo selecciona primero el proceso que tenga la prioridad correspondiente según la política utilizada.

Por ejemplo:

P1 → Prioridad 3

P2 → Prioridad 1

P3 → Prioridad 2

Si **1 representa la mayor prioridad**, el orden podría ser:

P2 → P3 → P1

Puede utilizarse tanto en sistemas **preemptivos** como **no preemptivos**.

Ventaja: permite atender rápidamente procesos importantes.

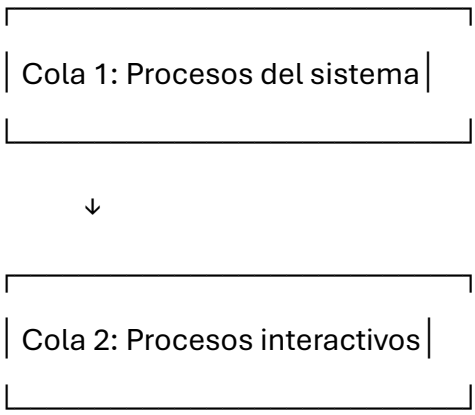
Desventaja: un proceso con baja prioridad podría esperar demasiado tiempo. Este problema se conoce como **inanición (*starvation*)**.

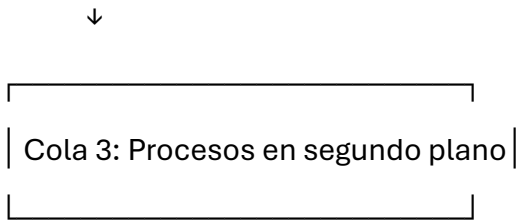
Una técnica llamada **aging (envejecimiento)** puede aumentar gradualmente la prioridad de los procesos que llevan mucho tiempo esperando.

Colas múltiples (*Multilevel Queue*)

En la **planificación mediante colas múltiples**, los procesos se organizan en **diferentes colas según sus características o función**.

Por ejemplo:





Cada cola puede utilizar un algoritmo diferente.

Por ejemplo:

- Procesos interactivos → **Round Robin**
- Procesos por lotes → **FCFS**

Una característica importante es que, en la **cola multinivel clásica**, los procesos normalmente **permanecen en la cola asignada** y no cambian de una cola a otra.

Ventaja: permite organizar procesos según sus necesidades.

Desventaja: una mala asignación de prioridades entre las colas puede provocar que algunos procesos esperen demasiado.

Primero el trabajo más corto (SJF)

SJF (Shortest Job First) significa **"Primero el trabajo más corto"**.

El sistema operativo selecciona el proceso que necesita **menos tiempo de CPU para terminar**.

Por ejemplo:

P1 → 8 segundos

P2 → 3 segundos

P3 → 5 segundos

El orden sería:

P2 → P3 → P1

3s 5s 8s

Ventaja: puede reducir considerablemente el **tiempo promedio de espera**.

Desventaja: requiere conocer o estimar cuánto tiempo necesitará cada proceso. Además, los procesos largos pueden sufrir **inanición** si continuamente llegan procesos cortos.

Una variante importante es **SRTF (Shortest Remaining Time First)**, que permite interrumpir un proceso si llega otro con un tiempo restante menor.

Planificación en dos niveles

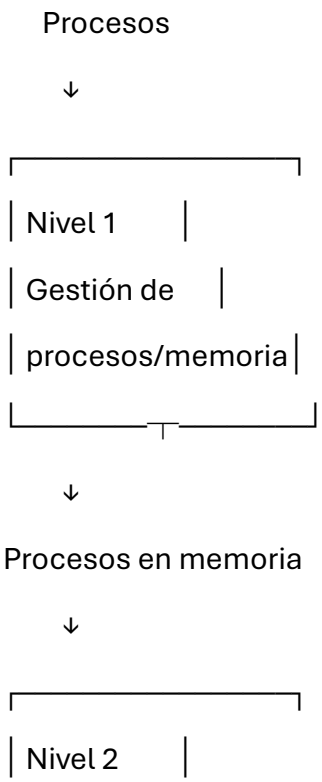
La **planificación en dos niveles** (*Two-Level Scheduling*) se utiliza especialmente cuando el sistema tiene **más procesos de los que puede mantener convenientemente en memoria principal**.

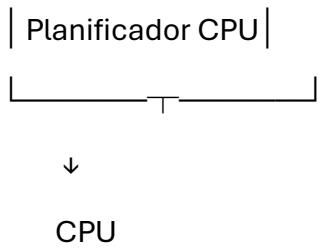
La idea es dividir la planificación en **dos niveles**:

Primer nivel: determina qué procesos pueden permanecer en memoria y competir por la CPU.

