Resumen Profundo y Ampliado: Fundamentos de Sistemas Operativos

Este documento presenta un análisis detallado y extendido de los conceptos clave en Sistemas Operativos. Se ha diseñado para ofrecer una comprensión robusta no solo de las definiciones, sino del "porqué" y el "cómo" funcionan estos mecanismos internos.

1. Conceptos Fundamentales: Procesos, Hilos y Arquitectura

Diferencias entre Proceso e Hilo: Arquitectura de Ejecución

La distinción entre proceso e hilo es la base de la programación concurrente moderna. Entender sus matices es vital para optimizar el rendimiento.

- **Proceso (Heavyweight Process):**
  - **Definición:** Es la entidad activa de un programa. No es solo el código (que es estático), sino el código en *ejecución* junto con sus recursos asignados.
  - **Aislamiento y Seguridad:** Cada proceso vive en una "burbuja" virtualizada. Si el proceso A escribe en la dirección de memoria `0x1234`, no afecta al proceso B, porque esa dirección es virtual y se mapea a lugares físicos distintos. Esto garantiza que un error en una aplicación (ej: Chrome) no corrompa los datos de otra (ej: Word).
  - **Coste de Contexto:** Cambiar de un proceso a otro (Context Switch) es costoso. El SO debe invalidar cachés (TLB), guardar tablas de páginas y restaurar un entorno totalmente nuevo.
- **Hilo (Lightweight Process):**
  - **Naturaleza Compartida:** Los hilos existen *dentro* de un proceso. Comparten la sección de código, la sección de datos (variables globales) y los recursos del sistema operativo (archivos abiertos, sockets).
  - **Lo que es Privado:** Para funcionar independientemente, cada hilo mantiene su propia **Pila (Stack)** (para llamadas a funciones y variables locales), **Registros de CPU** y **Contador de Programa**.
  - **Eficiencia:** La creación de hilos es rápida (10-100 veces más rápida que un proceso) porque no requiere duplicar el espacio de direcciones. El cambio de contexto entre hilos del mismo proceso es veloz, ya que la memoria virtual no cambia.
  - **Riesgo de Estabilidad:** Debido a la memoria compartida, un puntero mal direccionado en un hilo puede sobrescribir datos vitales de otro hilo, provocando un error fatal (`Segmentation Fault`) que termina todo el proceso inmediatamente.

Bloque de Control de Proceso (PCB): El "ADN" del Proceso

El PCB es la estructura administrativa más crítica del kernel. Sin ella, el SO vería la memoria como bytes sin sentido. Cada vez que un proceso se pausa, el SO toma una "foto" de su estado y la guarda aquí.

- **Identificación (PID/PPID):** No solo identifica al proceso (PID), sino también a su creador o padre (PPID), permitiendo estructurar árboles jerárquicos de procesos.
- **Estado del Proceso:** Indicador vital para el planificador (Scheduler). Define si el proceso es "elegible" para usar la CPU o si está esperando algo.
- **Contexto de Hardware (Registros):** Incluye el *Program Counter* (dónde me quedé), el *Stack Pointer* (dónde está mi pila) y los registros de propósito general. Restaurar esto permite reanudar la ejecución como si nunca se hubiera detenido.
- **Información de Planificación:** Prioridad estática y dinámica, punteros a colas de planificación y cuánto tiempo de CPU (quantum) ha consumido.
- **Gestión de Memoria:** Punteros a la Tabla de Páginas o Tabla de Segmentos que traducen las direcciones virtuales del proceso a direcciones físicas en la RAM.

- **Información de Contabilidad:** Tiempo real transcurrido, límites de tiempo, cuentas de usuario y grupos asociados.

Modos de Ejecución y Anillos de Protección

Para prevenir que un software malicioso o con errores destruya el sistema, las CPUs modernas implementan "Anillos de Protección" (Protection Rings).

1. **Modo Kernel (Ring 0):** Es el modo "dios". El código aquí puede ejecutar instrucciones privilegiadas como: "detener la CPU", "acceder directamente al disco duro" o "modificar las tablas de memoria". Un error aquí causa el temido "Pantallazo Azul" (BSOD) o "Kernel Panic".
2. **Modo Usuario (Ring 3):** Es un modo "seguro". Las aplicaciones normales (navegadores, juegos) corren aquí. No pueden tocar el hardware directamente.
   - **System Calls (Llamadas al Sistema):** Si una app necesita guardar un archivo, no escribe en el disco directamente. Invoca una *System Call* (ej: `write()`). La CPU cambia temporalmente a Modo Kernel, el SO verifica los permisos, realiza la operación de forma segura y devuelve el control al Modo Usuario.

