FACITEC

SISTEMA OPERATIVO 2026 SDG

Luis Torres

Índice de Contenidos

Sistema Operativo

Este material constituye el núcleo técnico de la asignatura **Sistema Operativo**, dictada en la **FACITEC – UNICAN** (Sede Saltos del Guairá). El programa académico (Plan 2024) contempla una carga de **128 horas totales**, distribuidas equitativamente entre trabajo presencial y actividades independientes.

> ℹ Propósito del Material
>
> Este recurso centraliza lecciones teóricas, guías de laboratorio y referencias bibliográficas para su consulta interactiva o en formato PDF offline.

> 💡📩 Material de Lectura
>
> **Descargar PDF del Libro**

Competencias Específicas

1. **Fundamentos**: Comprender con solidez los procesos, hilos, planificación, memoria y archivos para interpretar su interacción en el entorno informático.
2. **Gestión de Procesos**: Analizar la creación, ejecución y sincronización de procesos para aplicar técnicas eficientes de asignación de recursos.
3. **Administración de Memoria**: Explicar técnicas de particiones, paginación y segmentación para interpretar la gestión del espacio operativo.
4. **Entrada/Salida**: Describir mecanismos de interrupciones, controladores y buffers, analizando su impacto en la eficiencia del sistema.
5. **Sistemas de Archivos**: Interpretar la organización de archivos, directorios y metadatos en bloques de almacenamiento.
6. **Memoria Caché**: Explicar el uso de buffers y sincronización de datos para la mejora del rendimiento en operaciones de E/S.
7. **Memoria Virtual**: Interpretar la traducción de direcciones y algoritmos de reemplazo de páginas en sistemas modernos.
8. **Seguridad**: Aplicar mecanismos de protección, control de acceso y auditoría para salvaguardar la integridad de los recursos.

Metodología

El curso se desarrolla mediante clases magistrales, resolución de problemas técnicos y prácticas en sala de máquinas. La evaluación se basa en fichas de trabajo individuales y grupales que incluyen modelado

de procesos, algoritmos de planificación y auditorías de gestión. Además se de los examenes parciales y un examen finales contemplados en el curso.

I

I. Conceptos fundamentales de los Sistemas Operativos

1. Introducción a los Sistemas Operativos

«Un sistema operativo es como un gobierno. No realiza ninguna función útil por sí mismo, sino que proporciona el entorno dentro del cual otros programas pueden hacer un trabajo útil.» — Abraham Silberschatz.

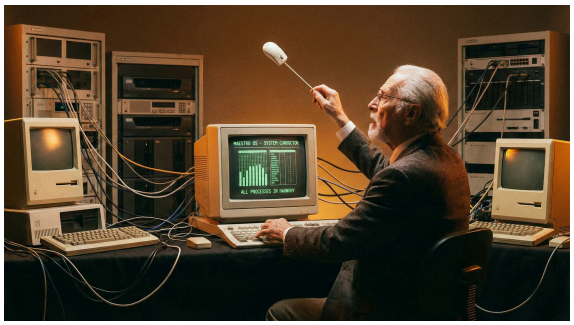


Figura 1.1: Un sistema operativo es como un director de orquesta

Imaginen una computadora moderna sin software. Sería, en esencia, un costoso pisapapeles de metal y silicio. El hardware A. S. Tanenbaum y H. Bos [1] (procesador, memoria, discos) tiene un potencial inmenso, pero carece de voluntad propia. Necesita un **director de orquesta** que le indique cuándo tocar, qué instrumento usar y con qué intensidad. Ese director es el **Sistema Operativo (SO)**.

1.1 Definición y Doble Naturaleza

El Sistema Operativo es el programa fundamental que actúa como intermediario entre el usuario y el hardware. Sin embargo, su definición cambia radicalmente según quién la formule. Esta es su **naturaleza dual**.

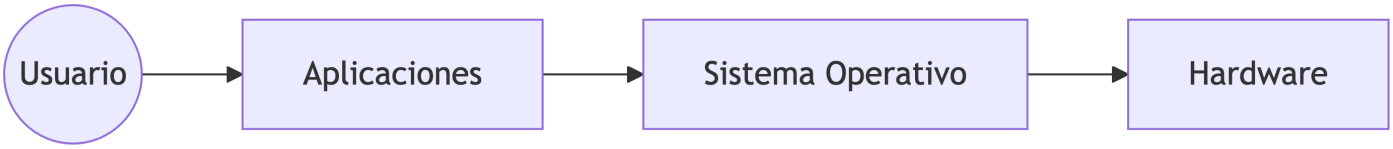


Figure 1.1: El Sistema Operativo como intermediario entre el usuario y el hardware.

1.2 La Perspectiva del Usuario: Abstracción y Conveniencia

Para nosotros, los usuarios y programadores, el hardware «desnudo» es complejo, incómodo y hostil. Lidiar directamente con los chips, discos y cables requiere conocer detalles técnicos abrumadores y propensos a errores. Aquí es donde el Sistema Operativo realiza su función más «humana»: la **Abstracción**.

¿Qué hace realmente?

El SO actúa como un traductor y embellecedor. Oculta la «fealdad» y complejidad de los componentes físicos (como tener que mover un brazo mecánico en el disco duro para leer un dato) y nos presenta interfaces limpias, elegantes y fáciles de entender

La Transformación: Convierte lo difícil en conceptos cotidianos.

En lugar de lidiar con sectores, pistas y motores de disco, el SO nos da «archivos» (como fotos o documentos) que podemos abrir y guardar fácilmente.

En lugar de gestionar la memoria física y direcciones numéricas, nos permite abrir múltiples «ventanas» y programas simultáneamente sin preocuparnos por dónde se guardan sus datos en el chip de RAM.

Definición: Punto de vista del usuario

Desde esta perspectiva, el Sistema Operativo no es solo un conjunto de programas, sino **una capa de software cuya misión es ocultar la complejidad del hardware** y ofrecernos un entorno amigable, consistente y conveniente para trabajar. A. Silberschatz, P. B. Galvin, y G. Gagne [2]

1.3 La Perspectiva del Sistema: El «Gestor de Recursos» (Visión Bottom-Up)

Si la visión anterior era «hacia abajo» (desde la aplicación hacia el hardware), esta es la visión «desde abajo». Para el sistema operativo, la computadora no es más que un conjunto de componentes electrónicos (procesadores, memorias, discos, interfaces de red) que deben administrarse.

El Problema del Caos:

Imaginemos qué pasaría si tres programas intentaran imprimir a la vez en la misma impresora sin un intermediario. Las líneas de texto del Programa 1 se mezclarían con las del Programa 2 y 3, resultando en un caos absoluto de papel inútil. El SO evita esto imponiendo un orden: almacena los datos en un búfer (spooling) y decide quién imprime primero.

La Estrategia de Gestión (Multiplexación):

Para gestionar la competencia por recursos limitados, el SO utiliza una técnica llamada multiplexación, que consiste en compartir un recurso de dos formas distintas:

`1. Multiplexación en el Tiempo (Turnos):`

Se utiliza para recursos que solo puede usar uno a la vez, como la CPU o la impresora. El SO decide quién lo usa, por cuánto tiempo, y quién sigue después.

`2. Multiplexación en el Espacio (División):`

Se usa para recursos que se pueden dividir, como la memoria RAM o el disco duro. En lugar de turnarse, cada programa obtiene una «parcela» del recurso para usarla simultáneamente con otros.

ℹ Definición: El SO según la perspectiva del sistema

El Sistema Operativo actúa como un asignador de recursos (Resource Allocator). Su objetivo es resolver conflictos entre solicitudes concurrentes para garantizar dos cosas:

1. **Eficiencia:** Que el hardware (como la CPU) esté ocupado la mayor parte del tiempo posible, evitando tiempos muertos.
2. **Equidad:** Que todos los programas reciban su parte justa de recursos y ninguno monopolice el sistema en detrimento de los demás.

1.4 El Núcleo del Asunto: Kernel vs. Sistemas

Es vital distinguir entre el sistema operativo propiamente dicho y el resto del software.

- **El Kernel (Núcleo):** Es el programa que se ejecuta **siempre**, desde que encendemos la computadora hasta que la apagamos. Tiene control total sobre todo lo que ocurre en el sistema.
- **Programas de Sistema:** Son programas que vienen con el SO (como el explorador de archivos, el compilador o la calculadora) pero no son parte del kernel. Si se cierran, el sistema sigue funcionando.

Curiosidad: ¿El huevo o la gallina? (Bootstrapping)

Si el sistema operativo es un programa que necesita ser cargado en memoria para funcionar, pero la memoria está vacía al encender la PC, ¿quién carga al sistema operativo?

Este dilema se resuelve con el proceso de **Bootstrapping**:

1. Al encender, un pequeño chip (ROM/UEFI) ejecuta un código rudimentario.
2. Este código busca el **Bootloader** en el primer sector del disco.
3. El Bootloader localiza el **Kernel** del SO y lo carga en memoria.
4. El Kernel toma el control y el sistema «cobra vida».

1.5 Servicios Fundamentales

Para cumplir sus roles, el SO ofrece servicios específicos. Podemos verlos como departamentos de una empresa:

1. **Gestión de Procesos (RRHH):** Decide qué programa usa la CPU. Pausa, reanuda y elimina tareas.
2. **Gestión de Memoria (Logística):** Asigna parcelas de RAM a cada programa. Evita que un juego invada la memoria de tu procesador de texto.
3. **Gestión de Archivos (Archivo):** Convierte sectores magnéticos o celdas en una jerarquía lógica.
4. **Gestión de E/S (Comunicaciones):** Habla con el teclado, mouse, pantalla e impresora, gestionando señales (interrupciones).

1.6 Evolución: Una Historia de Eficiencia

La evolución de los sistemas operativos es, en esencia, una carrera constante por resolver un problema económico: el hardware era extremadamente costoso y los humanos muy lentos. El objetivo siempre fue evitar que la CPU (el componente más caro) perdiera tiempo «sin hacer nada».

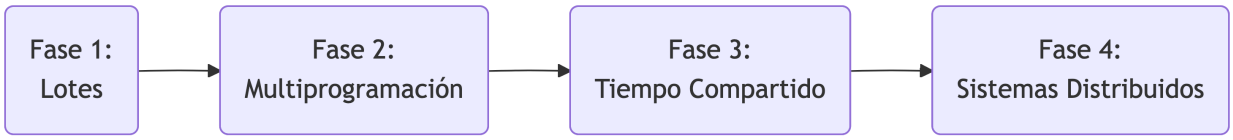


Figure 1.2: Línea de tiempo simplificada de la evolución de los sistemas operativos.

1.6.1 Fase 1: Procesamiento por Lotes (Batch Processing)

¿Qué es?

El procesamiento por lotes es una técnica donde los trabajos (programas) se agrupan en conjuntos o «lotes» similares y se ejecutan secuencialmente, uno tras otro, sin intervención manual entre ellos. **El Problema que resolvió:**

En las primeras computadoras (años 50), el programador tenía que reservar la máquina, cargar su programa manualmente, ejecutarlo y depurarlo. Si se detenía a pensar qué hacer, la máquina estaba ociosa. Esto desperdiciaba muchísimo tiempo de CPU. **La Solución:**

Se eliminó la interacción directa del usuario con la máquina. Los programadores entregaban sus tarjetas perforadas a un operador, quien las agrupaba en una cinta magnética. Un pequeño programa monitor (el ancestro del SO) leía el primer trabajo, lo ejecutaba, y al terminar, cargaba automáticamente el siguiente. Esto reducía drásticamente el tiempo muerto entre trabajos.

> ! Hito Histórico: GM-NAA I/O
>
> En 1956, General Motors creó el que se considera el primer sistema operativo básico para su IBM 704. Su única función era pasar automáticamente al siguiente trabajo.

Limitación:

Aunque mejoró el uso de la CPU, tenía un defecto grave: si un trabajo necesitaba leer datos de una cinta (una operación lenta), la CPU se detenía por completo a esperar. No podía hacer nada más hasta que terminara la lectura.

1.6.2 Fase 2: Multiprogramación

¿Qué es?

La multiprogramación es la técnica que permite tener múltiples procesos (programas) cargados en la memoria principal al mismo tiempo, listos para ejecutarse. El sistema operativo cambia rápidamente la atención de la CPU entre estos trabajos para mantenerla siempre ocupada. * **El Problema que resolvió:** Se diseñó para solucionar la ineficiencia del procesamiento por lotes. Como la CPU es miles de veces más rápida que los dispositivos de E/S (como discos o impresoras), en un sistema por lotes la CPU pasaba el 80-90% del tiempo inactiva esperando a que terminaran operaciones de lectura/escritura.

Cómo funciona (Contexto):

Imagina que tienes tres programas en memoria: A, B y C.

1. El SO comienza a ejecutar el Programa A.
2. De repente, el Programa A necesita leer un archivo del disco. Como esto tarda «una eternidad» (en tiempo de computadora), en lugar de dejar la CPU inactiva esperando, el SO bloquea al Programa A.
3. Inmediatamente, el SO asigna la CPU al Programa B.
4. Si el Programa B también tiene que esperar por algo (ej. una impresión), la CPU salta al Programa C.
5. Cuando el disco termina de leer los datos del Programa A, este vuelve a estar listo para usar la CPU.

