Proyecto I - Variantes de Problemas Clásicos de Sincronización en Sistemas Operativos y Redes

Fecha de Entrega y Presentación 09 de Mayo 2025

A continuación, se presentan variantes detalladas de los problemas clásicos de sincronización: Lectores y Escritores, Productor-Consumidor, Cena de los Filósofos y Barbero Dormilón. Todas las variantes requieren implementación usando hebras (threads) y mecanismos de sincronización como semáforos y variables mutex en C bajo Linux. Además, cada variante debe incluir una salida por texto que permita observar el comportamiento concurrente paso a paso. Se indica el mínimo y máximo número de hebras necesarias para cada caso.

# Lectores y Escritores

## Variante 1: Prioridad a Escritores (Manuela Duarte - Fernanda Catalán)

Hilos necesarios: mínimo 2, ideal hasta 5.

**Motivación:** Evitar que **escritores queden bloqueados indefinidamente** si siguen llegando lectores constantemente (inanición del escritor).

**Problema a resolver:** En la solución tradicional, una alta frecuencia de lectores puede impedir que un escritor obtenga el control.

**Estrategia de solución:**

* Si un escritor desea escribir, **se bloquea el ingreso de nuevos lectores**, aunque los que ya estén leyendo puedan terminar.
* Se usa un contador escritores\_esperando para marcar la intención de escritura.

**Estructuras necesarias:**

* mutex: protege contadores.
* sem\_lector, sem\_escritor: controlan entrada de lectores/escritores.
* int lectores\_leyendo = 0;
* int escritores\_esperando = 0;

**Ventajas:**

* Garantiza que los escritores no se queden esperando indefinidamente.
* Ideal para sistemas donde la actualización es crítica.

**Desventajas:**

* Los lectores pueden experimentar **retrasos frecuentes** si hay escritores esperando.

**Ejemplo de salida esperada:**

[Lector 1] intentando leer...

[Escritor 1] desea escribir. Bloqueando nuevos lectores.

[Lector 1] leyendo...

[Lector 1] termina de leer.

[Escritor 1] escribiendo...

## Variante 2: Acceso justo (FIFO) (Juan Pablo Acevedo - Bastián Andrade)

Hilos necesarios: mínimo 3, ideal hasta 6.

**Motivación:** Evitar la **inanición tanto de lectores como de escritores**, implementando acceso en **orden de llegada** (justicia FIFO).

**Problema a resolver:** Ni lectores ni escritores deben quedar en espera indefinidamente.

**Estrategia de solución:**

* Uso de una **cola de turnos** o una lista FIFO de solicitudes.
* Cada hilo toma un número de turno, y espera hasta que le toque.

**Estructuras necesarias:**

* cola\_turnos[]: cola compartida con tipo de operación (leer/escribir).
* mutex: para proteger el acceso a la cola.
* semáforo individual por hilo o condición basada en turno.

**Ventajas:**

* Acceso totalmente justo.
* Garantiza equidad a largo plazo.

**Desventajas:**

* Alta complejidad en la implementación.
* Puede ser menos eficiente si los escritores son lentos.

**Ejemplo de salida esperada:**

[Lector 1] solicita turno 1.

[Escritor 1] solicita turno 2.

[Lector 2] solicita turno 3.

[Lector 1] accede (turno 1).

[Escritor 1] accede (turno 2).

[Lector 2] accede (turno 3).

## Variante 3: Lector preferente con límite (Martín Vera - Vicente Farias)

Hilos necesarios: mínimo 3, ideal hasta 6.

**Motivación:** Evitar que los escritores sean bloqueados **sin eliminar la eficiencia de la lectura simultánea**.

**Problema a resolver:** Lectores que se suceden uno tras otro **pueden bloquear a los escritores indefinidamente**.

**Estrategia de solución:**

* Se permite que N lectores lean consecutivamente.
* Tras cierto umbral (MAX\_LECTURAS\_CONSECUTIVAS), se **fuerza el paso a un escritor**.

**Estructuras necesarias:**

* int lecturas\_consecutivas = 0;
* mutex: protege las variables compartidas.
* sem\_lector, sem\_escritor: bloquean o liberan paso.
* Variable para saber si hay escritores esperando.

