UNIVERSIDAD DE ANTIOQUIA
CURSO DE SISTEMAS OPERATIVOS
APUNTES Y REPASO DE LOS TEMAS:
PASTEL PARA PARCIAL 2

SEMÁFOROS Y CONCURRENCIA

I. SEMÁFOROS

El propósito de estos apuntes es repasar los conceptos vistos en clase de Sistemas Operativos con los profesores Henry y Danny, y ampliarlos de manera investigativa analizando en profundidad el funcionamiento, las propiedades y las aplicaciones de los semáforos en sistemas operativos, identificando ventajas, limitaciones y casos de uso típicos. De igual manera, dentro los aprendizajes del curso es imperativo aprender a:

- Describir el modelo teórico de semáforos, incluyendo las operaciones atómicas P y V y sus invariantes formales.
- Clasificar los tipos de semáforos (binarios, de conteo, con prioridad) y compararlos con otros mecanismos de sincronización (mutex, monitores, variables de condición).
- Presentar patrones de uso comunes en problemas clásicos de concurrencia (productor-consumidor, lectores-escritores, barreras).
- Implementar ejemplos en lenguaje C utilizando la API de semáforos POSIX y analizar su comportamiento.
- Evaluar problemas avanzados como deadlocks, inanición y estrategias de prevención, así como el rendimiento en sistemas multicore.

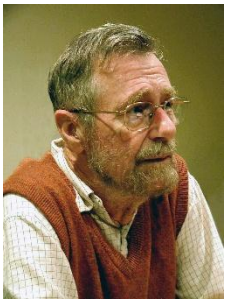
1. Introducción

Los sistemas operativos modernos deben gestionar concurrentemente múltiples procesos o hilos que compiten por recursos compartidos. En este contexto, los semáforos se erigen como uno de los mecanismos fundamentales para la sincronización y el control de acceso a estos recursos, evitando condiciones de carrera, *deadlocks* y garantizando la coherencia de los datos.

1.1. Contexto y motivación de los semáforos en la sincronización de procesos

En arquitecturas multiprocesador y multinúcleo, los procesos o hilos pueden ejecutarse de forma concurrente, lo cual incrementa la eficiencia y el rendimiento. Sin embargo, esta concurrencia introduce problemas de sincronización cuando dos o más entidades desean acceder simultáneamente a una misma sección crítica o recurso compartido. Sin mecanismos adecuados, surgen condiciones de carrera en las que el resultado de la ejecución depende del orden de acceso, así como situaciones de interbloqueo (*deadlock*) o inanición (*starvation*).

Edgser Dijkstra introdujo el concepto de semáforo en 1965 como una abstracción para resolver estos retos.



Un semáforo es un contador entero más un conjunto de operaciones atómicas que permiten a los procesos esperar (**operación P o wait**) y señalar (**operación V o signal**) la disponibilidad de recursos. Su simplicidad y eficacia lo convirtieron en un pilar tanto en sistemas de tiempo compartido como en entornos de tiempo real y sistemas distribuidos.

Alcance y organización de estos apuntes (“Pastel de estudio”)

Este “*Pastel de estudio*” abarca desde los fundamentos históricos y teóricos hasta implementaciones prácticas en C, incluyendo análisis comparativo y mejores prácticas. La estructura del documento de repaso es el siguiente:

Sección 2: Fundamentos teóricos de los semáforos.
Sección 3: Clasificación y tipos de semáforos.
Sección 4: Modelos de uso y patrones de diseño.
Sección 5: Ejemplos y casos prácticos.
Sección 6: Tablas explicativas.
Sección 7: Implementación en C.
Sección 8: Diseño avanzado y mejores prácticas.
Sección 9: Herramientas de depuración y verificación.
Sección 10: Conclusiones y perspectivas futuras.
Sección 11: Bibliografía.

Con este planteamiento, se pretende ofrecer una visión integral y rigurosa de los semáforos como técnica de sincronización en sistemas operativos y como material de estudio y repaso dentro del curso de Sistemas Operativos.

2. Fundamentos teóricos de los semáforos

2.1. Definición y concepto histórico (Dijkstra, 1965)

Un semáforo es un mecanismo de sincronización introducido por Edsger W. Dijkstra en 1965 para resolver los problemas de exclusión mutua y coordinación entre procesos concurrentes. Formalmente, un semáforo se define como un entero no negativo **S** junto con dos operaciones atómicas: $\boldsymbol{P}(\,)$ (proberen, probar o wait) y $\boldsymbol{V}(\,)$ (verhogen, incrementar o signal). Dijkstra lo propuso en su clásico artículo *Cooperating Sequential Processes* para modelar la disponibilidad de recursos y garantizar que solo un número controlado de procesos acceda a secciones críticas o recursos compartidos.