- **Resultado:** La utilización de la CPU mejora drásticamente, acercándose al 100% en condiciones ideales, ya que siempre hay «alguien» dispuesto a usar el procesador mientras otros esperan por dispositivos lentos.

1.6.3 Fase 3: Tiempo Compartido (Time-Sharing)

- **¿Qué es?** Es una extensión lógica de la multiprogramación diseñada para la interacción directa con el usuario. Permite que múltiples usuarios usen la computadora simultáneamente, dándoles la ilusión de que tienen la máquina dedicada solo para ellos.

- **Diferencia clave:** Mientras la multiprogramación busca maximizar el uso de la CPU, el tiempo compartido busca minimizar el tiempo de respuesta. La CPU cambia entre los programas de los usuarios tan rápido (cada pocos milisegundos) que parece que todos se ejecutan a la vez (**Quantum**).

1.6.4 Fase 4: Sistemas Distribuidos y Paralelos

Hoy, la computación ha trascendido un solo chasis.

- **Sistemas Paralelos (Multiprocesadores):** Son sistemas que tienen más de una CPU compartiendo la misma memoria física y reloj. Permiten la ejecución verdadera simultánea de procesos, aumentando la confiabilidad (si falla una CPU, las otras siguen) y la velocidad.
- **Sistemas Distribuidos:** Consisten en múltiples computadoras separadas, cada una con su propia memoria y procesador, conectadas por una red. Para el usuario, el sistema parece una sola computadora potente y unificada, aunque en realidad el trabajo se reparte entre muchas máquinas remotas. Esto permite resolver problemas masivos que una sola máquina no podría manejar (La Nube).

1.7 Arquitectura Moderna: Cliente-Servidor

Los sistemas operativos monolíticos son difíciles de mantener. La tendencia moderna es la **arquitectura modular**, inspirada en el modelo Cliente-Servidor.

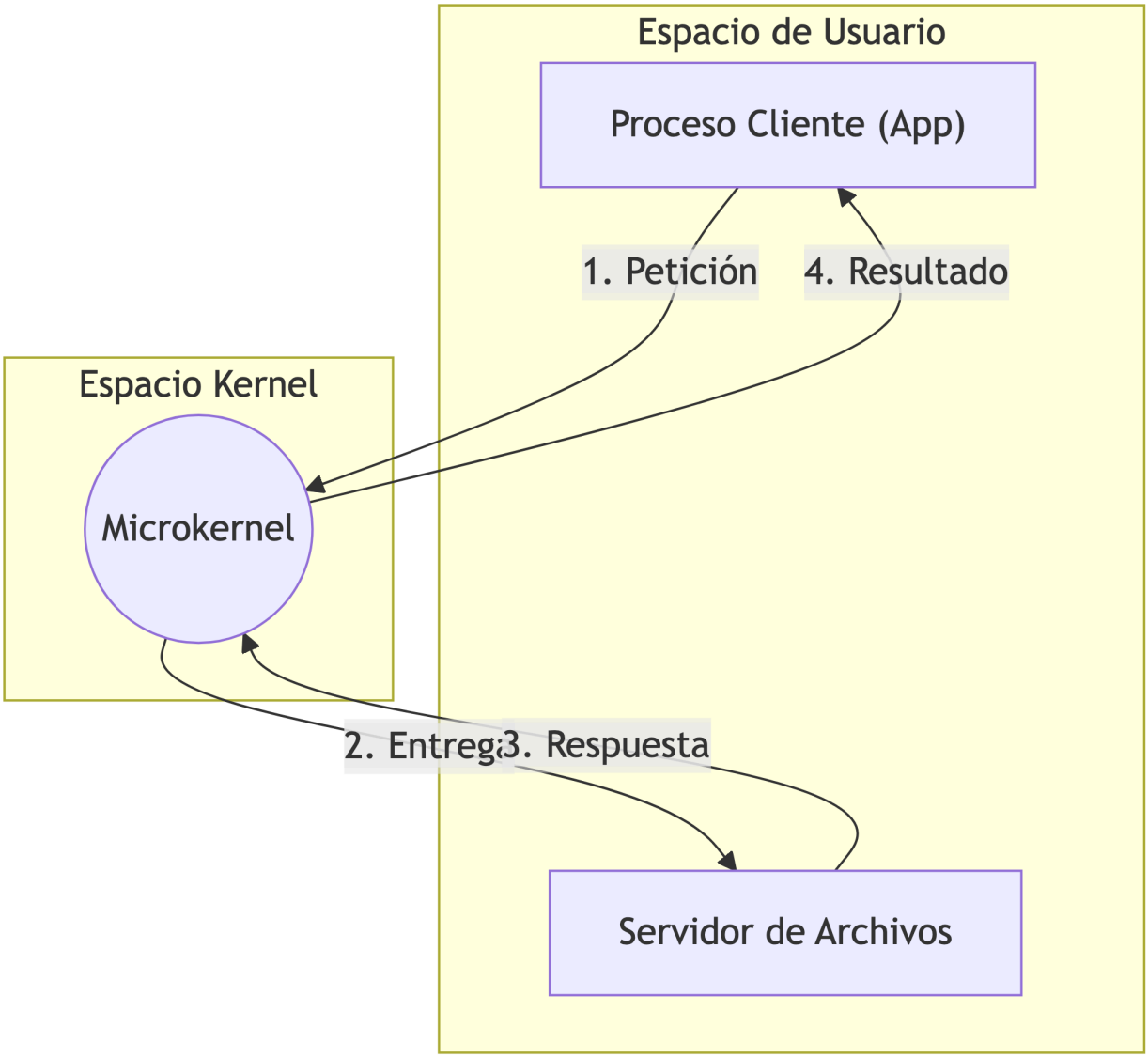


Figure 1.3: Arquitectura de Microkernel basada en el modelo Cliente-Servidor.

- **Microkernels:** Se intenta sacar lo máximo posible del Kernel y moverlo al espacio de usuario.
- **Ventaja:** Si el «Servidor de Archivos» falla, el Kernel lo reinicia sin que la máquina se cuelgue.
- **RPC (Llamada a Procedimiento Remoto):** Es el pegamento. Permite que un programa llame a una función en otro proceso como si fuera local.

2. Evolución y Arquitectura de los Sistemas Operativos

La arquitectura de los Sistemas Operativos (SO) es el resultado de una evolución histórica marcada por la necesidad de optimizar el uso del hardware. Cada salto generacional en la electrónica ha dictado el diseño del software de control, transitando desde la gestión manual de componentes hasta la abstracción total de servicios en la nube.

2.1 Evolución Histórica de los Sistemas Operativos

Las primeras máquinas de cómputo carecían de SO; eran dispositivos de propósito único del tamaño de habitaciones completas. Su evolución se analiza a través de las generaciones de hardware.

2.1.1 Primera Generación (1945–1955): Tubos de Vacío

Figure 2.1: ENIAC, la primera computadora electrónica de propósito general

- **Hardware:** Máquinas voluminosas construidas con tubos de vacío que consumían cantidades masivas de energía.
- **Operación:** No existían lenguajes de alto nivel ni SO. La ejecución era rudimentaria: los ingenieros programaban directamente en lenguaje de máquina mediante tableros de conexiones o tarjetas perforadas. El programador era también el operador físico de la máquina.

2.1.2 Segunda Generación (1955–1965): Transistores y Procesamiento por Lotes

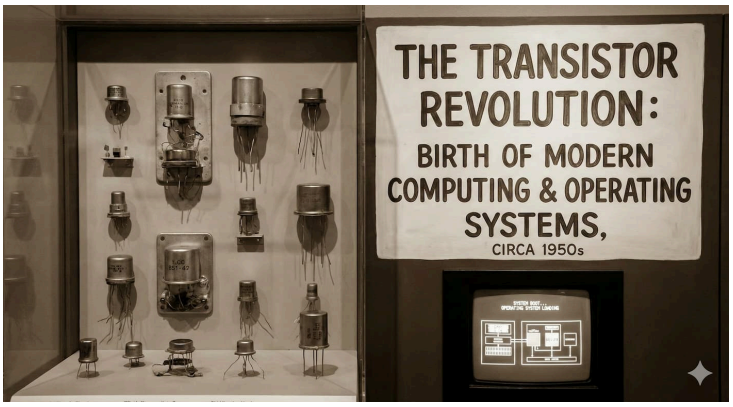


Figure 2.2: Los transistores revolucionaron la computación, reduciendo el tamaño y el consumo energético

- **Hito de Hardware:** El **Transistor**. Permitió equipos más pequeños, rápidos, confiables y económicos.
- **Hito de Software:** En 1957 surge **FORTRAN** (IBM), revolucionando el acceso para científicos.
- **Nacimiento del SO (Procesamiento por Lotes):** Para maximizar el costoso tiempo de CPU, se implementó la técnica de *batch processing*.
- **Flujo de Trabajo:** El programador entregaba sus tarjetas perforadas a un operador. Este agrupaba los trabajos y los cargaba secuencialmente.
- **Monitor Residente:** Antecesor directo del núcleo moderno, cuya función era cargar automáticamente el siguiente trabajo tras finalizar el anterior, eliminando el tiempo muerto de intervención humana.

2.1.3 Tercera Generación (1965–1980): Circuitos Integrados (IC)

- **Hardware:** Integración de múltiples transistores en un solo chip de silicio. Mayor potencia a menor costo (ej. Familia IBM 360).
- **Paradigmas del SO:**
  - **Multiprogramación:** Capacidad de mantener varios trabajos en memoria simultáneamente. Si un proceso esperaba por E/S, la CPU saltaba a otro trabajo, optimizando el ciclo de procesamiento.
  - **Tiempo Compartido (Time-Sharing):** Permitió que múltiples usuarios interactuaran con la máquina de forma concurrente mediante terminales, creando la ilusión de una máquina dedicada para cada uno.
- **Estandarización:** COBOL se popularizó en el sector empresarial, ampliando el alcance comercial de la informática.

2.1.4 Cuarta Generación (1980–Presente): Microprocesadores y PCs

- **Hardware:** El microprocesador integra la CPU completa en un solo chip (LSI/VLSI).
- **Evolución del SO:**
  - **Interfaces Gráficas (GUI):** El enfoque se desplazó hacia la usabilidad (Windows, macOS). La multitarea se convirtió en estándar.
  - **Conectividad y Redes:** El SO se volvió el nodo de acceso a Internet y, posteriormente, a la computación en la nube.

– **Ubicuidad:** Surgimiento de dispositivos móviles (smartphones/tablets) y la integración de Inteligencia Artificial como servicio del sistema.

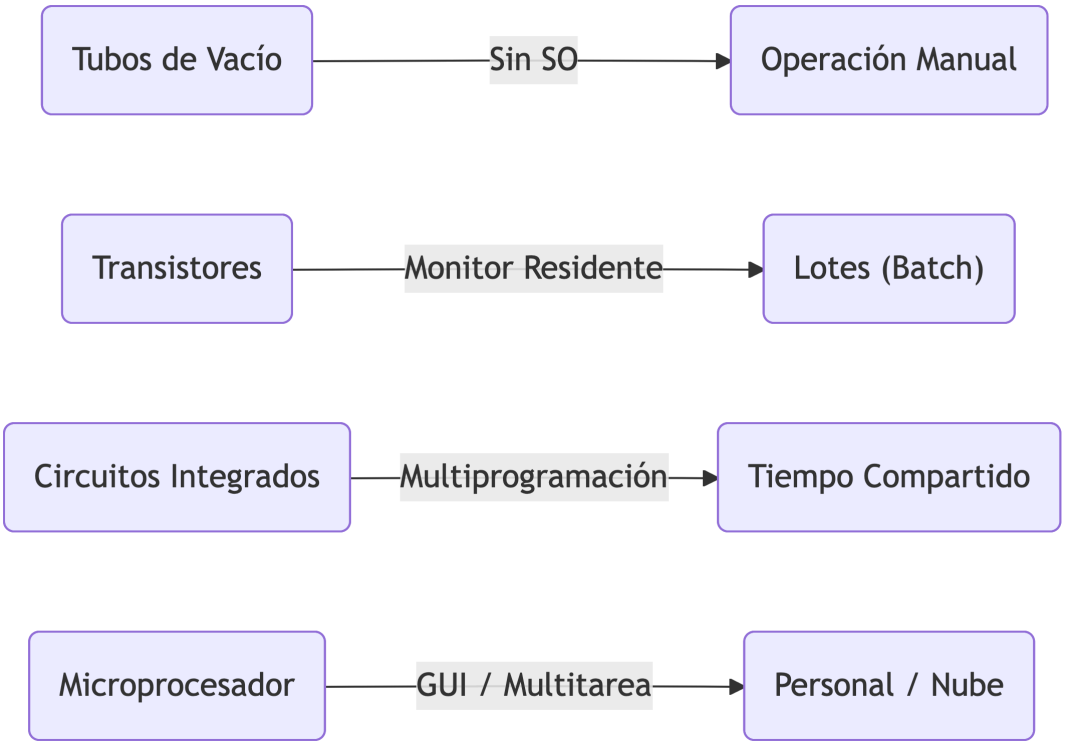


Figure 2.3: Relación entre hitos de hardware y paradigmas de sistemas operativos.