**Ventajas:**

* Conserva la eficiencia de múltiples lectores.
* Evita inanición de escritores sin sacrificar rendimiento.

**Desventajas:**

* Más variables de control.
* Difícil ajustar el umbral correctamente para todos los escenarios.

**Ejemplo de salida esperada:**

[Lector 1] leyendo (lectura 1/5)

[Lector 2] leyendo (lectura 2/5)

...

[Lector 5] leyendo (lectura 5/5)

[Escritor 1] esperando turno tras 5 lecturas.

[Lector 6] bloqueado temporalmente.

[Escritor 1] escribiendo...

# Productor-Consumidor

## Variante 1: Buffer sin límites (Martín Bascuñán - Benjamín Paz)

Hilos necesarios: mínimo 2, ideal hasta 6.

**Motivación:** Explorar un sistema donde el buffer puede **crecer dinámicamente**, simulando una cola sin tope de capacidad (como una lista enlazada).

**Problema a resolver:** ¿Cómo manejar correctamente la concurrencia cuando no existe límite físico, pero sí se debe evitar uso excesivo de recursos?

**Estrategia de solución:**

* Buffer representado con una lista o cola dinámica (linked list).
* Los productores agregan elementos; los consumidores los remueven.
* El sistema puede incorporar una política para desacelerar producción si el buffer crece demasiado.

**Estructuras necesarias:**

* pthread\_mutex\_t mutex\_buffer
* sem\_t items: cuenta cuántos ítems hay disponibles.
* Lista enlazada o cola dinámica protegida por el mutex.

**Ventajas:**

* Útil para simular cargas variables (ej. servidores web, sistemas asincrónicos).
* Permite observar crecimiento del buffer en tiempo real.

**Desventajas:**

* Si los consumidores son lentos, el buffer **puede crecer sin control**.
* Difícil de gestionar en entornos con recursos limitados.

**Ejemplo de salida esperada:**

[Productor 1] produjo item A

[Buffer] Estado actual: [A]

[Productor 2] produjo item B

[Buffer] Estado actual: [A B]

[Consumidor 1] consumió item A

[Buffer] Estado actual: [B]

## Variante 2: Múltiples productores y consumidores (Matías Oyarzún - Matías Peters)

Hilos necesarios: mínimo 4, ideal hasta 10.

**Motivación:** Simular sistemas reales con múltiples fuentes de datos y múltiples procesos que los consumen (como colas de trabajos).

**Problema a resolver:** Asegurar que múltiples productores y consumidores **no corrompan el estado del buffer** compartido.

**Estrategia de solución:**

* Usar un buffer circular de tamaño fijo.
* Controlar acceso concurrente con semáforos y mutex.

**Estructuras necesarias:**

* sem\_t empty: número de espacios vacíos.
* sem\_t full: número de ítems listos para consumir.
* pthread\_mutex\_t mutex: protege los accesos simultáneos.
* buffer[] circular con índices in, out.

**Ventajas:**

* Escalable y eficiente.
* Simula sistemas de procesamiento paralelos.

**Desventajas:**

* Requiere sincronización rigurosa para evitar errores como doble acceso o condiciones de carrera.
* Difícil de depurar si hay muchos hilos.

**Ejemplo de salida esperada:**

[Productor 1] produjo en posición 2

[Productor 3] producido en posición 3

[Consumidor 1] consumió de posición 2

[Consumidor 2] consumió de posición 3

## Variante 3: Buffer circular (José Hernández- Fabián Arévalo)

Hilos necesarios: mínimo 2, ideal hasta 6.

**Motivación:** Simular un sistema con **reciclaje de memoria** eficiente: cuando el buffer se llena, se reinicia el índice circular.

**Problema a resolver:** Cómo mantener el uso eficiente del espacio del buffer sin sobrescribir ítems aún no consumidos.

**Estrategia de solución:**

* Índices in y out avanzan circularmente: i = (i + 1) % N.
* Se bloquea al productor si el buffer está lleno y al consumidor si está vacío.
* Necesita semáforos y mutex.

**Estructuras necesarias:**

* buffer[N], in, out
* sem\_t empty, sem\_t full
* pthread\_mutex\_t mutex

**Ventajas:**

* Uso eficiente de memoria.
* Ideal para buffers de streaming o procesamiento en tiempo real.