2.2. Operaciones básicas: P (wait) y V (signal)

- **P(S)**: si **S > 0**, decrementa **S** en 1 y continúa; si $\boldsymbol{S} == \boldsymbol{0}$, bloquea el proceso hasta que otro realice una **V(S)**.
- **V(S)**: incrementa **S** en 1; si había procesos bloqueados en **P(S)**, despierta uno de ellos. Ambas operaciones son atómicas, de modo que no pueden interrumpirse ni intercalarse con operaciones de otros procesos sobre el mismo semáforo.

2.3. Propiedades e invariantes de los semáforos

1. **No negatividad**: en ningún momento el valor de **S** puede ser negativo.
2. **Contador de recursos**: **S** representa la cantidad de instancias de un recurso disponible o el grado de concurrencia permitido.
3. **Atomicidad**: **P( )** y **V( )** se implementan de manera que son indivisibles, evitando condiciones de carrera sobre **S**.
4. **Bloqueo y despertar**: un proceso en espera no consume CPU hasta ser liberado por una **V( )**.
   Conservación global: al final de una ejecución,

   donde #V y #P son el número total de llamadas a cada operación.

2.4. Leyes formales de Dijkstra sobre semáforos

Dijkstra estableció un conjunto de axiomas y leyes de transformación para razonar sobre programas concurrentes usando semáforos:

1. **Ley de mutua exclusión**: garantiza que solo un proceso a la vez ejecute la sección crítica si .
2. **Conmutatividad**: operaciones sobre distintos semáforos **S1** y **S2** conmutan:
3. **Ley de cancelación**: una llamada **V(S)** seguida inmediatamente de **P(S)** deja el semáforo en el mismo estado:
4. **Invariante de acumulación**: el valor de **S** nunca desciende por debajo de cero, asegurando ausencia de carreras de conteo.

Estas bases teóricas permiten diseñar y verificar algoritmos de sincronización correctos y libres de *deadlocks* en sistemas concurrentes y paralelos.

3. Clasificación y Tipos de Semáforos

3.1. Semáforo binario (mutex semaphore)

Un semáforo binario es aquel cuyo contador únicamente puede tomar valores 0 o 1. Se utiliza principalmente para garantizar **exclusión mutua** en secciones críticas, comportándose de forma equivalente a un mutex.

Valor 1: recurso libre.

Valor 0: recurso ocupado.

Su operación P( ) bloquea si el semáforo está en 0, y V( ) despierta exactamente a un proceso bloqueado (si lo hay), restaurando el contador a 1.

3.2. Semáforo de conteo (counting semaphore)

El semáforo de conteo extiende el concepto binario permitiendo que el contador **S** tome cualquier valor entero no negativo. Es útil para controlar el acceso concurrente a un conjunto de **n** instancias de un recurso:

Inicialización: S = número de instancias disponibles.

P( ): decrementa S hasta 0; bloquea si S = 0.

V( ): incrementa S, despertando procesos bloqueados.

3.3. Semáforos con prioridad y semáforos de barrera

Semáforos con prioridad: extienden los semáforos de conteo incorporando colas de espera con **prioridad**. Permiten despertar primero a procesos de mayor prioridad, evitando inanición de tareas críticas.

Semáforos de barrera (barrier semaphore): sincronizan **n** procesos hasta que todos alcanzan un punto de encuentro. Internamente combinan conteos y desbloqueos masivos al alcanzar el último proceso.

3.4. Comparativa con otros mecanismos: mutexes, monitores y variables de condición

Mecanismo	Alcance principal	Ventajas	Limitaciones
Mutex	Exclusión mutua simple	Implementación nativa en SO, bajo overhead	No gestiona contadores múltiples
Monitor	Agrupación de sección crítica y condiciones	Encapsula datos y sincronización	Puede ser complejo de implementar
Variable de condición	Señalización basada en predicados	Flexible para patrones complejos	Requiere mutex externo
Semáforo	Exclusion y conteo de recursos	Simple, poderoso para múltiples patrones	Uso incorrecto puede causar deadlock

Esta clasificación y comparación permiten seleccionar el mecanismo de sincronización más adecuado según los requisitos de concurrencia, complejidad y nivel de control deseado.

4. Modelos de Uso y Patrones de Diseño

4.1. Exclusión mutua: control de acceso concurrente

El patrón de exclusión mutua asegura que solo un proceso o hilo pueda ejecutar una sección crítica en un momento dado. Implementado con un semáforo binario (S = 1):

```
// Inicialización
sem_t mutex;
sem_init(&mutex, 0, 1);

// Sección crítica
down(&mutex);   // P(mutex)
// operaciones críticas
up(&mutex);     // V(mutex)
```

Este patrón previene condiciones de carrera garantizando la integridad de datos compartidos.

4.2. Sincronización de procesos: pasos de fase y barreras

Pasos de fase (phases): sincronización en etapas discretas; cada fase se completa cuando todos los procesos alcanzan un semáforo de barrera.