2.2 Estructura de los Sistemas Operativos

La organización interna del núcleo (kernel) define su robustez y eficiencia.

2.2.1 Sistemas Monolíticos

El SO se compila como un único programa que corre en **modo núcleo**. Todos los componentes (gestión de memoria, archivos, planificación) comparten el mismo espacio de direcciones.

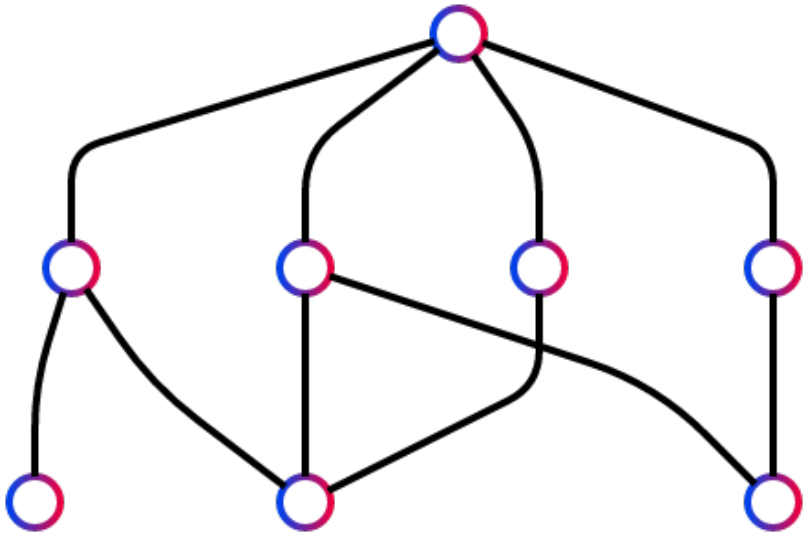


Figure 2.5: Arquitectura de un Sistema Monolítico

Figure 2.4: En los sistemas monolíticos, no existe una separación clara entre los servicios del sistema operativo.

- **Ventaja:** Alto rendimiento por llamadas directas a funciones.
- **Desventaja:** Un fallo en cualquier módulo (ej. un controlador) compromete la estabilidad de todo el sistema.

2.2.2 Sistemas de Microkernel

Se reduce el núcleo al mínimo (gestión de memoria básica, hilos e IPC). Los servicios tradicionales del SO corren como procesos de usuario independientes.

- **Ventaja:** Alta tolerancia a fallos y modularidad.
- **Desventaja:** Sobrecarga de rendimiento debido al paso de mensajes entre espacios de usuario y núcleo.

2.3 Estructura en Capas (Layered Structure)

Organiza el SO como una jerarquía de niveles. Cada capa se construye sobre la inferior y solo utiliza los servicios que esta le provee.

2.3.1 El Modelo «THE» (Dijkstra, 1968)

Este sistema pionero dividió las funciones del SO para garantizar la corrección lógica del diseño:

Table 2.1: Capas del sistema operativo THE según Dijkstra

Capa	Función	Responsabilidad
5	Usuario	Ejecución de aplicaciones finales.
4	Gestión de E/S	Manejo de flujos de datos (buffering) para periféricos.
3	Consola	Comunicación bidireccional entre operador y procesos.
2	Memoria	Gestión de la memoria virtual y almacenamiento masivo.
1	Planificación	Asignación de tiempo de CPU y multiprogramación.
0	Hardware	Gestión de interrupciones y estados físicos del procesador.

i Resumen de Estructuras

La elección de una arquitectura depende del equilibrio entre **rendimiento** (Monolítico) y **fiabilidad/seguridad** (Microkernel). La mayoría de los sistemas operativos modernos (como Windows NT o macOS) utilizan enfoques **híbridos**.

2.4 El Kernel (Núcleo) en los sistemas operativos

El kernel o núcleo puede considerarse el corazón mismo del sistema operativo, actuando como el intermediario de más bajo nivel entre el hardware físico y el software de interfaz (como el shell o las aplicaciones).

Para comprender su profundidad, es crucial entender su entorno de ejecución basado en una dualidad de modos de protección proporcionada por el hardware:

1. **Modo Kernel (o Modo Supervisor):**

   Es un estado de ejecución donde el software tiene un acceso ilimitado y completo a todos los recursos del procesador y la memoria. En este modo se pueden ejecutar «instrucciones privilegiadas» que afectan el control de la máquina, como administrar el hardware, gestionar interrupciones o alterar los límites de seguridad. El kernel es la pieza de software que opera exclusivamente bajo este modo.

2. **Modo Usuario:**

   Es el estado restringido en el que se ejecutan los programas de aplicación regulares. Tienen prohibido ejecutar instrucciones que interactúen directamente con el hardware y solo tienen acceso a un espacio de memoria confinado.

El kernel contiene los componentes esenciales para administrar los recursos del sistema: el **planificador de procesos (scheduler), el gestor de memoria, el gestor de entrada/salida (I/O) y el administrador de comunicación interprocesos**.

La forma en que se estructura este kernel define la arquitectura del sistema: en un sistema monolítico, todo el sistema operativo es un único gran programa que corre en modo kernel, lo que lo hace muy eficiente pero vulnerable, ya que un error puede colgar toda la máquina.

Como alternativa, arquitecturas como el Microkernel mantienen en el modo supervisor solo las funciones más primitivas (como el manejo básico de memoria y la comunicación interprocesos), trasladando la mayoría de los servicios a procesos que se ejecutan en modo usuario para maximizar la fiabilidad y evitar caídas totales.

2.5 Estructura Cliente-Servidor y mecanismos básicos de control

2.5.1 La Estructura Cliente-Servidor:

Esta arquitectura es una evolución conceptual de los sistemas operativos donde se dividen los procesos en dos clases: los servidores, que proporcionan servicios específicos (como un servidor de archivos o un servidor de procesos), y los clientes, que utilizan estos servicios. En lugar de realizar llamadas directas al sistema operativo, un cliente solicita un servicio construyendo un mensaje y enviándolo al servidor, el cual realiza el trabajo y devuelve la respuesta. Este modelo brilla especialmente por su extensibilidad a sistemas distribuidos, ya que para el cliente es transparente si el servidor está en su misma máquina o al otro lado de una red. A nivel técnico, esto se implementa frecuentemente mediante RPC (Llamadas a Procedimientos Remotos), un mecanismo que empaqueta (marshaling) los parámetros de una solicitud para que llamar a un servicio remoto parezca exactamente igual a ejecutar una función local.

2.5.2 Mecanismos de Control:

Llamadas, Interrupciones, Excepciones y Eventos:

Para que el software interactúe con el hardware y el kernel de manera ordenada y segura, se utilizan distintas transferencias de control:

Llamadas al Sistema (System Calls):

Son el puente controlado mediante el cual un programa en modo usuario solicita un servicio del kernel. El programa coloca los parámetros requeridos en registros y ejecuta una instrucción especial de Trampa (Trap). Esto detiene momentáneamente al programa, cambia el procesador a modo kernel y salta a una dirección de memoria fija para que el sistema operativo ejecute la acción solicitada de forma segura antes de devolver el control al usuario.

Interrupciones (Interrupts):

Son señales generadas principalmente por el hardware (de manera asíncrona) para avisar al procesador que ha ocurrido un evento importante, como la llegada de datos de un disco o presionar una tecla. Al recibir una interrupción, el hardware detiene inmediatamente el proceso actual, guarda su estado (su contador de programa y registros) e inicia la ejecución de una Rutina de Servicio de Interrupción (ISR).

Excepciones y Trampas (Traps):

Aunque a veces se usan indistintamente con las interrupciones, estas son síncronas (generadas por el software). Una trampa suele ser un salto intencional al kernel (como la llamada al sistema), mientras que una excepción ocurre de forma no planificada debido a un error crítico o a una violación de protección, como intentar dividir por cero o acceder a memoria no autorizada.

2.6 Máquinas Virtuales

La virtualización es una tecnología que consiste en crear un clon casi exacto del hardware físico de la máquina en un entorno de software. Su propósito es aislar recursos y permitir que múltiples sistemas operativos completamente diferentes convivan simultáneamente en un mismo computador, creyendo cada uno que posee control total sobre los procesadores, discos y memoria de la máquina.

El corazón de la virtualización es el Hipervisor (o Monitor de Máquina Virtual).

Este componente es el único que opera con los privilegios reales más altos de la máquina. Existen dos tipos principales:

Hipervisor Tipo 1: (Bare Metal)

Corre directamente sobre el hardware físico (bare metal), proporcionando y administrando las máquinas virtuales a las que se les instalan los «sistemas operativos invitados» (guests).

Hipervisor Tipo 2 (Hosted):

Corre como una aplicación estándar sobre un sistema operativo anfitrión (Host), dependiendo de él para interactuar con los dispositivos físicos (ej. VMware Workstation).

¿Cómo funciona la ilusión?

Dado que un sistema operativo invitado debe poder ejecutar instrucciones críticas (como modificar la memoria virtual), el hipervisor debe interceptar estas acciones para mantener el control. Cuando el SO invitado ejecuta una instrucción «sensible», el procesador genera un trap (trampa) hacia el hipervisor, el cual verifica la instrucción y la emula como si el hardware real la hubiera ejecutado (enfoque de atrapar y emular).

Para optimizar el altísimo costo de procesamiento que requiere emular instrucciones constantemente, han surgido dos caminos:

Paravirtualización:

Modifica el código fuente del sistema operativo invitado para que, en lugar de intentar usar instrucciones prohibidas, haga hypercalls (llamadas directas al hipervisor) pidiendo los recursos directamente.

Virtualización a Nivel de SO (Contenedores):

Una alternativa a las máquinas virtuales puras. En vez de ejecutar múltiples sistemas operativos completos manejados por un hipervisor, el kernel de un único sistema operativo particiona sus propios recursos para crear múltiples entornos de usuario aislados llamados «contenedores». Son mucho más ligeros y eficientes en rendimiento, pero con la limitante de que no pueden ejecutar sistemas operativos distintos (todos comparten el kernel anfitrión) y su aislamiento de seguridad es menos absoluto frente a fallos del sistema base.

II

II. Procesos de los Sistemas Operativos

3. Procesos de sistemas operativos

3.1 Procesos

Uno de los conceptos más fundamentales en la gestión de un sistema operativo es el de proceso. Este concepto se originó con la llegada de la **multiprogramación**, la cual permitió la ejecución simultánea de múltiples programas en un mismo sistema. Gracias a esta capacidad, los sistemas operativos pudieron transformar la forma en que gestionan los recursos y coordinan las tareas en un ordenador.

Consideremos el caso de una computadora portátil moderna. Desde el momento en que se enciende, el sistema operativo inicia una serie de procesos en segundo plano, muchos de los cuales el usuario ni siquiera nota. Por ejemplo, se ejecutan procesos como el explorador de Windows, servicios esenciales (red, impresión), programas de seguridad (antimalware), servidores de bases de datos, o incluso la Máquina Virtual de Java (JVM) cuando el entorno está configurado para desarrollo con Java. Mientras tanto, el usuario puede estar navegando por Internet o escuchando música, sin percibir la complejidad de las operaciones que ocurren simultáneamente.

Toda esta actividad se gestiona de manera eficiente gracias a la multiprogramación, que permite al sistema operativo administrar los recursos del procesador, la memoria y otros dispositivos de forma que varios procesos puedan ejecutarse «al mismo tiempo» sin interferir entre sí.

Podemos definir, entonces, a un proceso como:

> Proceso
>
> **Una instancia de un programa en ejecución.**

Es importante destacar que un proceso no es simplemente un programa almacenado en disco o en memoria, sino una entidad activa que utiliza los recursos del sistema para realizar tareas específicas. Cada proceso dispone de su propio espacio de memoria, un estado determinado y es gestionado por el sistema operativo para garantizar que todos puedan operar de forma eficiente y segura dentro del sistema.

3.2 Estructura de un proceso: el PCB

La **estructura** de un proceso se representa comúnmente mediante el Process Control Block (PCB), que es una estructura de datos que almacena toda la información relevante para la gestión del proceso. Entre los elementos contenidos en un PCB destacan:

- **Código del programa**

  Las instrucciones que el proceso ejecutará.

- **Datos**

  Las variables necesarias para la ejecución.

- **Recursos del sistema**

  CPU, memoria, dispositivos de E/S, que el proceso necesita para funcionar.