**Desventajas:**

* Difícil de visualizar si no se imprime el estado del buffer frecuentemente.
* Necesita control estricto de los índices.

**Ejemplo de salida esperada:**

[in=4] [out=2] → [Productor 1] produjo 'X'

[Buffer] Estado: [\_ \_ A B X \_ \_ \_]

[in=5] [out=2] → [Consumidor 2] extrajo 'A'

# Cena de los Filósofos

## Variante 1: Palillos individuales (Eduardo Cabezas - Franko Navarrete)

Hilos necesarios: mínimo 5, ideal hasta 10.

**Motivación:** El modelo clásico genera **interbloqueo (deadlock)** cuando todos los filósofos toman su palillo izquierdo y luego esperan indefinidamente el derecho. Esta variante busca **romper el ciclo de espera circular**.

**Problema a resolver:** Evitar que todos los filósofos esperen eternamente bloqueando los palillos vecinos.

**Estrategia:** Uno de los filósofos (por ejemplo, el último) **invierte el orden** de toma de palillos: toma primero el derecho y luego el izquierdo.

**Estructuras necesarias:**

* 5 mutexes (pthread\_mutex\_t palillos[5])
* 5 hebras (pthread\_t filosofos[5])
* Función tomar\_palillos(int id) y soltar\_palillos(int id)

**Ventajas:**

* Sencilla de implementar.
* Evita deadlock sin requerir estructuras adicionales.

**Desventajas:**

* No evita **inanición**: un filósofo podría comer mucho menos que otros.

**Ejemplo de salida esperada:**

[F0] pensando...

[F0] quiere comer.

[F0] toma palillos 0 y 1.

[F0] comiendo...

[F1] quiere comer.

[F2] quiere comer.

[F0] suelta palillos 0 y 1.

[F1] toma palillos 1 y 2.

[F1] comiendo...

## Variante 2: Mayordomo (Martín Álvarez - Víctor Barraza)

Hilos necesarios: mínimo 6, ideal hasta 6.

**Motivación:** Eliminar el riesgo de deadlock **limitando el número de filósofos que intentan comer simultáneamente**.

**Problema a resolver:** Evitar interbloqueo de manera general y estructurada, sin depender de qué filósofo invierte el orden.

**Estrategia:** Un semáforo (sem\_t mayordomo) inicializado en N−1 permite que como máximo N−1 filósofos estén en la mesa al mismo tiempo.

**Estructuras necesarias:**

* 5 mutexes para palillos
* 1 semáforo binario o contador (sem\_init(&mayordomo, 0, 4))
* Funciones sentarse(), levantarse()

**Ventajas:**

* Garantiza ausencia de deadlock.
* Solución genérica aplicable a N filósofos.

**Desventajas:**

* Aumenta la complejidad del código.
* Requiere mayor coordinación.

**Ejemplo de salida esperada:**

[F3] solicita permiso al mayordomo.

[F3] obtiene permiso y se sienta.

[F3] toma palillos 3 y 4.

[F3] comiendo...

[F4] solicita permiso al mayordomo.

[F4] debe esperar (ya hay 4 comiendo o intentando comer).

## Variante 3: Tiempos aleatorios (Felipe Pérez)

Hilos necesarios: mínimo 5, ideal hasta 8.

**Motivación:** Simular un comportamiento más realista e introducir **aleatoriedad para reducir sincronización exacta** que pueda llevar a conflictos.

**Problema a resolver:** Evitar condiciones de carrera y estudiar situaciones donde uno o más filósofos sean frecuentemente postergados.

**Estrategia:**

* Cada filósofo alterna entre “pensar” y “comer” con sleep(rand() % t)
* Controlar los estados (pensando, hambriento, comiendo) usando un arreglo de estados global.
* Solo se permite comer si **ningún vecino está comiendo**.

**Estructuras necesarias:**

* Array estado[5] (enum)
* Semáforos por filósofo (sem\_t s[5])
* Mutex general para proteger cambios de estado.

**Ventajas:**

* Equitativa: reduce inanición.
* Realista: simula variaciones de comportamiento.

**Desventajas:**

* Requiere lógica más compleja.
* Difícil de depurar si no se observa la salida cuidadosamente.