Barreras: semáforos de barrera que bloquean hasta que un contador (número de procesos) llega a cero, momento en que todos avanzan:

```
// Barrera simple para N procesos
sem_t barrier;
int count = 0;

void arrive_and_wait() {
    sem_wait(&mutex);
    count++;
    if (count == N) {
        for (int i = 0; i < N; i++) sem_post(&barrier);
    }
    sem_post(&mutex);
    sem_wait(&barrier);
}
```

4.3. Productor–Consumidor: control de buffer limitado

Coordina productores y consumidores usando 3 semáforos:

mutex para exclusión al buffer.

empty conteo de espacios libres.

full conteo de ítems disponibles.

```
// Inicialización
sem_init(&mutex, 0, 1);
sem_init(&empty, 0, BUFFER_SIZE);
sem_init(&full, 0, 0);

void producer() {
    // producir ítem
    sem_wait(&empty);
    sem_wait(&mutex);
    buffer[in++] = item;
    sem_post(&mutex);
    sem_post(&full);
}

void consumer() {
    sem_wait(&full);
    sem_wait(&mutex);
    item = buffer[out++];
    sem_post(&mutex);
    sem_post(&empty);
    // consumir ítem
}
```

4.4. Lectores–Escritores: escalonamiento de acceso a datos compartidos

Permite múltiples lectores concurrentes o un único escritor, usando dos semáforos y un contador de lectores:

```c
sem_t rw_mutex, mutex;
int read_count = 0;

void reader() {
    sem_wait(&mutex);
    read_count++;
    if (read_count == 1) sem_wait(&rw_mutex);
    sem_post(&mutex);
    // leer datos
    sem_wait(&mutex);
    read_count--;
    if (read_count == 0) sem_post(&rw_mutex);
    sem_post(&mutex);
}

void writer() {
    sem_wait(&rw_mutex);
    // escribir datos
    sem_post(&rw_mutex);
}
```

4.5. Otros casos de uso: consumidores múltiples, pool de hilos, etc.

Consumidores múltiples: extender el patrón productor-consumidor para varios consumidores y productores con semáforos de conteo adecuados.

Pool de hilos (thread pool): usar un semáforo de conteo para limitar el número de hilos activos, combinando con exclusión mutua para gestionar la cola de tareas.

Estos patrones demuestran la flexibilidad de los semáforos para resolver problemas clásicos de concurrencia en sistemas operativos y aplicaciones paralelas.

5. Ejemplos y Casos Prácticos

5.1. Ejemplo en pseudocódigo del patrón Productor–Consumidor

```
// Recursos compartidos
define BUFFER_SIZE = N
buffer = array[N]
in = 0
out = 0

semaphore mutex = 1       // Exclusión mutua
semaphore empty = N       // Espacios libres
semaphore full = 0        // Ítems disponibles

function Producer():
    while true:
        item = produce_item()
        P(empty)              // Esperar espacio libre
        P(mutex)              // Entrar sección crítica
        buffer[in] = item
        in = (in + 1) mod N
        V(mutex)              // Salir sección crítica
        V(full)               // Señalar ítem disponible

function Consumer():
    while true:
        P(full)               // Esperar ítem disponible
        P(mutex)              // Entrar sección crítica
        item = buffer[out]
        out = (out + 1) mod N
        V(mutex)              // Salir sección crítica
        V(empty)              // Señalar espacio libre
        consume_item(item)
```

Este pseudocódigo muestra cómo coordinan productores y consumidores mediante tres semáforos: mutex, empty y full.

5.2. Ejemplo en pseudocódigo del patrón Lectores–Escritores.

```
// Variables compartidas
data = 0
read_count = 0

semaphore mutex = 1        // Protege read_count
semaphore rw_mutex = 1     // Sincroniza lectores/escritores

function Reader():
    while true:
        P(mutex)
        read_count = read_count + 1
        if read_count == 1:
            P(rw_mutex)       // Primer lector bloquea escritores
        V(mutex)

        // Leer datos compartidos
        read_data(data)

        P(mutex)
        read_count = read_count - 1
        if read_count == 0:
            V(rw_mutex)       // Último lector libera escritores
        V(mutex)

function Writer():
    while true:
        P(rw_mutex)          // Esperar acceso exclusivo
        write_data(data)
        V(rw_mutex)          // Liberar acceso
```

Este esquema permite que múltiples lectores accedan simultáneamente, pero garantiza exclusión mutua para escritores.

5.3. Análisis de un caso real: aplicación de semáforos en sistemas embebidos

En sistemas embebidos, como microcontroladores en aplicaciones automotrices o IoT, se emplean semáforos en RTOS (Real-Time Operating Systems) para coordinar tareas de sensor y comunicación:

Contexto: Un RTOS gestiona tareas periódicas de lectura de sensores (temperatura, presión) y tareas de transmisión de datos por CAN bus.

Implementación: Cada tarea de lectura genera datos en un búfer circular protegido por un semáforo binario (mutex). Una tarea de envío espera con un semáforo de conteo (data_ready) que indica cuántos bloques de datos están listos.

Flujo:

- Las tareas de sensor realizan P(mutex) antes de escribir en el búfer y V(mutex) al terminar.
- Tras escribir, hacen V(data_ready) para notificar a la tarea de transmisión.
- La tarea de transmisión ejecuta P(data_ready) antes de leer cada bloque.