Para entender mejor el modelo del proceso, podemos utilizar la siguiente analogía de un taller mecánico:

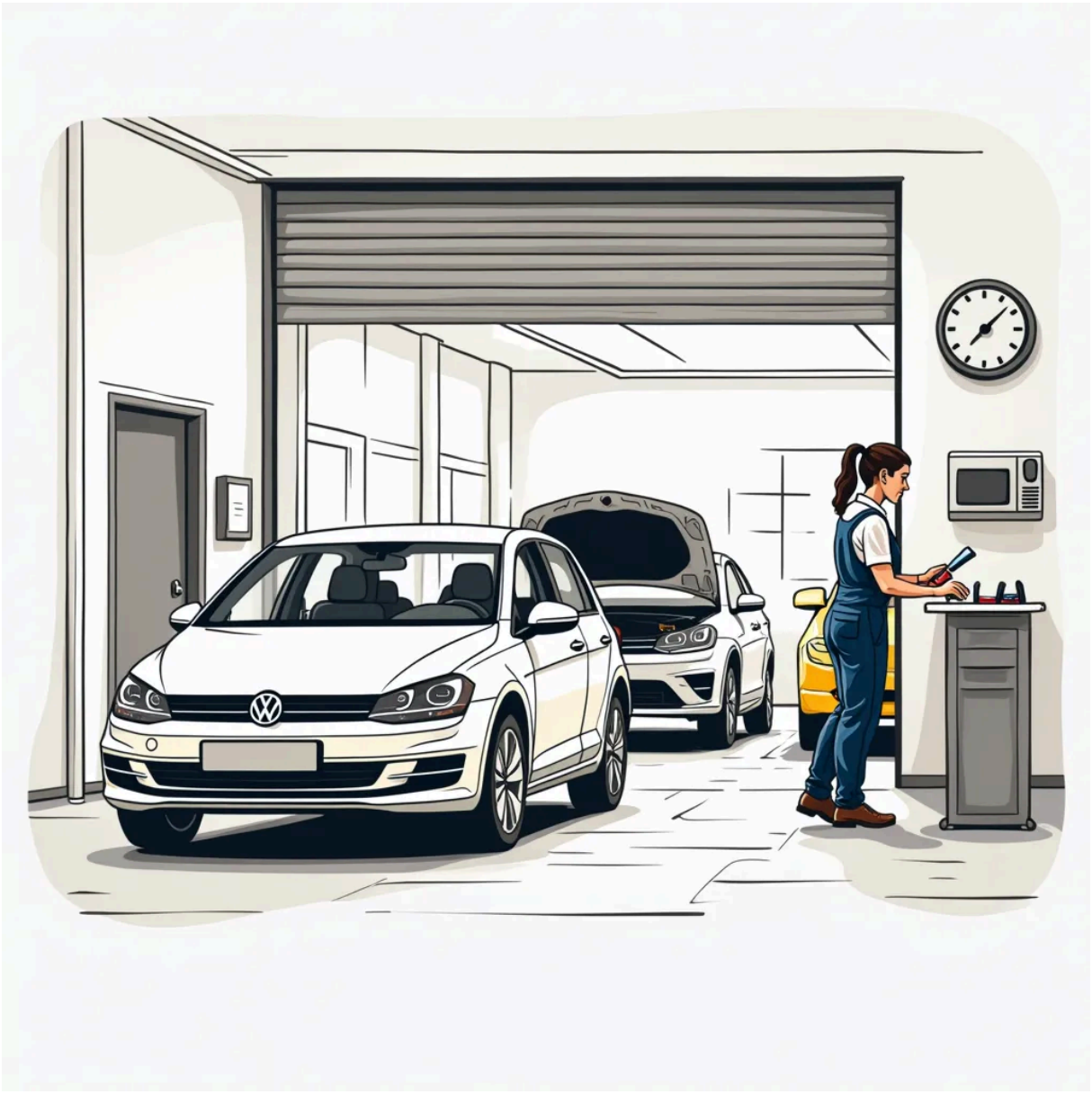


Figure 3.1: La analogía del taller mecánico

1. Vehículo en el Taller (Proceso)

Imagina que cada proceso es un vehículo que llega a un taller mecánico para ser reparado o mantenido. Este vehículo representa una instancia de un programa que ha sido activado y ahora requiere atención.

! 2. Mecánico (Procesador)

El mecánico del taller es como el procesador del sistema, encargado de realizar las reparaciones necesarias. Este mecánico debe seguir un conjunto de instrucciones específicas (el programa) para reparar el vehículo.

3. Manual de Reparaciones (Programa)

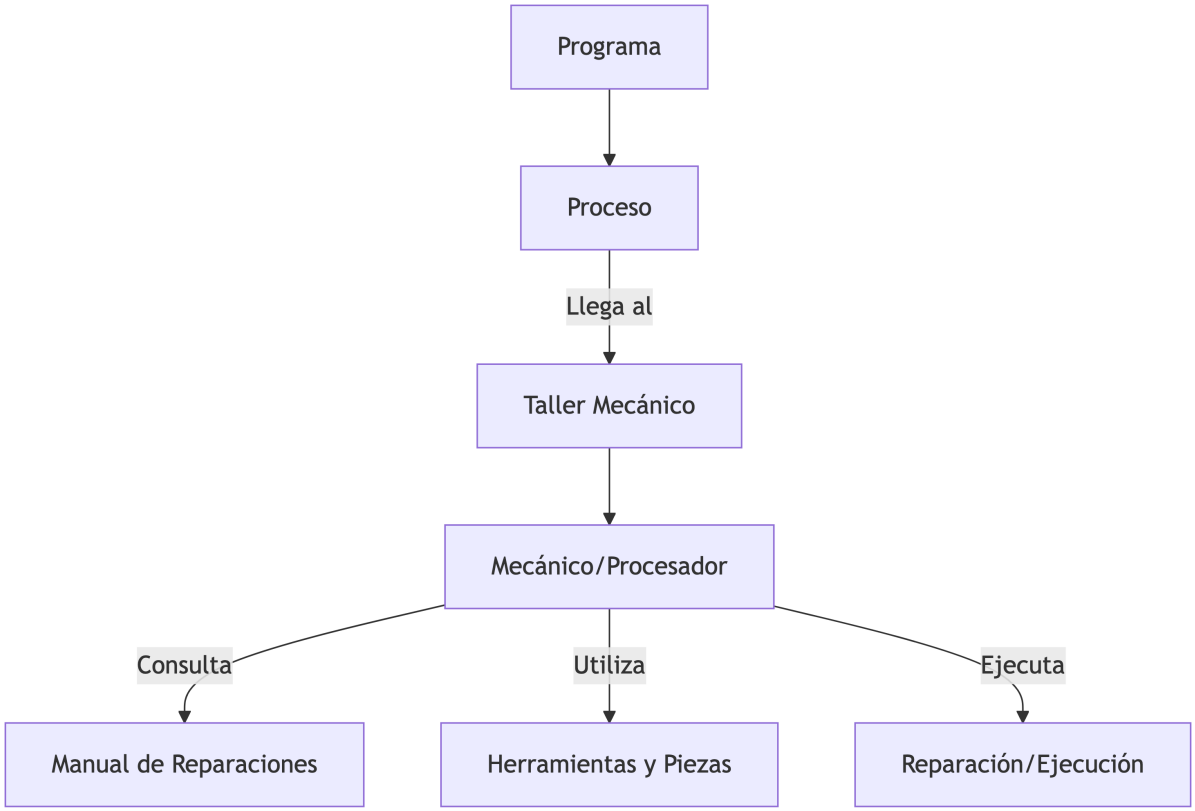
El mecánico consulta un manual de reparaciones, que contiene todas las instrucciones necesarias para realizar la reparación. Este manual es análogo al código del programa que el proceso necesita ejecutar.

⚠ 4. Herramientas y Piezas (Recursos del Sistema)

Para llevar a cabo las reparaciones, el mecánico necesita herramientas y piezas de recambio, que en el sistema operativo corresponden a los recursos del sistema, como la CPU, memoria, y dispositivos de entrada/salida.

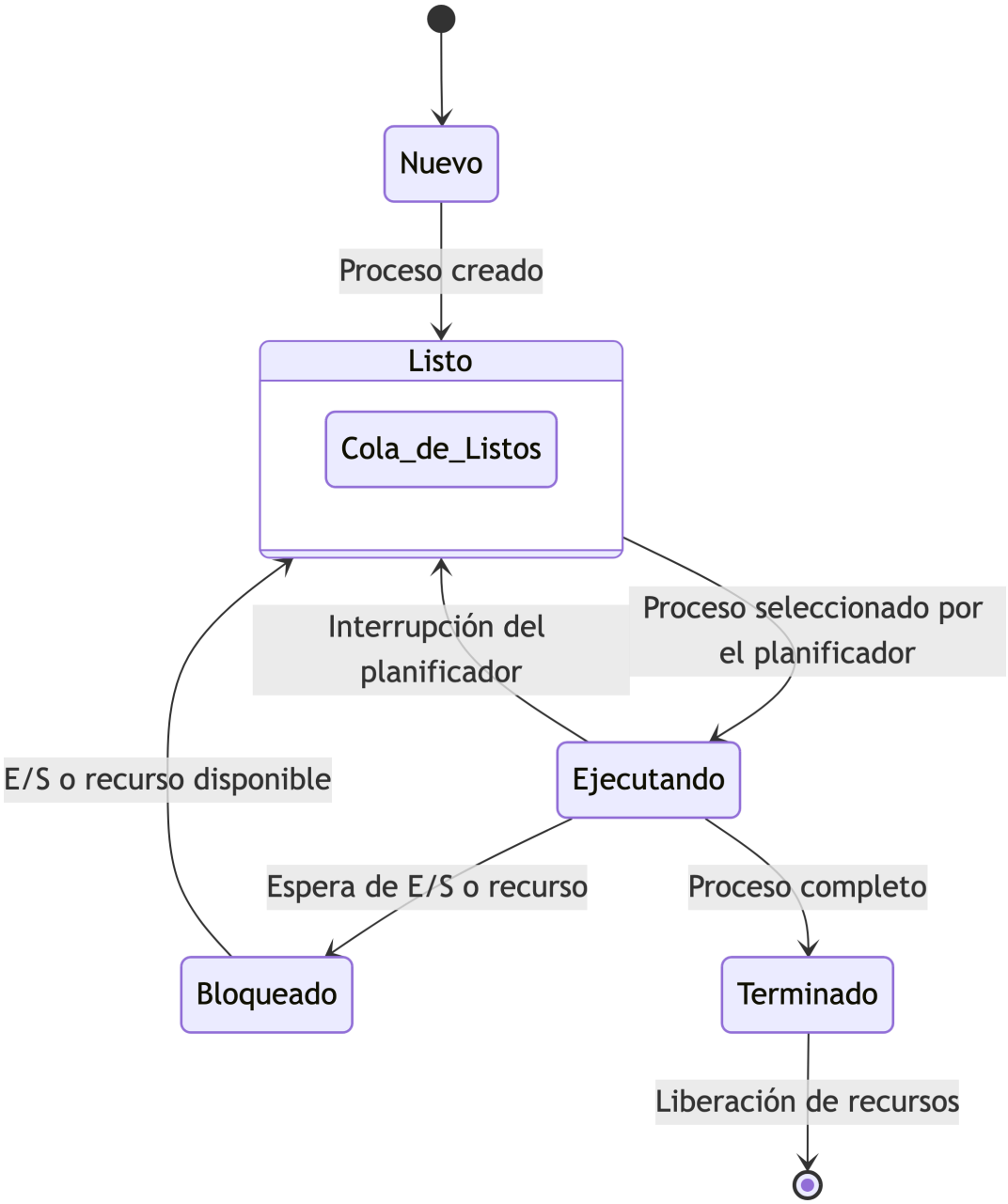
3.2.1 Ejecución

Mientras el vehículo está en el taller, el mecánico trabaja en él, siguiendo el manual, utilizando las herramientas y asegurándose de que la reparación se realice correctamente.



3.2.2 Analogía

Esto es similar a cómo un proceso se ejecuta en un sistema operativo, siguiendo las instrucciones del programa y utilizando los recursos disponibles.



Este diagrama muestra las principales etapas del ciclo de vida de un proceso en un sistema operativo:

1. **Nuevo**: El proceso se crea y se inicializa.
2. **Listo (Preparado)**: El proceso está listo para ejecutarse y espera en la cola de listos.
3. **Ejecutando**: El proceso está siendo ejecutado por la CPU.
4. **Bloqueado (Suspendido bloqueado o suspendido preparado)**: El proceso está esperando por algún evento o recurso.
5. **Terminado**: El proceso ha completado su ejecución.

Las transiciones entre estados representan las decisiones del sistema operativo y los eventos que ocurren durante la ejecución del proceso.

La idea clave es que un proceso es una actividad de cierto tipo: tiene un programa, una entrada, una salida y un estado. Varios procesos pueden compartir un solo procesador mediante el uso de un algoritmo de planificación para determinar cuándo se debe detener el trabajo en un proceso para dar servicio a otro.

3.3 Bloque de Control de Procesos (PCB)

Desde el punto de vista del sistema operativo, un proceso se representa como un conjunto de datos que describe su estado en cada momento, los recursos utilizados, los registros, y otra información relevante. Este conjunto de datos se denomina Bloque de control de procesos (PCB, por sus siglas en inglés: Process Control Block).

El PCB es fundamental para la gestión de procesos en un sistema operativo, ya que contiene toda la información necesaria para controlar y administrar un proceso. Cada proceso en el sistema tiene su propio PCB, y estos se almacenan en una estructura de datos que el sistema operativo utiliza para llevar un seguimiento preciso de todos los procesos.