Arquitectura del Kernel: Filosofías de Diseño

- **Núcleo Monolítico (Ej: Linux, UNIX tradicional, MS-DOS):**
  - **Filosofía:** "Todos para uno". El sistema de archivos, la pila de red, los drivers de video y la gestión de memoria conviven en un único espacio de memoria compartido en el kernel.
  - **Rendimiento:** Extremadamente alto, ya que los módulos se comunican mediante llamadas a funciones directas dentro de la memoria RAM.
  - **Fragilidad:** Un driver de impresora mal programado puede sobrescribir la memoria del gestor de archivos y colgar todo el ordenador.
- **Microkernel (Ej: Minix, QNX, Hurd):**
  - **Filosofía:** "Divide y vencerás". El kernel se reduce a lo mínimo absoluto (paso de mensajes y gestión básica de hardware). Todo lo demás (drivers, sistemas de archivos) son procesos normales aislados en espacio de usuario.
  - **Robustez:** Si el driver de audio falla, el kernel lo detecta y lo reinicia sin que el sistema se detenga. Es ideal para sistemas de misión crítica (aviones, medicina).
  - **Latencia:** La comunicación requiere paso de mensajes (IPC) constante entre el espacio de usuario y kernel, lo que introduce una sobrecarga (overhead) de rendimiento.

2. Gestión y Ciclo de Vida del Proceso

Secuencia de Arranque y Herramientas de Control

- **El Rol del Bootloader:**
  - Al encender el PC, la RAM está vacía. La BIOS/UEFI busca un dispositivo de arranque y carga el **Bootloader** (como GRUB en Linux o Windows Boot Manager).
  - Su tarea es compleja: debe reconocer sistemas de archivos, cargar el núcleo (kernel) y el disco RAM inicial (initrd) en memoria, y pasarle parámetros de arranque (como "modo seguro" o "modo verbose"). Una vez el kernel se carga, el bootloader muere y el SO toma el mando absoluto.
- **Comando `top` / `htop`:**
  - Es el "cuadro de mando" del administrador.
  - **Load Average:** Muestra la carga media del sistema en 1, 5 y 15 minutos. Si este número supera la cantidad de núcleos de tu CPU, significa que hay "atasco" (procesos esperando CPU).
  - **Estados (S):** Columna clave que muestra: `R` (Running), `S` (Sleeping), `D` (Disk Sleep - ininterrumpible), `Z` (Zombie).
  - **VIRT/RES:** Diferencia entre memoria virtual solicitada y memoria física real (RAM) ocupada.

Anomalías en el Ciclo de Vida: Zombies y Huérfanos

El ciclo de vida normal es: Padre crea Hijo (`fork`) -> Hijo termina (`exit`) -> Padre recoge estado (`wait`).

- **El Proceso Zombie (Defunct):**
  - Técnicamente, **todos** los procesos son zombies durante un instante tras morir. El problema surge cuando el padre *no* llama a `wait()` para leer el código de salida del hijo.
  - Aunque no consumen CPU ni RAM (su código ha sido liberado), ocupan un **PID**. Como los PIDs son finitos, una plaga de zombies puede impedir que se creen nuevos procesos en el sistema.
  - *Solución:* No se pueden matar con `kill` (ya están muertos). Hay que matar al proceso padre negligente para que el proceso `init` herede y limpie a los zombies.
- **El Proceso Huérfano:**
  - Ocurre cuando el padre muere antes que el hijo. En sistemas UNIX, los huérfanos son adoptados inmediatamente por el proceso PID 1 (`init` o `systemd`). Esto garantiza que siempre tengan un padre que recoja su estado final, evitando que se conviertan en zombies eternos.

Señales: El Lenguaje de Control de Procesos

Las señales son interrupciones de software enviadas a un proceso para notificar eventos.