Ventajas: Garantiza sincronización determinista, evita conflictos de acceso y mantiene el deadline de transmisión en sistemas en tiempo real.

Este ejemplo real ilustra cómo los semáforos proporcionan robustez y determinismo en entornos con restricciones de tiempo y recursos limitados

6. Tablas Explicativas

6.1. Tabla comparativa: semáforo vs mutex vs monitor

Mecanismo	Ámbito	Ventaja principal	Desventaja principal
Semáforo	Contador de recursos	Control de múltiples instancias de un recurso	Uso incorrecto puede causar deadlock
Mutex	Exclusión mutua binaria	Bajo overhead, integración directa en SO	No gestiona conteos de múltiples recursos
Monitor	Sección crítica + condiciones	Encapsula datos y sincronización juntos	Puede ser complejo de implementar correctamente

6.2. Tabla de llamadas al sistema / API y sus parámetros

Llamada	Biblioteca / API	Descripción	Parámetros clave
sem_init	POSIX sem_*	Inicializa un semáforo en memoria	sem_t *sem, int pshared, unsigned int value
sem_destroy	POSIX sem_*	Destruye un semáforo previamente inicializado	sem_t *sem
sem_wait	POSIX sem_*	Realiza P() (wait) bloqueante	sem_t *sem
sem_post	POSIX sem_*	Realiza V() (signal), incrementa y despierta	sem_t *sem
sem_open	POSIX sem_*	Abre o crea un semáforo nombrado	const char *name, int oflag, mode_t mode, unsigned int value
sem_close	POSIX sem_*	Cierra un semáforo nombrado	sem_t *sem
sem_unlink	POSIX sem_*	Elimina el nombre de un semáforo nombrado	const char *name

6.3. Tabla de estados y transiciones de un semáforo en ejecución

Estado	Descripción	Transición P()	Transición V()
Disponible	S > 0 (valor positivo) el recurso está libre	S = S - 1 → sigue disponible o pasa a cero	S = S + 1
Agotado	S = 0 y cola de espera vacía	bloquea proceso y lo añade a la	S = 1 → despierta un proceso bloqueado

		cola de espera	
Bloqueado	S = 0 y uno o más procesos en cola de espera	mantiene cola, proceso en espera	S = 1 → elimina primer proceso de cola y despierta

7. Implementación en Lenguaje C

7.1. Semáforos POSIX (API sem_*)

POSIX define una serie de funciones para trabajar con semáforos tanto anónimos (en memoria compartida) como nombrados.

7.1.1. sem_init(), sem_destroy()

- int sem_init(sem_t *sem, int pshared, unsigned int value);
    - **sem**: puntero a semáforo no nombrado.
    - **pshared**: 0 para procesos hilos de un mismo proceso, >0 para compartir entre procesos.
    - **value**: valor inicial del semáforo.
    - Retorna 0 en éxito o -1 y errno en caso de error.
- int sem_destroy(sem_t *sem);
    - **sem**: puntero a semáforo previamente inicializado.
    - Libera los recursos asociados.
    - Retorna 0 en éxito o -1 y errno en caso de error.

7.1.2. sem_wait(), sem_post()

- int sem_wait(sem_t *sem);
    - Bloquea hasta que sem->value > 0, luego decrementa el semáforo.
    - Retorna 0 en éxito o -1 y errno en caso de error.
- int sem_post(sem_t *sem);
    - Incrementa sem->value. Si había procesos bloqueados, despierta a uno.
    - Retorna 0 en éxito o -1 y errno en caso de error.

7.1.3. sem_open(), sem_close(), sem_unlink()

- sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
    - **name**: nombre del semáforo (debe empezar con '/').
    - **oflag**: O_CREAT, O_EXCL para control de creación.
    - **mode**: permisos tipo Unix.
    - **value**: valor inicial si se crea.
    - Retorna puntero al semáforo o SEM_FAILED y errno en error.
- int sem_close(sem_t *sem);
    - Desvincula un semáforo nombrado.
    - Retorna 0 en éxito o -1 en error.
- int sem_unlink(const char *name);
    - Elimina el nombre del semáforo del sistema.
    - Retorna 0 en éxito o -1 en error.