! Componentes clave del PCB

- **Identificador del proceso (PID)**: Un número único que identifica a cada proceso en el sistema.
- **Estado del proceso**: El estado actual del proceso, que puede ser: nuevo, en ejecución, preparado, bloqueado, suspendido bloqueado y suspendido preparado; el estado del proceso también contiene información relativa a la prioridad del proceso.
- **Contador de programa (Program Counter) (PC)**: La dirección de la siguiente instrucción que se ejecutará para este proceso.
- **Registros de la CPU**: Incluye el contenido de los registros del procesador utilizados por el proceso, como los registros generales, punteros de pila, registros de estado, entre otros.
- **Información de gestión de memoria**: Detalles sobre la memoria asignada al proceso, incluyendo las tablas de páginas o segmentos.
- **Información de gestión de recursos**: Información sobre los recursos que el proceso está utilizando, como archivos abiertos, dispositivos de E/S, y otras asignaciones de recursos.
- **Información de contabilidad del proceso**: Datos sobre el uso de la CPU, límites de tiempo, y otra información relevante para la contabilidad del sistema.

3.3.1 Función del PCB

El PCB funciona como el «documento de identidad» del proceso dentro del sistema operativo. Cuando el sistema operativo cambia de un proceso a otro (cambio de contexto), guarda el estado del proceso saliente en su PCB y carga el estado del proceso entrante desde su respectivo PCB.

3.3.2 Gestión del PCB

El sistema operativo mantiene los PCBs en estructuras como listas enlazadas o tablas, organizadas según el estado de los procesos. Esto permite localizar rápidamente todos los procesos en un estado particular, facilitando la planificación y gestión de recursos.

Importancia en la multitarea

La existencia del PCB es lo que permite la multitarea en los sistemas operativos modernos. Al guardar y restaurar el estado completo de cada proceso, el sistema puede alternar entre múltiples procesos, dando la ilusión de ejecución simultánea incluso en sistemas con un solo procesador.

4. Planificación de Procesos

En un sistema operativo moderno, es habitual que se ejecuten decenas o cientos de procesos de manera aparentemente simultánea. Sin embargo, cada núcleo de CPU solo puede dedicarse a un proceso a la vez.

Para administrar adecuadamente el tiempo de CPU y decidir qué proceso se ejecuta en cada momento, el sistema operativo hace uso de un **planificador** (conocido en inglés como scheduler). El planificador selecciona el próximo proceso a ejecutar basándose en los criterios definidos por distintos **algoritmos de planificación**.

Antes de la era de los sistemas multitarea (capaces de ejecutar múltiples procesos en paralelo de forma concurrida), los sistemas monotarea simplemente ejecutaban una tarea y hasta que esta finalizaba, no se atendía otra. Actualmente, esta dinámica es mucho más compleja, pues intervienen prioridades, tiempos de ejecución, tiempos de espera y otros factores que se almacenan en el **PCB** (Process Control Block o Bloque de Control de Proceso). Por ello, resulta fundamental comprender cómo se diseña y cómo funciona la planificación de procesos.

4.1 Objetivos de aprendizaje

1. Comprender los conceptos fundamentales asociados a la planificación de procesos (tiempo de espera, servicio y retorno).
2. Analizar y comparar distintos algoritmos de planificación (Round Robin, FIFO, SJF, prioridad, HRN y colas múltiples).
3. Interpretar diagramas de Gantt para visualizar el comportamiento temporal de los procesos en la CPU.

4.2 Criterios que determinan un buen algoritmo de planificación

Para evaluar la eficacia de un algoritmo de planificación, se suelen tener en cuenta varios criterios:

1. **Equitatividad** (Fairness): Garantizar que todos los procesos reciban una parte justa del tiempo de CPU.
2. **Eficiencia**: Pretende mantener la CPU ocupada la mayor parte del tiempo para maximizar el rendimiento del sistema.
3. **Tiempo de Respuesta** (Response Time): Reducir al mínimo el tiempo que tarda el sistema en responder a una solicitud de un proceso o de un usuario.
4. **Tiempo de Retorno** (Turnaround Time): Disminuir el intervalo que transcurre desde que un proceso entra al sistema hasta que finaliza su ejecución.
5. **Volumen de Producción** (Throughput): Maximizar la cantidad de procesos completados en un intervalo de tiempo determinado.

Estos criterios suelen combinarse y priorizarse de forma distinta según el tipo de sistema (por ejemplo, sistemas en tiempo real, sistemas interactivos, servidores de alto rendimiento, etc.).

4.2.1 Conceptos clave en la planificación de procesos

Quantum

El *quantum* (también llamado time slice) es un intervalo de tiempo fijo que se asigna a un proceso cuando le llega su turno de ejecución. En cuanto se agota este intervalo, el proceso se interrumpe y, si aún no ha terminado, se reubica al final de la cola de listos para esperar un próximo turno. Este mecanismo es fundamental en algoritmos como Round Robin, que busca compartir la CPU de manera equitativa.

Tiempo de espera

El tiempo de espera es el **tiempo total** que un proceso permanece en la cola de listos antes de poder usar la CPU. Un algoritmo de planificación intenta, en lo posible, minimizar este valor para evitar que los procesos estén inactivos durante demasiado tiempo.

Tiempo de servicio Es el tiempo que la CPU dedica efectivamente a ejecutar un proceso, sin contar los tiempos de espera.

Tiempo de retorno

El tiempo de retorno es el tiempo total que transcurre desde que un proceso entra al sistema (se crea o se pone en la cola de listos) hasta que finaliza su ejecución por completo.

4.3 Algoritmos de planificación de procesos

Existen numerosos algoritmos de planificación que buscan satisfacer uno o varios de los criterios mencionados. A continuación, se presentan tres de los más conocidos y utilizados en entornos académicos y prácticos.

4.3.1 Algoritmo Round Robin

Es uno de los más antiguos, sencillos y más utilizados en la planificación de procesos. El algoritmo Round Robin se caracteriza por:

- Usar un quantum (por ejemplo, 100 ms).
- Repartir la CPU entre los procesos en la cola de manera circular.
- Después de que un proceso utiliza la CPU durante un quantum, si no ha terminado, pasa al final de la cola de listos.

Este enfoque favorece la equitatividad, pues cada proceso recibe un turno de ejecución frecuente, evitando que algunos procesos monopolicen la CPU. Sin embargo, si el *quantum* es muy grande, el comportamiento puede parecerse a **FIFO** y aumentar el tiempo de espera de otros procesos.

Características del Round Robin (RR)

- **Política apropiativa:** Si el proceso no termina dentro de su *quatum* de tiempo, se interrumpe y se coloca al final de la cola de procesos listos.
- ***Quantum* fijo:** Todos los procesos reciben la misma cantidad de tiempo para ejecutarse en cada ciclo.
- **Cola circular de procesos listos:** Los procesos se ejecutan en orden y si no terminan, se colocan nuevamente al final de la cola en espera de terminar.

Ejemplo de Round Robin

Supongamos 3 procesos *P1, P2, P3* que llegan y un deben ser atendidos con un *quantum* de 4 unidades de tiempo:

- P1 con tamaño 8 llega en tiempo 0
- P2 tiene tamaño 4 y llega en tiempo 2
- P3 tiene un tamaño 2 y llega en tiempo 5

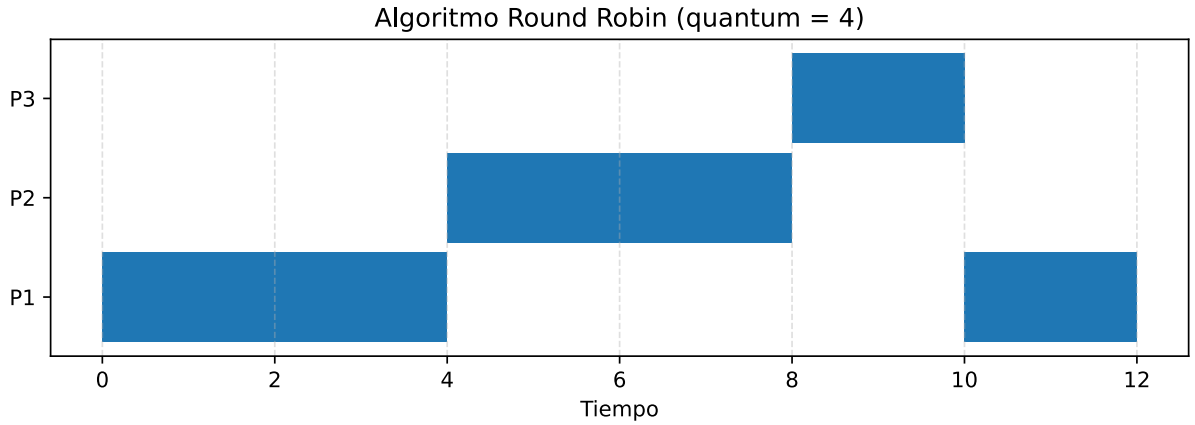


Figure 4.1: Diagrama de Gantt del algoritmo Round Robin (quantum = 4)

4.3.2 Algoritmo FIFO (First In First Out)

El algoritmo FIFO o FCFS (First Come First Served) es el más simple y consiste en atender los procesos en el orden en que llegan a la cola de listos. Una vez que un proceso empieza a ejecutarse, mantiene el uso de la CPU hasta que finaliza o bloquea.

- **Ventaja**: Es sencillo de implementar y entender.
- **Desventaja**: Puede crear largos tiempos de espera si un proceso extenso llega antes que otros muy cortos (fenómeno conocido como convoy effect).

Características de FIFO

- **Política no apropiativa:** Una vez que el proceso comienza a ejecutarse, no se interrumpe hasta que se complete.
- **Los procesos se ejecutan por orden de llegada:** El primer proceso en llegar, es el primero en ser atendido.
- **Todos los procesos esperan en una sola cola:** Los procesos en la cola deben esperar su turno hasta que la CPU esté libre.

Ejemplo de FIFO

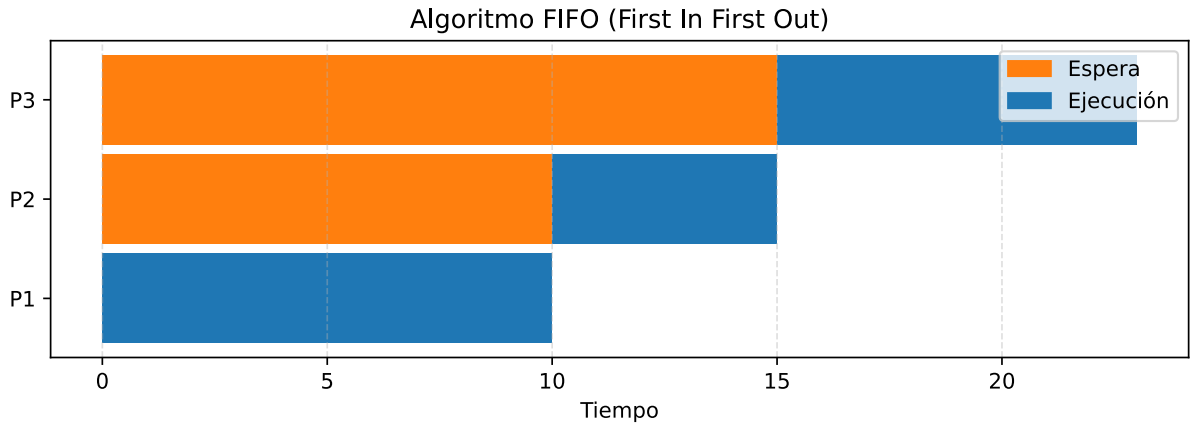


Figure 4.2: Diagrama de Gantt del algoritmo FIFO

4.3.3 Algoritmo SJF (Shortest Job First)

El algoritmo SJF (Shortest Job First, en español «Trabajo Más Corto Primero») selecciona para su ejecución el proceso con el tiempo de ráfaga (tiempo de CPU requerido) más corto.

- **Ventaja**: Minimiza el tiempo promedio de espera y, a menudo, el tiempo de retorno promedio.
- **Desventaja**: Requiere conocer (o estimar) la duración de la ráfaga de CPU de cada proceso con anticipación, lo cual no siempre es factible.

Características del SJF:

- **No Apropiativo:** Una vez que un proceso con la menor duración es seleccionado, se ejecuta hasta su finalización sin interrupciones.
- **Minimización del tiempo promedio de espera:** Como se priorizan los procesos más cortos, se optimiza el tiempo promedio de espera de todos los procesos en el sistema.
- **Problema del «hambre»:** Los procesos largos pueden quedar esperando indefinidamente si continuamente llegan procesos más cortos, lo que se conoce como *starvation*.