- **SIGTERM (15):** Es como pedirle a alguien "Por favor, sal de la habitación". El programa recibe la señal, guarda sus documentos, cierra conexiones a bases de datos y termina limpiamente. Es la forma predeterminada del comando `kill`.
- **SIGKILL (9):** Es como teletransportar a la persona fuera de la habitación instantáneamente. El programa se detiene en la línea de código exacta donde estaba. **Peligro:** Puede dejar archivos corruptos, bases de datos inconsistentes o archivos temporales basura, ya que no se ejecutó ningún código de limpieza.

Capacidades del Sistema: La Ilusión de la Simultaneidad

- **Multitarea Apropiativa (Preemptive):** El SO moderno usa un reloj de hardware. Asigna un "cuanto" de tiempo (ej: 100ms) a un proceso. Si no termina, el SO interrumpe forzosamente la CPU (interrupción de reloj), guarda el estado y pasa al siguiente proceso. Esto evita que un programa colgado bloquee todo el PC.
- **Multiusuario Real:** Va más allá de tener varias carpetas. Implica la gestión de permisos (rwx) a nivel de kernel (UID/GID). Si el usuario "Ana" ejecuta un proceso, ese proceso hereda los permisos de Ana y no podrá leer archivos del usuario "Luis" a menos que se permita explícitamente.

3. Concurrencia y Sincronización: El Reto del Paralelismo

Problemas Críticos de Concurrencia

Cuando múltiples hilos operan sobre los mismos datos, el determinismo desaparece y nace el caos.

- **Condición de Carrera (Race Condition) - Analogía Bancaria:**
  - Imagina dos cajeros (hilos) intentando retirar $100 de una cuenta con $150 al mismo tiempo.
  - Ambos leen el saldo ($150). Ambos verifican que $150 > $100. Ambos aprueban la operación. Ambos restan $100 y escriben el nuevo saldo ($50).
  - Resultado real: Se retiraron $200 pero el saldo es $50. El banco perdió dinero. Esto es una condición de carrera: el resultado depende de la suerte en el orden de micro-instrucciones.
- **Sección Crítica:** Es el fragmento de código específico donde ocurre el acceso al recurso compartido (la línea donde se resta el dinero). El objetivo de la sincronización es proteger esta sección para que sea **atómica**.

Mecanismos de Sincronización Detallados

1. **Semáforos (Dijkstra):**
   - Son variables enteras protegidas por el SO.
   - **Semáforo Contador:** Permite el paso a un número limitado de hilos (ej: permitir solo 5 conexiones simultáneas a una base de datos).
   - **Flexibilidad vs Riesgo:** Son muy potentes, pero peligrosos. Si un hilo hace `wait` y olvida hacer `signal`, o si hace `signal` sin haber hecho `wait`, puede corromper la lógica del sistema. No distinguen *quién* incrementó o decrementó el valor.
2. **Mutex (Mutual Exclusion):**
   - Es un "candado" con llave.
   - **Propiedad de Dueño:** Si el Hilo A cierra el Mutex, *solo* el Hilo A puede abrirlo. Si el Hilo B intenta abrirlo, el SO genera un error. Esto previene muchos bugs humanos comunes en los semáforos.
   - **Recursividad:** Algunos mutex permiten que el mismo hilo lo bloquee varias veces (siempre que lo desbloquee igual número de veces), evitando auto-bloqueos.
3. **Monitores:**
   - Son construcciones del lenguaje de programación (como `synchronized` en Java). El compilador se encarga de insertar el código de entrada (lock) y salida (unlock) automáticamente alrededor de los métodos, reduciendo el error humano.

Escenarios de Fallo en Sincronización

- **Interbloqueo (Deadlock) - El Abrazo Mortal:**
  - Ocurre en sistemas complejos. Imagina un cruce de cuatro caminos donde cuatro coches avanzan y se bloquean entre sí. Nadie puede retroceder.
  - **Las 4 Condiciones de Coffman (Necesarias para Deadlock):**

1. *Exclusión Mutua:* Los recursos no se pueden compartir (ej: una impresora).
2. *Retención y Espera:* Tengo un recurso y pido otro.
3. *No Apropiamiento:* Nadie puede quitarme lo que tengo a la fuerza.
4. *Espera Circular:* A espera a B, B espera a C, C espera a A.

  - *Solución:* El SO debe romper una de estas condiciones (ej: matar un proceso o permitir preemption).
- **Inanición (Starvation):**
  - No es un bloqueo técnico, sino administrativo. Un algoritmo de planificación injusto siempre da prioridad a los procesos "VIP". Los procesos de baja prioridad pueden pasar años sin ejecutarse, aunque el sistema tenga recursos libres.
  - *Solución:* "Envejecimiento" (Aging) - aumentar gradualmente la prioridad de los procesos que llevan mucho tiempo esperando.