7.2. Ejemplos completos de código en C para patrones básicos.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0;
sem_t mutex, empty, full;

void *producer(void *arg) {
    int item;
    while (1) {
        item = rand() % 100;
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        printf("Produced %d\n", item);
        sem_post(&mutex);
        sem_post(&full);
        sleep(1);
    }
}
```

```c
void *consumer(void *arg) {
    int item;
    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("Consumed %d\n", item);
        sem_post(&mutex);
        sem_post(&empty);
        sleep(1);
    }
}

int main() {
    pthread_t prod, cons;
    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    pthread_create(&prod, NULL, producer, NULL)
    pthread_create(&cons, NULL, consumer, NULL)

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
    return 0;
}
```

7.3. Manejo de errores y retorno de llamadas
En producción, siempre comprobar el valor de retorno de cada llamada a semáforos y pthreads:

```c
if (sem_init(&mutex, 0, 1) == -1) {
    perror("sem_init");
    exit(EXIT_FAILURE);
}
```

8. Diseño Avanzado y Mejores Prácticas

8.1. Prevención y detección de deadlocks

Condiciones necesarias: exclusión mutua, retención y espera, no-preempción, espera circular.

Prevención: imponer un orden global en la adquisición de semáforos, usar semáforos jerárquicos, o detectar y evitar ciclos en el grafo de asignación.

Detección: mantener un grafo de asignación de recursos, buscar ciclos en tiempo de ejecución y desencadenar recuperación (abortos o rollbacks).

8.2. Estrategias contra la inanición (starvation)

Asignar prioridades: usar semáforos con prioridades, garantizando que procesos críticos no se queden esperando indefinidamente.

Política FIFO: despertar procesos en orden de llegada al semáforo.

Aging: incrementar dinámicamente la prioridad de procesos bloqueados cuanto más tiempo esperan.

8.3. Asignación de prioridades y orden FIFO

Semáforos FIFO: colas FIFO garantizan orden justo.

Semáforos con prioridad: colas múltiples basadas en niveles de prioridad.

Implementación: exponer APIs sem_prio_wait() y sem_prio_post() que admitan parámetros de prioridad.

8.4. Escalabilidad y rendimiento en sistemas multicore

Contención de semáforos: minimizar secciones críticas, usar bloqueos de grano fino.

Combinar con lock-free: mix semáforos y estructuras lock-free (colas MPMC).

Reducción de coste de wake-up: usar técnicas como wake-up grupos y batching.

Evaluación de rendimiento: medir latencia de P/V y throughput con benchmarking en múltiples cores.

Este conjunto de buenas prácticas ayuda a diseñar sistemas concurrentes robustos y eficientes en arquitecturas modernas.

9. Herramientas de Depuración y Verificación

9.1. Uso de Valgrind y Helgrind para detectar condiciones de carrera

- **Valgrind:** framework para detectar fugas de memoria y accesos inválidos.
- **Helgrind (Valgrind tool):** especializado en detectar condiciones de carrera en programas multihilo.

- Reporta accesos concurrentes a la misma dirección sin sincronización.
- Comando: valgrind --tool=helgrind ./mi_programa.

9.2. Modelado y simulación con herramientas formales (SPIN, TLA+)

SPIN: verificador de modelos para sistemas concurrentes. Usa PROMELA para modelar procesos y semáforos.

Detecta deadlocks, ciclos infinitos y violaciones de propiedades LTL.

TLA+: lenguaje de especificación para sistemas distribuidos. Permite definir semáforos y propiedades lógicas, y usar TLC para verificar invariantes.

9.3. Pruebas unitarias y de integración en entornos concurrentes

Frameworks: Google Test con extensiones para multihilo, CUnit con pthread mocks.

Estrategias: aislar secciones críticas, usar tées de escapes (mock semáforos) y tests deterministas fomentando control de scheduling.

Estas herramientas permiten detectar y corregir errores de sincronización y validar la corrección de algoritmos concurrentes.

10. Conclusiones y perspectivas futuras

10.1. Resumen de hallazgos y aportes clave

- Los semáforos, introducidos por Dijkstra en 1965, constituyen un mecanismo fundamental para la sincronización y exclusión mutua en sistemas operativos y aplicaciones concurrentes.
- Se verificaron las propiedades esenciales: atomicidad de las operaciones P/V, no negatividad y conservación global del contador.
- La clasificación mostró semáforos binarios para exclusión mutua, semáforos de conteo para múltiples recursos y variantes con prioridad o barreras, cada uno adaptado a patrones específicos.
- Ejemplos prácticos (productor-consumidor, lectores-escritores) y casos reales en sistemas embebidos demostraron su eficacia para coordinar tareas y asegurar la coherencia de datos.
- Buenas prácticas y diseños avanzados mitigan deadlocks e inanición, mientras que herramientas de verificación (Valgrind/Helgrind, SPIN, TLA+) facilitan la depuración.

10.2. Limitaciones del estudio

- El análisis se ha centrado en semáforos a nivel de sistema operativo clásico y POSIX; no se han explorado en profundidad implementaciones específicas de kernels o RTOS propietarias.
- La comparativa con mecanismos lock-free y transaccionales sólo se mencionó de forma general, sin un análisis de rendimiento cuantitativo exhaustivo.
- La sección de pruebas avanzadas requiere un mayor despliegue de escenarios reales y métricas de evaluación en arquitecturas multicore y distribuidas.

10.3. Líneas de investigación futura

- Semáforos distribuidos: extender el modelo a entornos sin memoria compartida, integrando algoritmos de consenso y tolerancia a fallos.
- GPU y aceleradores: adaptar semáforos para sincronización eficiente en arquitecturas masivamente paralelas, considerando latencia y jerarquías de memoria.
- RTOS y sistemas críticos: investigar variantes de semáforos en sistemas de tiempo real, garantizando propiedades de deadline y minimizando jitter.
- Mecanismos híbridos: combinar semáforos con técnicas lock-free y transaccionales para optimizar rendimiento en aplicaciones de alta concurrencia.

11. Problemas Clásicos de Concurrencia

11.1. Cena de Filósofos

El problema de la cena de filósofos, propuesto por Dijkstra en 1965, modela la sincronización de cinco filósofos sentados alrededor de una mesa circular. Entre cada par de filósofos hay un tenedor. Cada

filósofo alterna entre pensar y comer, pero para comer debe tomar simultáneamente los dos tenedores adyacentes.

- **Desafíos**:
  - Interbloqueo (Deadlock): si cada filósofo toma primero el tenedor de la izquierda y luego espera el de la derecha, todos pueden quedarse bloqueados.
  - Inanición (Starvation): incluso sin deadlock, algunos filósofos podrían nunca conseguir los dos tenedores si otros acaparan los recursos.
- **Soluciones comunes**:
  - Orden estricto: numerar los tenedores y obligar a los filósofos a tomarlos en orden creciente (primero el de menor número, luego el mayor).
  - Filósofo impar primero tenedor derecho: diseño simétrico que rompe la circularidad.
  - Semáforo global: permitir un máximo de cuatro filósofos simultáneos intentando comer.

Ejemplo de código en C (usando semáforos POSIX):

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

#define N 5
sem_t forks[N];
sem_t room;  // Limita filósofos en mesa
```

```c
void *philosopher(void *arg) {
    int id = *(int*)arg;
    while (1) {
        // Pensar
        printf("Filósofo %d está pensando\n", id);
        usleep(rand() % 1000);

        // Entrar al comedor
        sem_wait(&room);
        // Tomar tenedores
        sem_wait(&forks[id]);
        sem_wait(&forks[(id + 1) % N]);

        // Comer
        printf("Filósofo %d está comiendo\n", id);
        usleep(rand() % 1000);

        // Soltar tenedores
        sem_post(&forks[id]);
        sem_post(&forks[(id + 1) % N]);
        sem_post(&room);
    }
    return NULL;
}
int main() {
    pthread_t tid[N];
    int ids[N];

    sem_init(&room, 0, N-1);  // Máximo N-1 filósofos comiendo o intentando
    for (int i = 0; i < N; i++) sem_init(&forks[i], 0, 1);

    for (int i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&tid[i], NULL, philosopher, &ids[i]);
    }
    for (int i = 0; i < N; i++) pthread_join(tid[i], NULL);
    return 0;
}
```