Segundo nivel: decide cuál de los procesos que están en memoria obtiene la CPU.

Conceptualmente:





Ventaja: permite administrar eficientemente un número grande de procesos y combinar decisiones relacionadas con **memoria y CPU**.

Desventaja: es más complejo que utilizar un único planificador.

Comparación de los algoritmos

Algoritmo	Idea principal	Ventaja	Desventaja
Round Robin	Cada proceso recibe un turno de CPU.	Equitativo y adecuado para sistemas interactivos.	Muchos cambios de contexto si el quantum es pequeño.
Prioridades	Se ejecutan primero procesos de mayor prioridad.	Permite atender procesos importantes.	Puede producir inanición.
Colas múltiples	Los procesos se separan en diferentes colas.	Organiza procesos según sus características.	Puede existir espera excesiva en colas de baja prioridad.
SJF	Se ejecuta primero el proceso más corto.	Reduce el tiempo promedio de espera.	Es necesario estimar la duración de los procesos.
Dos niveles	Combina gestión de procesos en memoria y planificación de CPU.	Permite manejar muchos procesos.	Mayor complejidad.

Idea fundamental

Todos estos algoritmos buscan resolver el mismo problema:

Hay muchos procesos que necesitan utilizar la CPU, pero el procesador debe decidir cuál atender y en qué orden.

El **Scheduler del Kernel** utiliza políticas de planificación para tomar esas decisiones y lograr un equilibrio entre **rendimiento, tiempo de respuesta, equidad y aprovechamiento de la CPU**.

15. ¿Cómo administra un sistema operativo la memoria y qué problemas busca solucionar mediante intercambio, paginación y memoria virtual?

Investigue:

- Particiones fijas.
- Particiones variables.
- Intercambio.
- Paginación.
- Memoria virtual.
- Relación entre memoria RAM y procesos.

¿Cómo administra un sistema operativo la memoria?

La **gestión de memoria** es una de las funciones fundamentales del sistema operativo. Su objetivo es **administrar la memoria RAM, asignándola y liberándola para los procesos que la necesitan**, además de proteger los espacios de memoria de unos procesos frente a otros.

Los procesos compiten por una cantidad de RAM que es limitada. Por eso, el sistema operativo necesita decidir **qué procesos permanecen en memoria, dónde se ubican y qué partes pueden trasladarse temporalmente al almacenamiento secundario**. (L-Università ta' Malta)

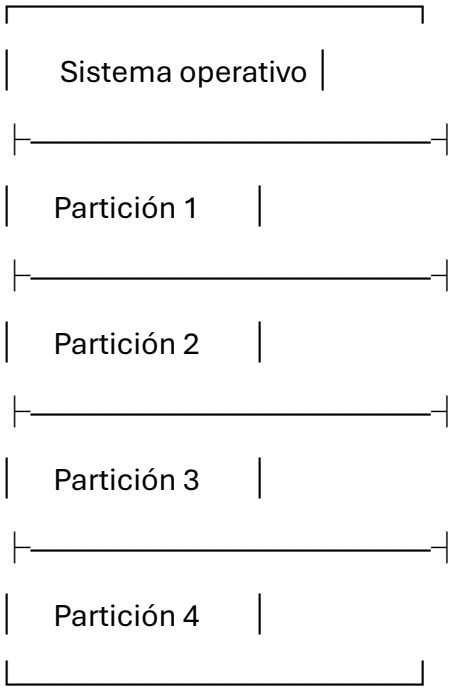
Una forma sencilla de verlo es:

Procesos → Gestión de memoria → RAM → CPU

Particiones fijas

Las **particiones fijas** consisten en dividir la memoria RAM en varias partes de tamaño determinado previamente.

Por ejemplo, una memoria podría dividirse así:

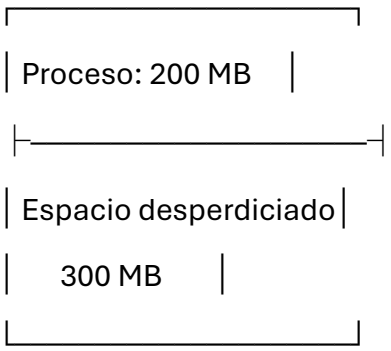


Cada proceso se coloca dentro de una partición disponible.

Problema principal: fragmentación interna

Si un proceso necesita 200 MB pero se coloca en una partición de 500 MB, quedan **300 MB sin utilizar dentro de esa partición**.