4. Comunicación entre Procesos (IPC) Avanzada

Dado que el SO aísla la memoria de los procesos, la IPC es necesaria para construir aplicaciones modulares.

- **Memoria Compartida (Shared Memory):**
  - **Mecánica:** El SO mapea un segmento de RAM en el espacio de direcciones de dos procesos distintos. Ambos ven los mismos bytes.
  - **Ventaja:** Velocidad "nativa". No hay copias de datos intermedias ni llamadas al sistema para leer/escribir.
  - **Desventaja:** Peligro extremo. El SO se retira una vez configurada la memoria. Los procesos son 100% responsables de usar semáforos para no escribir a la vez.
- **Tuberías (Pipes) y Flujos:**
  - **Pipes Anónimos (`|`):** Son volátiles. Viven en la memoria del kernel y mueren con los procesos. Son unidireccionales (Half-duplex) en la mayoría de sistemas: uno escribe, otro lee.

- **FIFOs (Named Pipes):** Persisten en el disco duro como un archivo especial. Un proceso puede escribir un mensaje hoy, apagarse, y otro proceso leerlo mañana (siempre que ambos se coordinen, pues actúan como tuberías bloqueantes por defecto).
- **Paso de Mensajes (Message Queues / Sockets):**
- A diferencia de la memoria compartida, aquí el SO copia los datos del Proceso A al kernel y luego del kernel al Proceso B.
- **Seguridad:** Es mucho más seguro y fácil de sincronizar (el mensaje llega o no llega).
- **Sockets:** La forma definitiva de IPC que permite comunicación no solo entre procesos de la misma máquina, sino entre procesos en diferentes continentes a través de Internet (TCP/IP).

Investigación Profunda: Problemas Clásicos de Concurrencia y Sincronización

1. Problema del Productor–Consumidor (Buffer Limitado)

Este es el paradigma fundamental de la cooperación entre procesos y modela situaciones cotidianas como un servidor web generando logs que un demonio de disco debe escribir, o un buffer de streaming de video.

- Escenario Ampliado:

Dos o más procesos comparten un búfer circular de tamaño fijo.

- **El Productor:** Genera flujos de datos y los deposita en el búfer. Si produce más rápido de lo que el consumidor puede procesar, el búfer se llena.
- **El Consumidor:** Extrae datos del búfer para procesarlos. Si es más rápido que el productor, el búfer se vacía.
- **Desafíos Críticos:**
  1. **Sincronización Condicional:** El productor debe bloquearse (dormir) si intenta añadir datos a un búfer lleno, despertando solo cuando haya espacio. Inversamente, el consumidor debe bloquearse si el búfer está vacío.
  2. **Exclusión Mutua:** Los punteros de inserción (`in`) y extracción (`out`) del búfer circular son variables compartidas. Si ambos procesos intentan actualizar el contador de elementos simultáneamente sin protección, se produce una condición de carrera que corrompe el estado del búfer.
  3. **Riesgo de Interbloqueo (Deadlock):** Un error común es adquirir el acceso exclusivo (`mutex`) *antes* de comprobar si hay espacio disponible. Si el búfer está lleno y el productor bloquea el mutex, el consumidor nunca podrá entrar para vaciar un hueco, causando un bloqueo eterno.
- Solución Arquitectónica:

Se emplean tres semáforos para orquestar la danza:

- `mutex` (Semáforo Binario): Garantiza que solo uno toque los punteros del búfer a la vez.
- `vacios` (Semáforo Contador): Inicializado en N (tamaño del búfer). Representa los "permisos para escribir".
- `llenos` (Semáforo Contador): Inicializado en 0. Representa los "recursos disponibles para consumir".

2. Problema de Lectores y Escritores

Este modelo es crucial para el diseño de bases de datos y sistemas de archivos donde las operaciones de lectura son frecuentes y seguras en paralelo, pero las escrituras son críticas y destructivas.

- Escenario y Reglas:

Muchos procesos desean acceder a un recurso compartido (archivo/registro).