Ejemplo de SJF no apropiativo

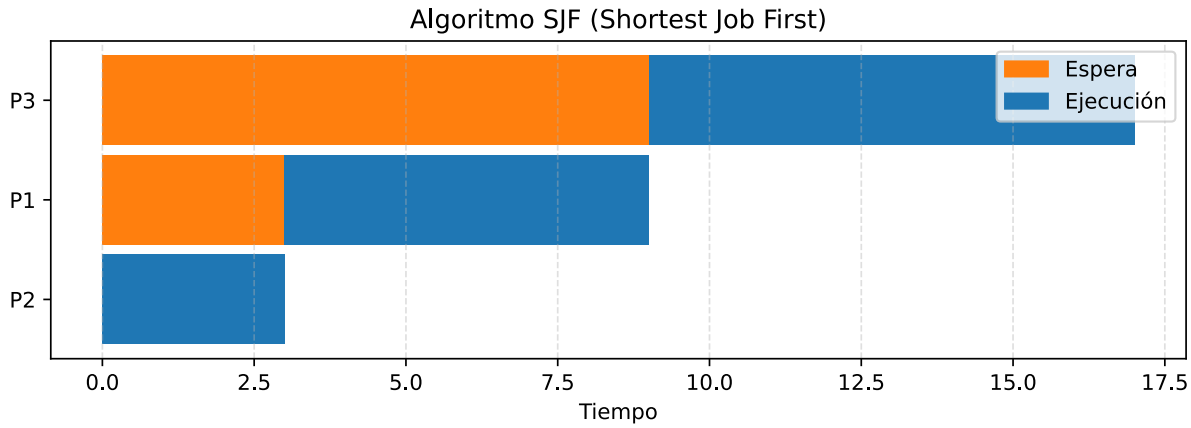


Figure 4.3: Diagrama de Gantt del algoritmo SJF no apropiativo

Se puede observar que el **P2**, con menor tiempo de ejecución es atendido primero, luego es ejecutado el **P1** (ráfaga 6), y finalmente, el **P3**.

4.4 Algoritmo de Prioridad

El algoritmo de Prioridad es una política de planificación en la cual cada proceso se asocia con un valor de prioridad. Los procesos con mayor prioridad tienen preferencia para acceder a la CPU y se ejecutan primero. Este enfoque es útil en sistemas donde ciertos procesos requieren un tratamiento preferencial debido a su importancia o urgencia.

La prioridad puede asignarse de dos maneras:

- **Prioridad externa**: Determinada por factores ajenos al sistema operativo, como la importancia del proceso para el usuario o la organización.
- **Prioridad interna**: Calculada por el sistema operativo en función de parámetros como el tiempo estimado de ejecución, los recursos solicitados o el historial de uso de CPU.

Características del Algoritmo de Prioridad:

- **Puede ser apropiativo o no apropiativo**:
  - En la versión no apropiativa, una vez que un proceso comienza a ejecutarse, continúa hasta terminar.
  - En la versión apropiativa, si llega un proceso con mayor prioridad, el proceso en ejecución se interrumpe.
- **Problema del «hambre»**: Los procesos de baja prioridad pueden sufrir inanición si continuamente llegan procesos de mayor prioridad.
- **Solución al problema de inanición**: Implementar el «envejecimiento» (aging), donde la prioridad de un proceso aumenta gradualmente con el tiempo de espera.

Ejemplo de Algoritmo de Prioridad (no apropiativo)

Supongamos que tenemos tres procesos con las siguientes características:

- P1: Tiempo de llegada = 0, Tiempo de ejecución = 7, Prioridad = 2
- P2: Tiempo de llegada = 2, Tiempo de ejecución = 4, Prioridad = 1
- P3: Tiempo de llegada = 4, Tiempo de ejecución = 5, Prioridad = 3

Donde la prioridad 1 es la más alta y 3 la más baja.

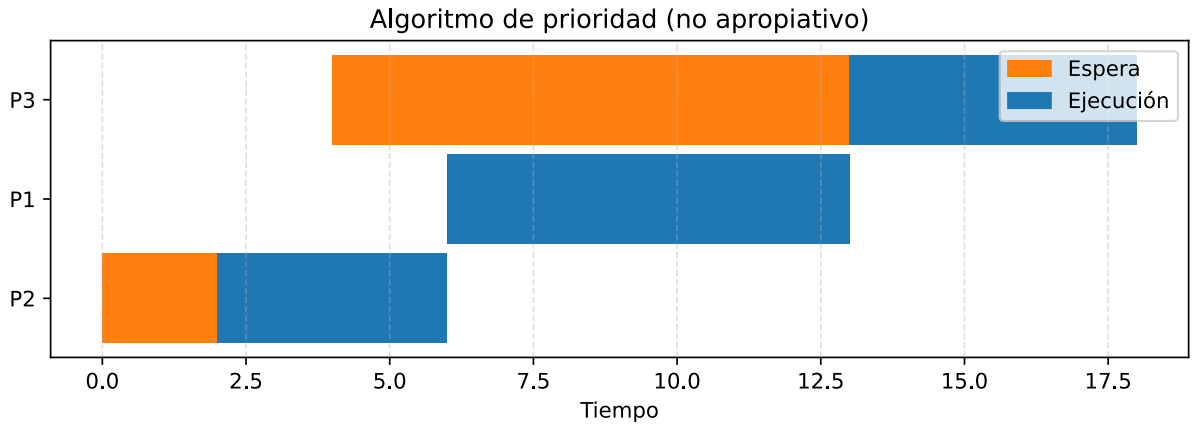


Figure 4.4: Diagrama de Gantt del algoritmo de prioridad (no apropiativo)

En este ejemplo, aunque P1 llega primero, debe esperar a que P2 (con mayor prioridad) termine su ejecución. P3, al tener la prioridad más baja, debe esperar a que tanto P2 como P1 terminen.

Ejemplo de Algoritmo de Prioridad (apropiativo)

Usando los mismos procesos del ejemplo anterior:

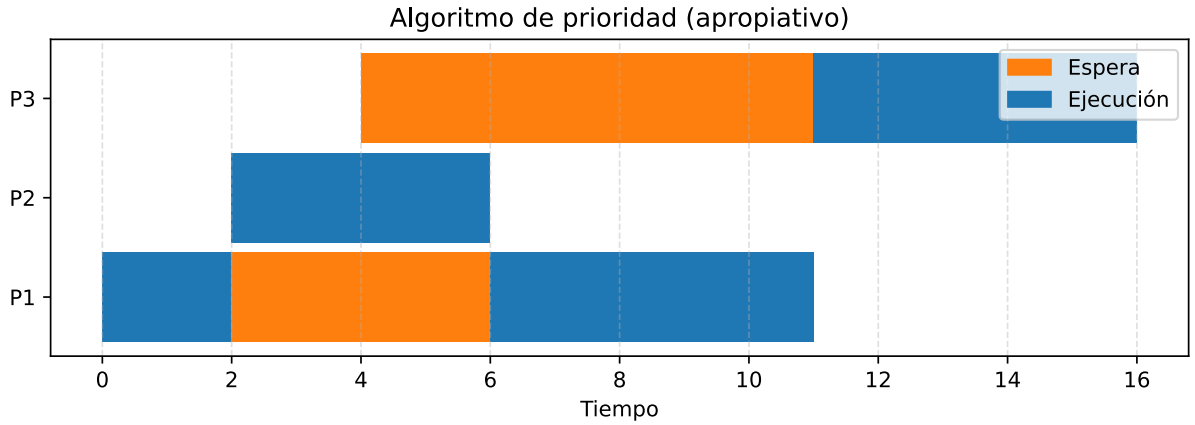


Figure 4.5: Diagrama de Gantt del algoritmo de prioridad (apropiativo)

En la versión apropiativa, P1 comienza su ejecución pero es interrumpido cuando llega P2 (con mayor prioridad). Una vez que P2 termina, P1 reanuda su ejecución, y finalmente se ejecuta P3.

4.5 Algoritmo High Response Ratio Next (HRN)

El algoritmo High Response Ratio Next (HRN) es una política de planificación que prioriza los procesos en función de la relación entre el tiempo de espera y el tiempo de ejecución. A diferencia de otros algoritmos, HRN tiene en cuenta tanto el tiempo de servicio como el tiempo de espera para determinar qué proceso debe ejecutarse a continuación.

La fórmula utilizada para calcular la prioridad en HRN es:

$$\text{Prioridad} = \frac{\text{Tiempo de Espera} + \text{Tiempo de Servicio}}{\text{Tiempo de Servicio}} \tag{4.1}$$

Características del Algoritmo HRN:

- **No apropiativo**: Una vez que un proceso comienza a ejecutarse, continúa hasta completarse.
- **Balance entre SJF y FIFO**: Combina las ventajas de ambos algoritmos, favoreciendo procesos cortos pero sin ignorar aquellos que han esperado mucho tiempo.
- **Prevención de inanición**: A medida que un proceso espera, su relación de respuesta aumenta, incrementando su prioridad.
- **Requiere conocer el tiempo de servicio**: Al igual que SJF, necesita una estimación del tiempo de ejecución de cada proceso.

Ejemplo de Algoritmo HRN

Supongamos que tenemos tres procesos con las siguientes características:

- P1: Tiempo de llegada = 0, Tiempo de servicio = 10
- P2: Tiempo de llegada = 2, Tiempo de servicio = 5
- P3: Tiempo de llegada = 3, Tiempo de servicio = 2

En el tiempo t=0, solo P1 está disponible, por lo que comienza a ejecutarse. En t=10, P1 termina y calculamos las prioridades:

- P2: (8 + 5) / 5 = 2.6

- P3: (7 + 2) / 2 = 4.5

P3 tiene mayor prioridad, por lo que se ejecuta a continuación. En t=12, P3 termina y P2 es el único proceso restante, por lo que se ejecuta.

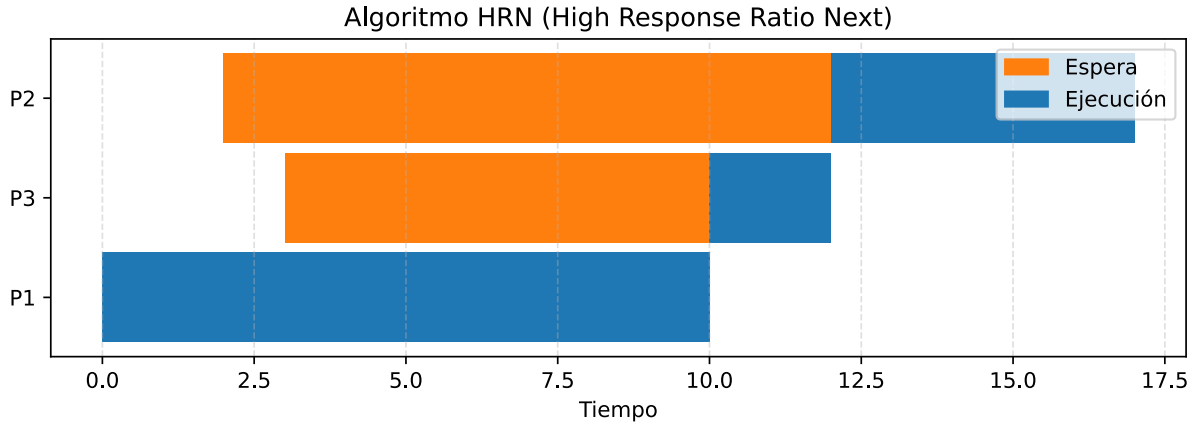


Figure 4.6: Diagrama de Gantt del algoritmo HRN

4.6 Algoritmo de Colas Múltiples

El algoritmo de Colas Múltiples es una política de planificación compleja que divide la cola de procesos listos en varias colas separadas, cada una gestionada con su propio algoritmo de planificación. Este enfoque permite tratar diferentes tipos de procesos de manera distinta, optimizando el rendimiento global del sistema.

Características del Algoritmo de Colas Múltiples:

- **Clasificación de procesos**: Los procesos se clasifican en diferentes categorías según sus características (interactivos, por lotes, en tiempo real, etc.).
- **Múltiples algoritmos**: Cada cola puede utilizar un algoritmo de planificación diferente (Round Robin para procesos interactivos, FIFO para procesos por lotes, etc.).
- **Prioridad entre colas**: Generalmente existe una jerarquía entre las colas, donde las de mayor prioridad se atienden primero.
- **Migración entre colas**: En algunas implementaciones, los procesos pueden moverse entre colas según su comportamiento o tiempo de ejecución.