Partición: 500 MB



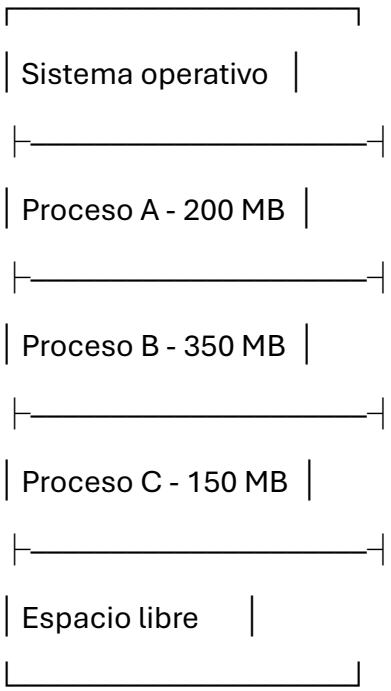
Por tanto, las particiones fijas son sencillas de administrar, pero pueden desperdiciar memoria.

Particiones variables

Las **particiones variables** permiten asignar a cada proceso una cantidad de memoria más cercana a la que realmente necesita.

Por ejemplo:

RAM



Esto reduce el desperdicio interno, pero puede generar **fragmentación externa**: pequeños espacios libres distribuidos entre procesos que, aunque sumados sean grandes, pueden no formar un bloque continuo suficientemente grande. (L-Università ta' Malta)

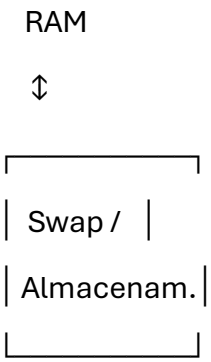
Diferencia fundamental

- **Particiones fijas:** tamaños establecidos previamente.
- **Particiones variables:** tamaño adaptado a las necesidades del proceso.

Intercambio (*Swapping*)

El **intercambio**, o *swapping*, consiste en **mover temporalmente procesos o partes de su memoria entre la RAM y el almacenamiento secundario**, como un SSD o disco.

Por ejemplo:



Si la RAM está muy ocupada, el sistema operativo puede retirar temporalmente información que no está siendo utilizada y recuperarla cuando sea necesaria.

¿Qué problema soluciona?

Principalmente, permite **liberar RAM para otros procesos** cuando la memoria física es limitada. (L-Università ta' Malta)

Sin embargo, el almacenamiento secundario es mucho más lento que la RAM. Por eso, un intercambio excesivo puede disminuir considerablemente el rendimiento.

Paginación

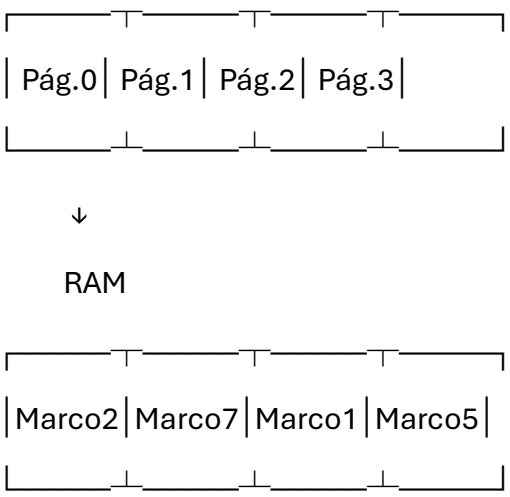
La **paginación** divide:

- La memoria virtual del proceso en **páginas**.
- La memoria física RAM en **marcos de página (*page frames*)**.

Las páginas y los marcos tienen el mismo tamaño. Una página de un proceso puede colocarse en cualquier marco disponible de la RAM. (L-Università ta' Malta)

Por ejemplo:

Proceso



El sistema operativo mantiene una **tabla de páginas** para saber dónde se encuentra cada página del proceso.

¿Qué problema soluciona?