- **Lectores:** Pueden acceder simultáneamente sin límite, ya que no modifican datos.
- **Escritores:** Requieren exclusión mutua absoluta. Si un escritor está activo, nadie más (ni lectores ni otros escritores) puede acceder.

- El Dilema de la Prioridad (Inanición):

  Existen dos variantes principales según a quién se le dé preferencia:

  1. **Prioridad a los Lectores (Primer Problema):** Ningún lector espera si el archivo ya está abierto para lectura.
- *Consecuencia:* Si llegan lectores continuamente, un escritor podría esperar indefinidamente (**inanición del escritor**). Se usa el mecanismo de "interruptor de luz": el primer lector que entra bloquea la sala para los escritores, y el último que sale la desbloquea.
  2. **Prioridad a los Escritores (Segundo Problema):** Tan pronto como un escritor solicita acceso, se prohíben nuevas lecturas.
- *Consecuencia:* Los lectores actuales terminan, pero los nuevos se bloquean hasta que el escritor finalice. Esto minimiza el retraso de los datos frescos pero puede causar **inanición de los lectores**.

3. Problema del Barbero Dormilón (Sleeping Barber)

Ilustra los desafíos en sistemas de colas y coordinación cliente-servidor (como un Help Desk o un servidor web con pool de hilos).

- Escenario Detallado:

  Una barbería tiene un barbero, una silla de trabajo y una sala de espera con N sillas.

- **Dinámica de Eventos:**
- **Estado Ocioso:** Si no hay clientes, el barbero se sienta en su silla y duerme (bloqueo esperando señal).
- **Llegada de Cliente:**
- Si el barbero duerme, el cliente lo despierta (envía señal) y ocupa la silla de barbero.
- Si el barbero está cortando el pelo, el cliente revisa la sala de espera. Si hay sillas libres, se sienta (se encola). Si no, se marcha inmediatamente (petición rechazada/dropped).

- Desafío de Sincronización:

  El problema real es evitar condiciones de carrera entre la comprobación de las sillas y la acción de sentarse.

- *Ejemplo de Fallo:* Un cliente entra, ve al barbero ocupado y decide sentarse. Justo en ese instante, el barbero termina, revisa la sala de espera, la ve "vacía" (porque el cliente aún no se ha sentado formalmente) y se va a dormir. El cliente se sienta esperando ser llamado. Resultado: Barbero durmiendo y cliente esperando eternamente. Se requiere un `mutex` para proteger la acción de "entrar a la sala de espera".

4. Problema de los Fumadores y el Agente (Cigarette Smokers)

Propuesto por Suhas Patil, este problema demuestra que los semáforos tradicionales no son suficientes para resolver problemas donde se requieren múltiples recursos simultáneos de tipos específicos.

- Escenario:

  Hay tres fumadores. Cada uno posee infinitos suministros de un solo ingrediente: (A) Tabaco, (B) Papel, (C) Cerillas.

Un "Agente" (el SO) coloca aleatoriamente dos ingredientes en la mesa (ej: Tabaco y Papel).

- El Conflicto:

  El fumador que tiene las Cerillas (C) necesita los recursos (A) y (B) de la mesa para fumar.

  - *Limitación:* Los semáforos simples no permiten preguntar "¿Están A y B disponibles?". Si el fumador A toma el papel de la mesa pero el tabaco no está (o lo tomó otro por error), se bloquea reteniendo un recurso vital, pudiendo causar un **Interbloqueo**.
- Solución Compleja:

  Se requiere introducir hilos intermediarios ("pushers" o ayudantes) que verifiquen el estado de la mesa y solo despierten al fumador correcto cuando ambos recursos estén garantizados. Esto ilustra la necesidad de primitivas más avanzadas o lógica de orquestación adicional.

5. Problema de los Cochecitos del Río (River Crossing)

Conocido también como el problema de los Hackers y Serfs (o Misioneros y Caníbales), modela la sincronización de barrera.

- Escenario y Restricciones:

  Pasajeros de dos bandos (Linuxeros y Microsofties) deben cruzar un río.

  - **Regla de Capacidad:** La barca solo sale con exactamente 4 pasajeros.
  - **Regla de Seguridad:** Para evitar peleas, la tripulación debe ser homogénea (4 del mismo bando) o perfectamente mixta (2 de cada bando). Una combinación de 3 vs 1 es insegura y está prohibida.
- Implementación y Desafío:

  Los hilos llegan de forma aleatoria. Deben "esperar en el muelle" hasta que se forme una combinación válida.