Tipos de Colas Múltiples:

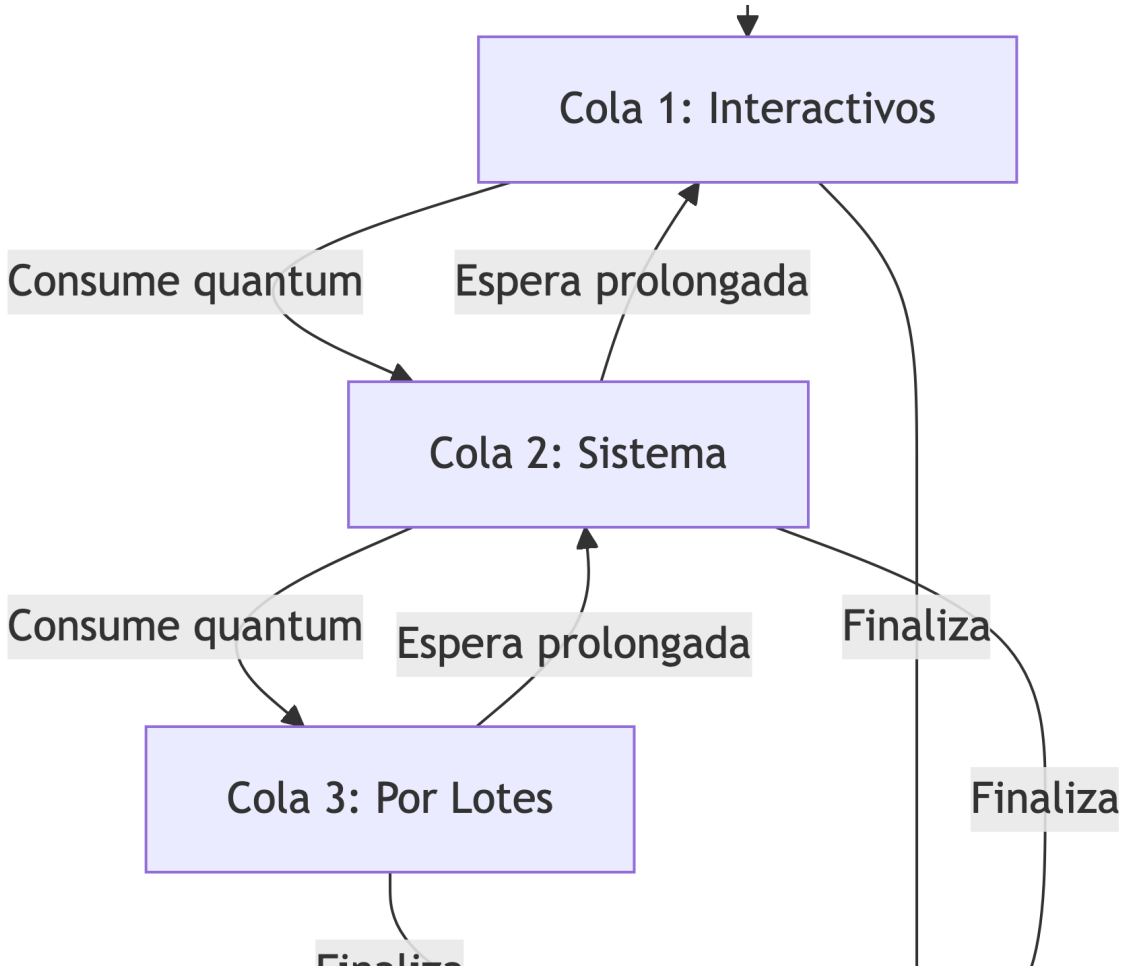
1. **Colas Múltiples sin Realimentación**:
   - Los procesos permanecen en la misma cola durante toda su vida.
   - La asignación a una cola específica se realiza al crear el proceso.
   - No hay movilidad entre colas.
2. **Colas Múltiples con Realimentación (Feedback)**:
   - Los procesos pueden moverse entre colas según su comportamiento.
   - Típicamente, los procesos nuevos entran en la cola de mayor prioridad.
   - Si un proceso consume demasiado tiempo de CPU, puede descender a colas de menor prioridad.
   - Los procesos que esperan mucho tiempo pueden ascender a colas de mayor prioridad.

Ejemplo de Colas Múltiples con Realimentación

Supongamos un sistema con tres colas:

- Cola 1: Procesos interactivos (Round Robin con quantum = 2)
- Cola 2: Procesos de sistema (Round Robin con quantum = 4)
- Cola 3: Procesos por lotes (FIFO)

La prioridad es descendente: Cola 1 > Cola 2 > Cola 3.



En este sistema:

1. Todos los procesos nuevos entran en la Cola 1.
2. Si un proceso en la Cola 1 consume su quantum sin terminar, pasa a la Cola 2.
3. Si un proceso en la Cola 2 consume su quantum sin terminar, pasa a la Cola 3.
4. La CPU atiende primero todos los procesos de la Cola 1, luego los de la Cola 2, y finalmente los de la Cola 3.
5. Para evitar la inanición, los procesos que esperan demasiado tiempo en colas inferiores pueden ascender a colas superiores.

Este algoritmo es muy flexible y permite ajustar el comportamiento del sistema según las necesidades específicas, balanceando la atención entre procesos interactivos (que requieren tiempos de respuesta cortos) y procesos por lotes (que buscan maximizar el throughput).

4.7 Resumen

- La planificación de procesos permite compartir de forma ordenada el tiempo de CPU entre múltiples programas.
- Los criterios clásicos de evaluación incluyen equitatividad, tiempo de respuesta, tiempo de retorno y throughput.
- Algoritmos como FIFO, SJF, prioridad y HRN optimizan distintos criterios, pero pueden provocar hambre en ciertos procesos.
- Round Robin y las colas múltiples resultan más adecuados para sistemas interactivos, al ofrecer tiempos de respuesta más predecibles.

4.8 Ejercicios propuestos

1. Dibuja a mano los diagramas de Gantt de Round Robin y FIFO para conjuntos de procesos diferentes a los ejemplos del texto. Compara tiempos de espera y de retorno.
2. Para un conjunto de procesos de tu elección, calcula el tiempo de espera y tiempo de retorno promedio usando FIFO, SJF y HRN. ¿Qué algoritmo ofrece mejor equilibrio?
3. Propón una política de prioridades para un laboratorio de computación (procesos interactivos de estudiantes, tareas de respaldo, procesos de administración). Justifica qué algoritmo usarías en cada cola de un esquema de colas múltiples.

5. Los hilos (threads) y el TCB

5.1 ¿Qué es un hilo?

Un **hilo** (*thread*) es la unidad más pequeña de ejecución que el sistema operativo puede planificar en la CPU. Se trata de una secuencia de instrucciones que se ejecuta de forma independiente, pero **dentro del contexto de un proceso**.

La diferencia clave entre un proceso y un hilo radica en los recursos: un proceso posee su propio espacio de direcciones, archivos abiertos y memoria asignada. Los hilos que pertenecen a un mismo proceso **comparten** todos esos recursos, pero cada uno mantiene sus propios elementos de ejecución:

- **Contador de programa:** indica la siguiente instrucción a ejecutar.
- **Pila de ejecución (*stack*):** almacena variables locales y llamadas a funciones.
- **Conjunto de registros:** refleja el estado actual de la CPU para ese hilo.

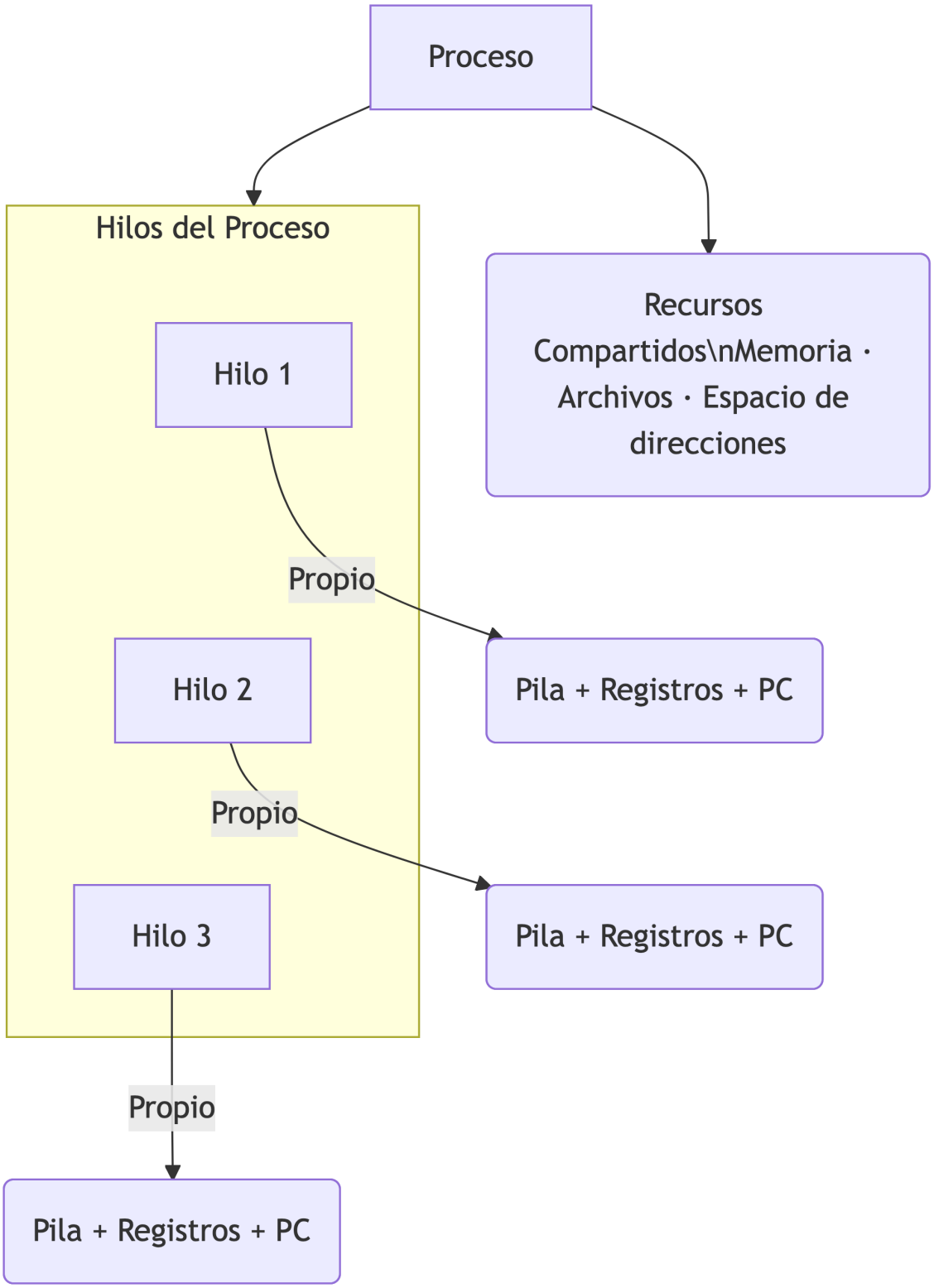


Figure 5.1: Estructura de hilos dentro de un proceso: recursos compartidos vs. elementos propios.

Esta forma de organización convierte al hilo en un **proceso ligero**: crear un hilo es más rápido que crear un proceso completo, y el cambio de contexto entre hilos del mismo proceso tiene menor costo, ya

que no es necesario cambiar el espacio de direcciones. Por ello, los hilos son ampliamente utilizados en aplicaciones modernas para lograr concurrencia y mejorar la capacidad de respuesta.

5.1.1 Ejemplo didáctico: hilos en una aplicación móvil

Consideremos una aplicación móvil que realiza varias tareas al mismo tiempo. Cada tarea se asigna a un hilo diferente:

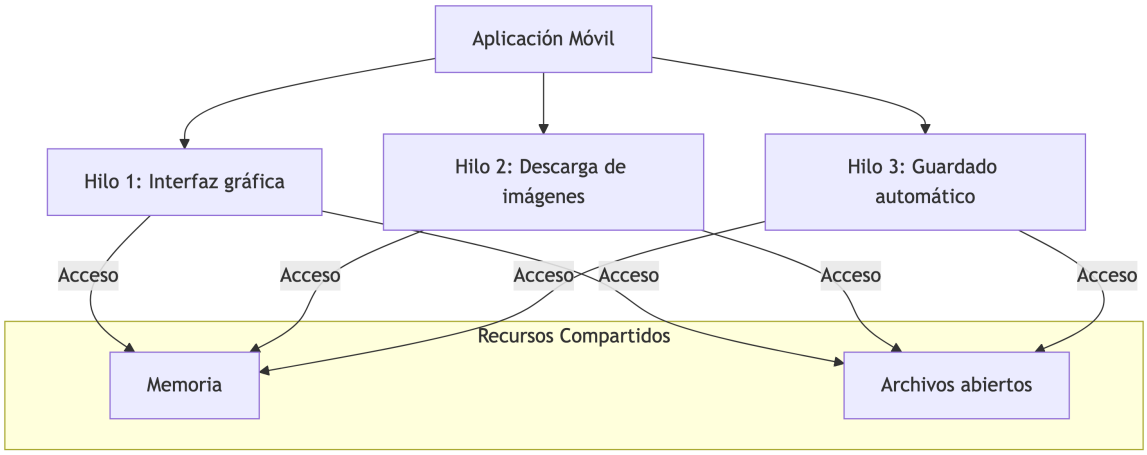


Figure 5.2: Aplicación móvil con tres hilos que comparten los mismos recursos.

- **Hilo 1 (interfaz gráfica):** mantiene la pantalla actualizada y responde a las interacciones del usuario (toques, gestos, navegación).
- **Hilo 2 (descarga de imágenes):** obtiene contenido desde internet en segundo plano, sin bloquear la interfaz.
- **Hilo 3 (guardado automático):** persiste el progreso del usuario de forma periódica.

La ventaja de esta separación es directa: si la descarga de imágenes se demora o falla, la interfaz sigue respondiendo con normalidad y el progreso se guarda sin interrupciones. Sin hilos, la aplicación tendría que esperar a que cada tarea termine antes de atender la siguiente, lo que produciría una experiencia lenta y poco fluida.

5.2 Thread Control Block (TCB)

Así como el sistema operativo utiliza un **PCB** (*Process Control Block*) para gestionar cada proceso, utiliza un **TCB** (*Thread Control Block*) para gestionar cada hilo. El TCB es la estructura de datos que almacena toda la información necesaria para que el sistema operativo pueda planificar, pausar, reanudar o finalizar un hilo de forma individual.