Este enfoque evita deadlock al no permitir que los cinco filósofos entren simultáneamente.

11.2. Barbero Dormilón

El problema del barbero dormilón, formulado por Dijkstra en 1965, describe una barbería con una única silla de barbero y una sala de espera con capacidad para n clientes. Si no hay clientes, el barbero se duerme. Cuando un cliente llega y el barbero está dormido, lo despierta; si la sala de espera está llena, el cliente se va.

- Desafíos:
  - Coordinar correctamente la señalización entre clientes y barbero.
  - Evitar que clientes despercibidos no sean atendidos (inanición).
- Solución con semáforos: usar tres semáforos y un mutex:

- sem_t chairs (clientes esperando, inicializado a n).
- sem_t barber (barbero listo, inicializado a 0).
- sem_t customer (cliente listo, inicializado a 0).
- sem_t mutex (protejendo contador de sillas libres).

Ejemplo de código en C:

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

#define CHAIRS 3
sem_t chairs, barber, customer, mutex;

void *barber_thread(void *arg) {
    while (1) {
        sem_wait(&customer);   // Espera cliente
        sem_wait(&mutex);
        sem_post(&chairs);     // Libera silla de espera
        sem_post(&barber);     // Barbero listo para cortar
        sem_post(&mutex);
        // Cortar pelo
        printf("Barbero está cortando el pelo\n");
        usleep(rand() % 1000);
    }
    return NULL;
}
void *customer_thread(void *arg) {
    int id = *(int*)arg;
    while (1) {
        sem_wait(&mutex);
        if (sem_trywait(&chairs) == 0) {
            printf("Cliente %d está esperando\n", id);
            sem_post(&customer);
            sem_post(&mutex);
            sem_wait(&barber);  // Espera a que el barbero esté listo
            // Ser atendido
            printf("Cliente %d está siendo atendido\n", id);
            usleep(rand() % 1000);
        } else {
            sem_post(&mutex);
            printf("Cliente %d se va (sala llena)\n", id);
            usleep(rand() % 1000);
        }
    }
    return NULL;
}
```