  - **La Barrera:** Se necesitan contadores para saber cuántos de cada tipo están esperando.
  - **El Capitán:** El último hilo que llega y completa una combinación válida (ej: llega el segundo Hacker y ya había 2 Serfs esperando) asume el rol de "Capitán". Este hilo libera los semáforos de los otros 3 pasajeros, llama a la función `Remar()` y asegura que nadie más suba hasta que el viaje termine.

6. Problema de Interbloqueo Circular o Doble Recurso

Es el caso de estudio más simple y común de *Deadlock* en sistemas operativos y bases de datos.

- Secuencia Fatal (Hold and Wait):

  Dos procesos (P1, P2) y dos recursos (R1, R2).

1. P1 adquiere el bloqueo de R1.
2. P2 adquiere el bloqueo de R2.
3. P1 intenta adquirir R2. Como lo tiene P2, P1 se bloquea (espera).
4. P2 intenta adquirir R1. Como lo tiene P1, P2 se bloquea (espera).

- **Resultado:** Un ciclo de dependencia infinito. Ninguno cede, ninguno avanza.
- **Estrategias de Prevención:**

- **Ordenación de Recursos (Resource Ordering):** Se impone una jerarquía global. Todos los procesos deben solicitar recursos en el mismo orden (ej: Primero R1, luego R2). Esto hace matemáticamente imposible que se forme un ciclo, ya que nadie puede tener un recurso "mayor" y pedir uno "menor".

7. Problema del Bufón o del Turno Justo (Fair Turn)

Este problema ataca la falta de justicia (*fairness*) en los algoritmos de sincronización estándar.

- Escenario:

Varios procesos compiten por una sección crítica protegida por un cerrojo (lock).

- *El problema:* Un bloqueo estándar no garantiza orden. Si 5 procesos esperan, y el bloqueo se libera, el hardware o el SO eligen al "ganador" arbitrariamente. Un proceso con mala suerte podría esperar indefinidamente mientras otros entran y salen repetidamente (**Inanición**).
- Analogía y Solución:

Similar a una panadería o carnicería ("Tome su turno").

- **Algoritmo del Ticket (Bakery Algorithm):**

1. Al llegar, el proceso toma un número: `mi_ticket = max(tickets_actuales) + 1`.
2. Existe una pantalla global: `turno_actual`.
3. El proceso espera en un bucle: `while (mi_ticket != turno_actual) esperar;`.
4. Al salir, incrementa `turno_actual`.

- Esto impone un orden **FIFO (First-Come, First-Served)** estricto, eliminando la inanición.

8. Algoritmos de Exclusión Mutua Clásicos

Evolución histórica de las soluciones por software antes de tener soporte de hardware atómico.

- Algoritmo de Dekker (1965):

El primer algoritmo correcto para 2 procesos. Combina la idea de "turnos" (para resolver conflictos estrictos) con "banderas de intención" (para evitar bloqueos si el otro no quiere entrar). Es complejo de entender y difícil de generalizar a N procesos.

- Algoritmo de Peterson (1981):

Una simplificación elegante de Dekker.

- *Lógica:* Cada proceso tiene una bandera `flag[i]` ("quiero entrar") y hay una variable compartida `turn`.
- *Cortesía:* Para entrar, un proceso dice "Quiero entrar" (`flag[yo] = true`) pero cede el paso preventivamente (`turn = otro`). Solo entra si el otro no quiere O si es su turno.
- Algoritmo de Lamport (Bakery):

Diseñado para N procesos. Utiliza la lógica de tickets numerados. Es robusto incluso si los tickets se reinician, pero requiere gestión cuidadosa de la memoria compartida.

Preguntas de Repaso Ampliadas

1. ¿Cuál es el principal riesgo en el problema de los Lectores y Escritores y cómo se manifiesta en ambos casos?

El riesgo fundamental es la **inanición (starvation)**.

- En la solución de **Prioridad a Lectores**, si hay un flujo constante de lectores (siempre hay alguien leyendo), los escritores nunca obtendrán el bloqueo y sus actualizaciones nunca se escribirán en la base de datos.
- En la solución de **Prioridad a Escritores**, si se privilegia a los escritores, los lectores pueden quedar bloqueados en masa durante largos periodos, afectando la latencia de lectura del sistema.