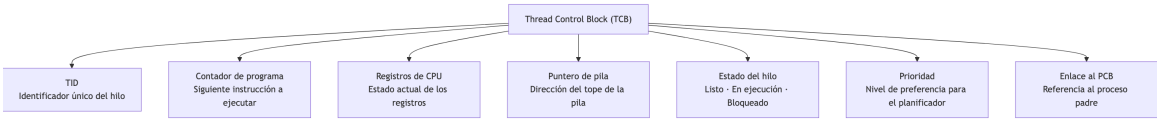


Figure 5.3: Campos principales del Thread Control Block.

Cada campo cumple un rol específico:

- **TID (*Thread ID*):** identifica de manera única al hilo dentro del sistema.
- **Contador de programa:** indica exactamente en qué punto del código se encuentra el hilo.
- **Registros de CPU:** se guardan y restauran durante cada cambio de contexto para que el hilo retome su ejecución sin perder su estado.
- **Puntero de pila:** señala la posición actual en la pila del hilo, necesaria para gestionar llamadas a funciones y variables locales.
- **Estado del hilo:** puede ser *listo* (esperando turno en la CPU), *en ejecución* o *bloqueado* (esperando un recurso o evento).
- **Prioridad:** determina qué tan pronto el planificador le asignará tiempo de CPU frente a otros hilos.
- **Enlace al PCB:** vincula al hilo con su proceso padre, permitiendo acceder a los recursos compartidos (memoria, archivos abiertos, etc.).

Gracias al TCB, el sistema operativo puede realizar cambios de contexto entre hilos de forma eficiente: guarda el estado del hilo saliente en su TCB y carga el estado del hilo entrante desde el suyo.

5.3 Comunicación entre Procesos (IPC)

La **Comunicación entre Procesos** o **IPC** (*Inter-Process Communication*) es el conjunto de mecanismos que el sistema operativo ofrece para que los procesos puedan intercambiar datos y coordinar sus acciones. Sin IPC, cada proceso sería una entidad aislada, incapaz de colaborar con los demás.

En sistemas Unix/Linux, los principales mecanismos de IPC son:

- **Señales:** notificaciones asíncronas que un proceso envía a otro para indicar que ocurrió un evento. Por ejemplo, al presionar `Ctrl+C` en la terminal, el sistema envía la señal `SIGINT` al proceso en primer plano para solicitar su terminación.
- **Pipes (tuberías):** canales unidireccionales que conectan la salida estándar de un proceso con la entrada estándar de otro. En la terminal se representan con el símbolo `|`. Por ejemplo, `ls | grep ".txt"` envía la lista de archivos al proceso `grep`, que filtra solo los que terminan en `.txt`.
- **Semáforos:** mecanismos de sincronización que controlan el acceso a recursos compartidos. Funcionan como un contador: antes de acceder al recurso, el proceso «baja» el semáforo; si el valor llega a cero, los demás procesos deben esperar. Esto evita las **condiciones de carrera**, es decir, situaciones donde el resultado depende del orden impredecible en que los procesos acceden a un mismo recurso.
- **Colas de mensajes:** permiten que los procesos envíen y reciban mensajes estructurados a través de una cola gestionada por el sistema operativo. Los mensajes se conservan en orden (FIFO) hasta que el proceso receptor los extrae.
- **Memoria compartida:** permite que múltiples procesos accedan a una misma región de memoria RAM. Es el mecanismo de IPC más rápido, ya que no requiere copiar datos entre procesos. Sin embargo, exige mecanismos de sincronización adicionales (como semáforos o *mutexes*) para evitar lecturas y escrituras simultáneas que corrompan los datos.

La elección del mecanismo depende del caso de uso: las señales son ideales para notificaciones simples, los pipes para encadenar procesos secuenciales, y la memoria compartida para transferencias de datos de alto volumen donde el rendimiento es crítico.

III

III. Memoria de los Sistemas Operativos

6. La Administración de Memoria en Sistemas Operativos

La administración de memoria constituye uno de los pilares fundamentales en el diseño y operación de sistemas operativos modernos. Su objetivo principal radica en optimizar el uso de la memoria física y virtual, garantizando que los procesos accedan a los recursos necesarios mientras se mantiene la estabilidad del sistema. Este campo ha evolucionado desde enfoques rudimentarios, como el **monitor residente**, hasta técnicas sofisticadas como la **partición dinámica**, cada una abordando desafíos específicos de fragmentación, seguridad y eficiencia. En este material, exploraremos las políticas clásicas de administración de memoria, analizando su funcionamiento, ventajas y limitaciones en contextos reales.

6.1 Evolución Histórica de la Gestión de Memoria

6.1.1 Del Monitor Residente a la Multiprogramación

Los primeros sistemas operativos, como el **monitor residente**, surgieron en la década de 1950 para automatizar la carga de tareas en entornos *batch*. Este software primitivo residía permanentemente en memoria, coordinando la ejecución secuencial de programas mediante tarjetas de control. Aunque revolucionario para su época, presentaba serias limitaciones:

1. **Ausencia de concurrencia**: Solo un proceso ocupaba memoria en cada instante.
2. **Gestión manual**: El operador debía agrupar tareas compatibles para evitar conflictos de recursos.
3. **Inflexibilidad**: La memoria se dividía estáticamente entre el sistema y los programas de usuario.

Estos obstáculos motivaron el desarrollo de esquemas más avanzados, como la **multiprogramación**, que permitió ejecutar múltiples procesos simultáneamente mediante la asignación dinámica de memoria.

6.2 Políticas Clásicas de Administración de Memoria

6.2.1 Monitor Residente: La Cuna de la Gestión

El monitor residente operaba bajo un principio de **separación estricta** entre el espacio del sistema y el usuario. Utilizaba un registro de «cerca» (*fence register*) para delimitar ambas zonas, verificando cada acceso a memoria mediante hardware o software.

Ejemplo didáctico: Imaginemos una biblioteca donde el bibliotecario (monitor) ocupa siempre las primeras estanterías. Los usuarios (programas) solo pueden acceder a las secciones posteriores, y cualquier intento de entrar al área reservada genera una alerta.

Limitaciones:

- **Fragmentación interna**: Si el programa del usuario requería menos espacio que la partición asignada, el excedente se desperdiciaba.

- **Rigidez**: La ubicación del monitor no podía modificarse durante la ejecución, dificultando la escalabilidad.

6.2.2 Partición Estática: Organización Predefinida

En este esquema, la memoria se divide en **regiones de tamaño fijo** antes de la ejecución de procesos. Cada partición alberga un único programa, y el sistema operativo emplea algoritmos simples para asignarlas:

1. **Primer ajuste (*First-fit*)**: Asigna la primera partición libre que cumpla con el tamaño requerido.
2. **Mejor ajuste (*Best-fit*)**: Busca la partición más pequeña que pueda contener al proceso.
3. **Peor ajuste (*Worst-fit*)**: Utiliza la partición más grande disponible, dejando espacio residual para futuros procesos.

Caso de estudio: Un sistema con particiones de 64 KB, 128 KB y 256 KB. Si un proceso de 100 KB solicita memoria:

- *Primer ajuste*: Ocupa la partición de 128 KB, desperdiciando 28 KB.
- *Mejor ajuste*: Usa la misma partición de 128 KB.
- *Peor ajuste*: Asigna la de 256 KB, generando 156 KB no utilizados.

Problemas inherentes:

- **Fragmentación interna**: Espacio no utilizado dentro de particiones asignadas.
- **Subutilización**: Procesos pequeños ocupan particiones grandes, reduciendo la capacidad de multiprogramación.

6.2.3 Partición Dinámica: Flexibilidad con Costos

Para superar las limitaciones de la partición estática, se introdujo la **asignación dinámica**, donde las particiones se crean en tiempo de ejecución según las necesidades de cada proceso. Este enfoque requiere estructuras de datos complejas (tablas de huecos libres) y algoritmos de coalescencia para fusionar espacios adyacentes liberados.

Mecanismo operativo:

1. **Solicitud de memoria**: El proceso indica cuánto espacio necesita.
2. **Búsqueda de hueco**: El sistema identifica un bloque libre usando algoritmos como:
   - *Next-fit*: Continúa buscando desde la última asignación.
   - *Buddy system*: Divide la memoria en bloques de potencias de dos para facilitar la coalescencia.
3. **Asignación y actualización**: Se marca el espacio como ocupado y se actualizan las tablas de gestión.

6.3 Comparativa entre Enfoques

Aspecto	Monitor Residente	Partición Estática	Partición Dinámica
Flexibilidad	Baja	Moderada	Alta
Fragmentación	Interna	Interna	Externa
Complejidad	Simple	Moderada	Alta

Aspecto	Monitor Residente	Partición Estática	Partición Dinámica
Multiprogramación	No soportada	Limitada	Soportada
Ejemplos históricos	IBM 1401	OS/360	UNIX

Tabla 1: Características comparativas de políticas de gestión de memoria

6.3.1 Paginación

La **paginación** es una técnica de gestión de memoria utilizada por los sistemas operativos que permite dividir un programa en bloques de tamaño fijo llamados páginas, los cuales pueden ubicarse en diferentes partes de la memoria física, sin necesidad de estar contiguos.

Antes de ejecutar un programa, el sistema realiza los siguientes pasos:

1. **Calcula el número de páginas necesarias**, según el tamaño total del programa.
2. **Identifica los marcos de memoria disponibles**, que son bloques de igual tamaño en la memoria física.
3. **Carga cada página del programa** en uno de esos marcos disponibles.

Este enfoque permite que un programa sea cargado en regiones dispersas de la memoria, evitando la necesidad de bloques contiguos y reduciendo la fragmentación externa.

No obstante, este método requiere un **mecanismo de seguimiento** para mantener el control sobre dónde se ha almacenado cada página. Ese mecanismo es la **tabla de páginas**, la cual permite traducir direcciones lógicas (generadas por el programa) en direcciones físicas (utilizadas por la memoria real).

La paginación también es la base de la **virtualización de la memoria,** una técnica esencial en los sistemas modernos que permite a cada proceso creer que dispone de su propio espacio de memoria continuo, aunque en realidad este espacio esté repartido en bloques dispersos e incluso parcialmente almacenado en disco.

Gracias a esto, los sistemas pueden ejecutar programas que exceden el tamaño de la memoria física, mejorar el aislamiento entre procesos y optimizar el uso de los recursos disponibles.

6.3.2 Segmentación

La segmentación es un método de gestión de memoria donde un programa se divide en partes lógicas de diferente tamaño, llamadas segmentos. A diferencia de la paginación —donde todo se divide en bloques iguales— en segmentación cada parte puede tener un tamaño distinto, según la función que cumpla en el programa.

Un programa no es una sola cosa continua: normalmente está compuesto por varias secciones, como:

- El programa principal (la función principal o main)
- Varias subrutinas o funciones auxiliares
- Un bloque de datos globales
- Una pila para las llamadas de funciones

Cada una de estas partes se puede convertir en un segmento. Por ejemplo:

- Segmento 0: código principal (main)
- Segmento 1: subrutina A
- Segmento 2: subrutina B

- Segmento 3: datos globales
- Segmento 4: pila

Cada segmento se carga en una parte de la memoria principal, pero no necesariamente uno al lado del otro. El sistema operativo lleva un registro de cada segmento mediante una tabla de segmentos, que guarda dos datos importantes:

1. **La dirección base**: dónde empieza ese segmento en memoria.
2. **El límite**: cuánto ocupa (es decir, su tamaño).

6.3.3 Carga dinámica

La carga dinámica es una técnica que permite a un programa cargar en memoria ciertas partes de su código solo **cuando realmente las necesita** durante la ejecución.

Por ejemplo, una aplicación que incluye una función para exportar archivos en diferentes formatos puede decidir cargar la biblioteca necesaria para `"Exportar a PDF"` solo si el usuario elige esa opción. Hasta ese momento, esa parte del programa ni siquiera está en memoria.

En sistemas operativos como Linux, esto se puede implementar usando funciones como `dlopen()` y `dlsym()`, que permiten cargar y utilizar componentes de forma dinámica. Esto le da al programa mayor flexibilidad, permitiendo incluso extender su comportamiento en tiempo de ejecución, como ocurre en programas que cargan plugins o módulos personalizados.

En resumen, la carga dinámica retrasa el momento en que ciertos componentes se incorporan al programa en ejecución, mejorando el uso de memoria y permitiendo estructuras más modulares.

6.3.4 Enlace dinámico

El enlace dinámico es un proceso que ocurre cuando se carga el programa en memoria, y consiste en vincular el programa con las bibliotecas compartidas que necesita, sin que esas bibliotecas estén incluidas directamente dentro del archivo ejecutable.

Los ejecutables contienen referencias a bibliotecas externas (como archivos `.dll` en Windows o `.so` en Linux), que el sistema operativo se encarga de buscar y conectar cuando se ejecuta el programa.

Este mecanismo permite que varios programas compartan una misma copia de una biblioteca en memoria, lo que ahorra recursos. Además, si se actualiza o corrige una biblioteca, todos los programas que la utilizan se benefician automáticamente, sin necesidad de recompilarlos.

Bibliografía

[1] A. S. Tanenbaum y H. Bos, *Modern Operating Systems*, 5th ed. Pearson, 2017.

[2] A. Silberschatz, P. B. Galvin, y G. Gagne, *Operating System Concepts*, 10th ed. Wiley, 2018.