```c
int main() {
    pthread_t b, customers[5];
    int ids[5];

    sem_init(&chairs, 0, CHAIRS);
    sem_init(&barber, 0, 0);
    sem_init(&customer, 0, 0);
    sem_init(&mutex, 0, 1);

    pthread_create(&b, NULL, barber_thread, NULL);
    for (int i = 0; i < 5; i++) {
        ids[i] = i;
        pthread_create(&customers[i], NULL, customer_thread, &ids[i]);
    }
    pthread_join(b, NULL);
    return 0;
}
```

Este código ilustra la coordinación mediante semáforos para evitar que más clientes esperen de los que caben y garantizar que el barbero atienda correctamente.

II. CONCURRENCIA

1. Introducción a la concurrencia

1.1. Definición y motivación

La concurrencia es la capacidad de un sistema para gestionar la ejecución de múltiples tareas (procesos o hilos) que progresan de manera solapada en el tiempo. No siempre implica paralelismo físico (multiprocesador), sino la interleaving de operaciones en un único núcleo o la ejecución simultánea en múltiples núcleos.

- **Motivación:**

Maximizar el uso del CPU: aprovechar ciclos ociosos durante I/O.

Responsividad: en interfaces gráficas y servidores, atender múltiples clientes simultáneamente.

SLA y rendimiento: en aplicaciones de alta demanda (servidores web, bases de datos).

1.2. Modelos de concurrencia: hilos vs procesos

Característica	Proceso	Hilo (thread)
Espacio de direc.	Aislado (memoria separada)	Compartido (mismo espacio de direcciones)
Peso	Pesado (overhead de cambio de contexto)	Ligero (cambio de contexto más rápido)
Creación	fork()	pthread_create()
Comunicación	IPC (pipes, sockets, shared memory)	Variables globales, semáforos, mutex
Fallos	Un proceso falla no afecta a otros	Un hilo falla puede colapsar todo el proceso

1.3. Necesidad de sincronización y ejemplos de aplicaciones reales

Cuando múltiples tareas acceden a recursos o datos compartidos, surge la necesidad de sincronización para garantizar:

- Coherencia de datos: evitar escrituras y lecturas inconsistentes.
- Exclusión mutua: proteger secciones críticas.
- Orden correcto de ejecución: mantener invariantes.

Ejemplos:

- Servidores web multihilo: cada hilo atiende una petición; comparten cachés y estructuras de datos.
- Bases de datos: transacciones concurrentes actualizan registros; se usan locks y semáforos para aislamiento.
- Sistemas embebidos: tareas de sensor/comunicación en RTOS; semáforos POSIX para coordinar acceso a periféricos.

Este planteamiento muestra por qué la concurrencia es esencial y cómo los modelos difieren, sentando las bases para técnicas de sincronización posteriores.

2. Violación de atomicidad

2.1. Concepto de operación atómica

Una operación atómica es aquella que se ejecuta de manera indivisible: completa sin que pueda interrumpirse ni mezclarse con otras. En sistemas concurrentes, garantiza que ningún otro hilo o proceso vea un estado intermedio ni modifique los operandos durante su ejecución.

2.2. Casos de violación: lectura-escritura concurrente

Cuando una secuencia de instrucciones que debería comportarse como una unidad se divide en varias suboperaciones (por ejemplo, cargar, modificar y almacenar un valor), la intercalación con otro hilo puede provocar resultados inesperados:

Secuencia no atómica	Suboperaciones	Posible interleaving
x = x + 1;	1) cargar x2) incrementar3) almacenar x	Hilo A carga x=0, Hilo B carga x=0, A almacena 1, B almacena 1 → pérdida de actualización

2.3. Ejemplos de código en C y consecuencias

A continuación, un ejemplo de incremento de un contador compartido sin sincronización:

```c
#include <stdio.h>
#include <pthread.h>

volatile int counter = 0;

void* worker(void* arg) {
    for (int i = 0; i < 1000000; i++) {
        // Operación no atómica: load, inc, store
        counter++;
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, worker, NULL);
    pthread_create(&t2, NULL, worker, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Valor final de counter: %d (esperado 2000000)\n", counter);
    return 0;
}
```

- Resultado típico: counter queda por debajo de 2000000 debido a condiciones de carrera.
- Consecuencia: pérdida de incrementos, estados inconsistentes y errores de cálculo.

Solución: usar operaciones atómicas proporcionadas por el compilador o semáforos/mutex:

```c
#include <stdatomic.h>

atomic_int counter = 0;