2. En el problema del Productor-Consumidor, ¿por qué el orden de los `wait` (o `down`) es crítico para evitar Deadlocks?

El orden es vital. El proceso debe "reservar espacio" (semáforo contador) antes de "entrar a la zona crítica" (semáforo mutex).

Si un productor adquiere el mutex (bloqueo exclusivo del búfer) y luego intenta hacer wait(vacios) cuando el búfer está lleno, se bloqueará durmiendo mientras retiene el mutex. El consumidor, que necesita el mutex para sacar elementos y liberar espacio, nunca podrá entrar. Se produce un interbloqueo inmediato.

3. Explique la condición de seguridad y el rol del "Capitán" en el problema de los Cochecitos del Río.

La condición de seguridad evita conflictos (3 vs 1). La barrera solo se abre para grupos de 4 del mismo bando o 2 y 2.

El "Capitán" no es un hilo especial predeterminado, sino el hilo que llega en último lugar y detecta que se cumple una combinación válida. Su responsabilidad es orquestar la salida: despierta a los otros 3 hilos (que estaban dormidos en la barrera), realiza la acción de "Remar" y libera el bloqueo (mutex) del muelle para permitir que nuevos pasajeros comiencen a agruparse.

4. ¿Qué diferencia conceptual existe entre el Algoritmo de Peterson y una instrucción de hardware `Test-and-Set`?

El Algoritmo de Peterson es una solución puramente de software; funciona mediante lógica de programación y variables compartidas normales, pero es lento y difícil de escalar a múltiples CPUs debido a la coherencia de caché.

La instrucción Test-and-Set es una solución de hardware; es una instrucción atómica de la CPU que lee y modifica una variable en un solo ciclo de reloj ininterrumpible. Es la base sobre la que se construyen los mutex y semáforos modernos en los sistemas operativos.

5. ¿Por qué el problema de los Fumadores requiere un "Agente" o hilos intermediarios?

El problema radica en la incapacidad de un hilo para esperar por "Múltiples Recursos" simultáneamente con semáforos básicos. Un fumador necesita Tabaco Y Papel. Si intenta tomar Tabaco y lo consigue, pero el Papel no está, se queda con el Tabaco bloqueado.

El Agente (o los intermediarios "Pushers") resuelve esto al tener una visión global o al despertar selectivamente solo al proceso que tiene garantizados ambos recursos, evitando que se retengan recursos parciales que causarían Deadlock.

6. ¿Cómo el Algoritmo del Ticket (Bakery) garantiza la propiedad de "Progreso" y "Espera Limitada"?

Garantiza Progreso porque si nadie está en la sección crítica, el proceso con el ticket más bajo entra inmediatamente.

Garantiza Espera Limitada (evita inanición) mediante el orden secuencial de los tickets. Como los tickets se asignan incrementalmente ($N, N+1, N+2...$), un proceso recién llegado siempre tendrá un turno posterior a los que ya están esperando. Eventualmente, todos los anteriores serán atendidos y le tocará su turno, sin importar la velocidad relativa de los procesos.

7. En el problema del Barbero Dormilón, ¿qué función cumple el semáforo de "Clientes" versus el semáforo de "Barbero"?

Son mecanismos de señalización opuestos:

- El semáforo **Clientes** actúa como un contador de "trabajo pendiente". El barbero hace `wait(Clientes)` para dormir si es 0, y los clientes hacen `signal(Clientes)` para despertarlo.
- El semáforo **Barbero** (o su silla) actúa como señal de "estoy listo para cortar". El cliente hace `wait(Barbero)` para esperar a que le corten el pelo, asegurando que no se vaya hasta que el corte termine.

8. Describa cómo la estrategia de "Ordenación de Recursos" previene el Interbloqueo Circular.

El interbloqueo circular requiere que A tenga R1 y pida R2, mientras B tiene R2 y pide R1.

Si imponemos una regla global de que los recursos deben pedirse en orden numérico (R1 antes que R2), la situación anterior es imposible:

- Si A tiene R1, puede pedir R2 (orden correcto).
- Pero B no puede tener R2 y pedir R1, porque eso violaría la regla de orden (pedir R2 antes que R1). B tendría que haber pedido R1 primero.

Al romper la posibilidad de ciclos en el grafo de asignación de recursos, se garantiza matemáticamente que nunca ocurrirá un interbloqueo circular.