**Ejemplo de salida esperada:**

[F1] pensando por 2s...

[F1] hambriento.

[F1] no puede comer, F0 está comiendo.

[F2] pensando por 4s...

[F1] intenta comer de nuevo...

[F1] ahora sí puede comer.

[F1] comiendo.

# Barbero Dormilón

## Variante 1: Múltiples barberos (Jonathan Catalán)

Hilos necesarios: mínimo 3, ideal hasta 6.

**Motivación:**  
Expandir el problema original (con un solo barbero) para manejar **múltiples barberos trabajando en paralelo**, como ocurre en barberías reales.

**🔧 Problema a resolver:**  
Sincronizar adecuadamente el acceso de múltiples clientes con múltiples barberos sin que se atiendan a la vez al mismo cliente ni se pierdan turnos.

**Estrategia de solución:** Cada barbero es un hilo que espera clientes.

* Se utiliza una **sala de espera compartida** y **una cola FIFO** para el orden de atención.
* Los barberos duermen si no hay clientes.

**Estructuras necesarias:**

* sem\_t clientes; // incrementado por clientes cuando llegan
* sem\_t barberos[N]; // uno por cada barbero, señalizado por el cliente cuando quiere ser atendido por ese barbero
* pthread\_mutex\_t acceso\_sala; // protege el acceso a la cola de espera
* Cola clientes\_esperando

**Ventajas:**

* Refleja escenarios reales.
* Aumenta la capacidad de atención.

**Desventajas:**

* Requiere asignar clientes a barberos de forma segura.
* Más complejidad en la lógica de distribución.

**Ejemplo de salida esperada:**

[Cliente 1] entra a la barbería.

[Cliente 1] se sienta en la sala de espera.

[Barbero 1] despierta y atiende al Cliente 1.

[Cliente 2] entra.

[Cliente 2] se sienta.

[Barbero 2] atiende al Cliente 2.

## Variante 2: Clientes impacientes (Daniel Muñoz)

Hilos necesarios: mínimo 3, ideal hasta 6.

**Motivación:** Introducir un comportamiento más realista: **no todos los clientes están dispuestos a esperar indefinidamente**.

**Problema a resolver:** Evitar el crecimiento ilimitado de clientes esperando, y simular que los clientes pueden retirarse.

**Estrategia de solución:**

* Se define un número fijo de sillas (MAX\_SILLAS).
* Si no hay sillas libres, el cliente **se retira** sin ser atendido.
* Se muestra en la salida si un cliente fue atendido o no.

**Estructuras necesarias:**

* int sillas\_ocupadas; protegido por mutex
* sem\_t clientes; y sem\_t barbero;
* Condición: if (sillas\_ocupadas == MAX\_SILLAS) → cliente se va.

**Ventajas:**

* Mejora la simulación de sistemas de colas reales.
* Controla carga del sistema.

**Desventajas:**

* Algunos clientes pueden perderse.
* Requiere control estricto del contador de sillas.

**Ejemplo de salida esperada:**

[Cliente 3] entra a la barbería.

[Cliente 3] se sienta. Sillas ocupadas: 2/3.

[Cliente 4] entra. Sillas ocupadas: 3/3.

[Cliente 5] entra. No hay sillas. Se retira.

[Barbero] atiende al Cliente 3.

## Variante 3: Prioridad por tipo de cliente (Paulo Gutiérrez - Ivonne Santander)

Hilos necesarios: mínimo 3, ideal hasta 6.

**Motivación:** Simular un escenario donde **algunos clientes tienen prioridad** (ej. VIP), lo cual es común en servicios preferenciales.

**Problema a resolver:** Asegurar que los **clientes VIP sean atendidos antes que los normales**, sin causar inanición de los clientes estándar.

**Estrategia de solución:**

* Usar una cola de prioridad (dos listas separadas o una heap).
* El barbero siempre atiende primero a un cliente VIP si hay alguno esperando.
* Usar pthread\_cond\_t o lógica con semáforos separados.

**Estructuras necesarias:**

* cola\_vip y cola\_normal
* Mutex para proteger acceso a ambas colas
* Semáforo por tipo de cliente o política de turnos

**Ventajas:**

* Permite simular prioridades y políticas diferenciadas.
* Útil para estudiar justicia en planificación.