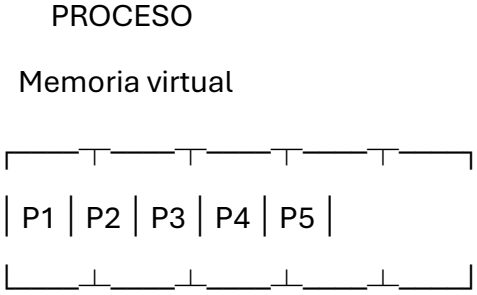
La paginación permite evitar la necesidad de que todo un proceso ocupe un **bloque contiguo** de memoria y reduce significativamente la fragmentación externa. (CS NYU)

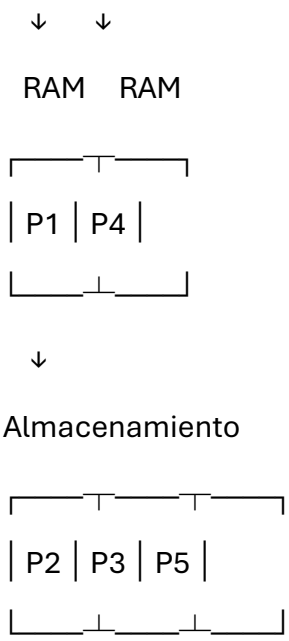
Memoria virtual

La **memoria virtual** permite que cada proceso tenga un **espacio de direcciones virtuales** que no tiene que coincidir directamente con las posiciones físicas de la RAM. El sistema operativo y el hardware realizan la traducción entre las direcciones virtuales y físicas. (D3S)

Esto permite que **no sea necesario mantener todo un proceso en la RAM al mismo tiempo**.

Por ejemplo:





Cuando un proceso intenta acceder a una página que no está actualmente en RAM, se produce un **fallo de página (*page fault*)**. El sistema operativo debe localizar esa página en el almacenamiento y cargarla en memoria física. (OpenStax)

¿Qué problemas busca solucionar?

La memoria virtual permite:

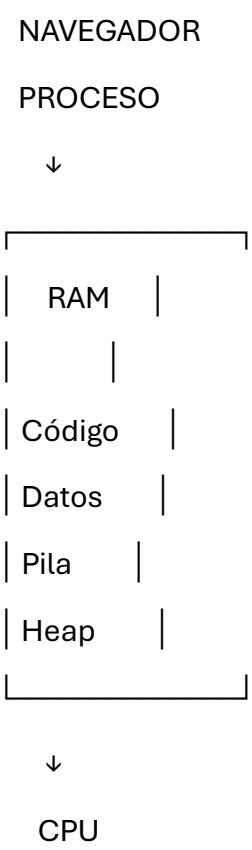
- Ejecutar programas cuyo espacio de memoria virtual es mayor que la RAM disponible.
- Mantener más procesos activos.
- Utilizar la RAM de manera más eficiente.
- Proporcionar aislamiento y protección entre procesos.
- Cargar en RAM solamente las partes de los procesos que se necesitan. (Sistemas Operativos)

Relación entre memoria RAM y procesos

La **RAM es el espacio de trabajo principal de los procesos**.

Cuando ejecutamos un programa, el sistema operativo necesita cargar en memoria las instrucciones y datos necesarios para que la CPU pueda trabajar con ellos.

Por ejemplo, si abrimos un navegador:



Si tenemos muchos programas abiertos, **varios procesos compiten por la RAM**. El sistema operativo administra esa memoria y decide qué partes deben permanecer en ella y cuáles pueden mantenerse temporalmente fuera de la RAM. (L-Università ta' Malta)

Comparación general

Técnica	¿Cómo funciona?	Problema que busca solucionar
Particiones fijas	Divide la RAM en bloques predeterminados.	Administración sencilla de memoria.

Técnica	¿Cómo funciona?	Problema que busca solucionar
Particiones variables	Asigna bloques según las necesidades de cada proceso.	Reducir desperdicio interno.
Intercambio	Mueve información entre RAM y almacenamiento.	Liberar RAM cuando no es suficiente.
Paginación	Divide procesos en páginas y RAM en marcos.	Evitar la necesidad de memoria contigua y reducir fragmentación externa.
Memoria virtual	Permite utilizar un espacio de direcciones mayor que la RAM física disponible.	Ejecutar más procesos y aprovechar mejor la memoria.

Ejemplo integrador

Imaginemos que un computador tiene **8 GB de RAM**, pero el usuario abre simultáneamente un navegador, un videojuego, Word, un editor de imágenes y otros programas.

La RAM puede comenzar a llenarse. El sistema operativo puede utilizar **paginación y memoria virtual** para mantener en RAM las partes activas de los procesos y trasladar otras páginas al almacenamiento cuando sea necesario. (OpenStax)

En conclusión: la gestión de memoria busca que los procesos tengan el espacio que necesitan, que estén protegidos entre sí y que la RAM se utilice eficientemente. Las técnicas evolucionan desde modelos sencillos como **particiones fijas y variables** hasta mecanismos modernos como **paginación, intercambio y memoria virtual**.