**Desventajas:**

* Si hay muchos VIPs, los normales podrían experimentar inanición.
* Implementación más compleja.

**Ejemplo de salida esperada:**

[Cliente normal 1] se sienta.

[Cliente VIP 1] entra y se sienta.

[Barbero] atiende a Cliente VIP 1.

[Cliente VIP 2] entra. Cliente normal 1 sigue esperando.

[Barbero] atiende a Cliente VIP 2.

[Barbero] finalmente atiende a Cliente normal 1.

# Rúbrica de Evaluación

# 1. Implementación (40%)

## a. Calidad de las variables (10 pts)

|  |  |
| --- | --- |
| Puntaje | Descripción |
| 0 pts | Uso de nombres genéricos o confusos; variables sin relación con su función. |
| 3 pts | Algunos nombres son representativos, pero hay inconsistencias o repeticiones innecesarias. |
| 7 pts | Mayoría de nombres claros y coherentes, pero con algunas excepciones menores. |
| 10 pts | Todos los nombres de variables son descriptivos, claros y coherentes con las buenas prácticas. |

## b. Uso de comentarios en el código (10 pts)

|  |  |
| --- | --- |
| Puntaje | Descripción |
| 0 pts | Código sin comentarios o con comentarios irrelevantes. |
| 3 pts | Comentarios presentes en pocas secciones, no explican partes clave. |
| 7 pts | Comentarios en partes importantes pero con falta de claridad o profundidad. |
| 10 pts | Comentarios explicativos en cada sección clave del código, útiles para comprender la lógica del programa. |

## c. Estructura modular del código (10 pts)

|  |  |
| --- | --- |
| Puntaje | Descripción |
| 0 pts | Todo el código está en la función principal. |
| 3 pts | Una o dos funciones definidas, pero sin claridad o reutilización. |
| 7 pts | Código dividido en funciones lógicas, pero algunas mal diseñadas o redundantes. |
| 10 pts | Código claramente modularizado, cada función cumple una única responsabilidad. |

## d. Cumplimiento del problema planteado (10 pts)

|  |  |
| --- | --- |
| Puntaje | Descripción |
| 0 pts | El programa no compila o no aborda la variante solicitada. |
| 3 pts | La estructura general es coherente, pero faltan elementos obligatorios (hebras, mutex, semáforos). |
| 7 pts | Implementación funcional con pequeños errores o ausencias menores. |
| 10 pts | Implementación completa y funcional de la variante, con sincronización correcta y salida coherente. |

# 2. Presentación (60%)

## a. Disertación y discurso (20 pts)

|  |  |
| --- | --- |
| Puntaje | Descripción |
| 0 pts | Lectura del código sin explicación; lenguaje poco técnico. |
| 10 pts | Presentación con ideas generales; falta profundidad o claridad técnica. |
| 15 pts | Discurso bien estructurado, pero con fallas menores en la precisión técnica. |
| 20 pts | Explicación clara, lógica, con buen manejo técnico del tema y sus conceptos. |

## b. Demostración práctica (15 pts)

|  |  |
| --- | --- |
| Puntaje | Descripción |
| 0 pts | No se presenta ejecución del código. |
| 8 pts | Código ejecuta, pero con errores o sin claridad en lo que se observa. |
| 12 pts | Ejecución correcta, pero con fallas en cómo se destaca la sincronización. |
| 15 pts | Ejecución fluida, sincronización visible y explicada en contexto. |

## c. Control del tiempo (5 pts)

|  |  |
| --- | --- |
| Puntaje | Descripción |
| 0 pts | Menos de 6 minutos o más de 12 minutos. |
| 3 pts | Entre 6 y 7 min o entre 10 y 12 min. |
| 5 pts | Entre 8 y 10 minutos. |

## d. Respuestas a preguntas del docente (20 pts)

|  |  |
| --- | --- |
| Puntaje | Descripción |
| 0 pts | No responde o desconoce detalles fundamentales del código. |
| 10 pts | Responde algunas preguntas, pero con vacilación o imprecisión técnica. |
| 15 pts | Responde correctamente con algunos errores menores. |
| 20 pts | Responde con seguridad, claridad técnica y justificación del diseño. |