// En worker():
for (int i = 0; i < 1000000; i++) {
    atomic_fetch_add(&counter, 1);
}
```

Con atomic_fetch_add, la operación se ejecuta en una única instrucción atómica, evitando la intercalación de hilos y garantizando el valor correcto.

Este análisis ilustra cómo la violación de atomicidad causa errores impredecibles en entornos concurrentes y la importancia de utilizar primitivas atómicas o mecanismos de sincronización adecuados.

3. Violación de orden (reordenamientos)

3.1. Visión y coherencia de memoria

La consistencia de memoria define el orden en que las operaciones de lectura/escritura son percibidas por distintos hilos. El modelo más intuitivo es la consistencia secuencial, donde el resultado de la ejecución concurrente equivale a alguna intercalación de las órdenes de cada hilo, respetando el orden de programa dentro de cada uno.

Modelo	Garantía principal
Consistencia secuencial	Todas las operaciones aparecen en un único orden global consistente con cada hilo
Consistencia débil	Permite reordenamientos visibles por otros hilos salvo barreras explícitas
Modelo C11/C++11	Ofrece distintos niveles (relaxed, acquire/release, sequentially consistent)

3.2. Compilador y CPU: optimizaciones que reordenan instrucciones

Tanto el compilador como la arquitectura CPU pueden alterar el orden de instrucciones para mejorar rendimiento:

- Reordenamiento del compilador: agrupar accesos a memoria o registros, eliminar redundancias.
- Reordenamiento a nivel de CPU: pipelines, ejecución fuera de orden y buffers de escritura.
- Impacto: sin barreras de memoria o primitivas atómicas, un hilo puede leer valores “antiguos” o ver operaciones de otros hilos en orden distinto al de programa.

3.3. Ejemplos de código y anomalías de orden

```c
#include <stdio.h>
#include <pthread.h>
#include <stdatomic.h>

int x = 0, y = 0;
int r1 = 0, r2 = 0;

void* thread1(void* arg) {
    x = 1;          // A
    r1 = y;         // B
    return NULL;
}

void* thread2(void* arg) {
    y = 1;          // C
    r2 = x;         // D
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("r1=%d, r2=%d\n", r1, r2);
    return 0;
}
```

- Situación esperada: al menos uno de r1 o r2 debe ser 1.
- Anomalía: es posible observar $r1 == 0 \,\&\&\, r2 == 0$ debido a reordenamientos (A ↔ B, C ↔ D) en ausencia de barreras.

Solución: usar barreras de memoria (atomic_thread_fence(memory_order_seq_cst)) o accesos atómicos con ordenamiento apropiado.

Este subtema muestra cómo la violación del orden de programa afecta a la coherencia y por qué las primitivas de memoria son esenciales en concurrencia.

4. Condiciones de Carrera

4.1. Definición y ejemplos clásicos

Una condición de carrera ocurre cuando el comportamiento de un programa concurrente depende del orden o interleaving de operaciones sobre datos compartidos, provocando resultados no deterministas.

> Ejemplo clásico: Incremento de un contador global sin sincronización (counter++) produce pérdida de actualizaciones.
>
> Otro ejemplo: Iniciar sesión en un sistema escribiendo un archivo de bitácora, donde dos hilos escriben simultáneamente y mezclan líneas.

4.2. Detección en tiempo de ejecución y análisis estático

- Detección dinámica: instrumentar el ejecutable y monitorizar accesos concurrentes.
- Análisis estático: revisar el código para identificar accesos no protegidos a variables compartidas.

4.3. Herramientas de análisis

Herramienta	Tipo	Uso principal
Valgrind Helgrind	Dinámica / Instrumental	valgrind --tool=helgrind ./programa detecta condiciones de carrera
ThreadSanitizer (TSan)	Dinámica / Clang/GCC	Compilar con -fsanitize=thread para detectar data races
Coverity / Clang Static Analyzer	Estática	Analiza código fuente para patrones riesgosos de concurrencia

Estas técnicas y herramientas permiten identificar condiciones de carrera antes de que provoquen fallos en producción.

5. Interbloqueos (deadlock)

5.1. Definición y consecuencias

Un interbloqueo o **deadlock** ocurre cuando un conjunto de procesos queda permanentemente bloqueado, cada uno esperando un recurso que otro posee. Esto impide que cualquier proceso avance, causando pérdida de disponibilidad.

5.2. Las 4 condiciones necesarias para deadlock

1. Exclusión mutua: cada recurso está asignado a lo sumo a un proceso.
2. Retención y espera: un proceso retiene recursos mientras espera otros.
3. No preempción: los recursos solo se liberan voluntariamente por el proceso.
4. Espera circular: existe un ciclo de procesos donde cada uno espera un recurso del siguiente.

5.3. Ejemplo en C: demostración de deadlock

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

pthread_mutex_t m1, m2;

void* f1(void* arg) {
    pthread_mutex_lock(&m1);
    sleep(1);  // Asegura que f2 adquiera m2
    pthread_mutex_lock(&m2);
    printf("Hilo 1 en sección crítica\n");
    pthread_mutex_unlock(&m2);
    pthread_mutex_unlock(&m1);
    return NULL;
}

void* f2(void* arg) {
    pthread_mutex_lock(&m2);
    sleep(1);
    pthread_mutex_lock(&m1);
    printf("Hilo 2 en sección crítica\n");
    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&m1, NULL);
    pthread_mutex_init(&m2, NULL);
    pthread_create(&t1, NULL, f1, NULL);
    pthread_create(&t2, NULL, f2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

En este código, f1 bloquea m1 y luego intenta m2, mientras f2 bloquea primero m2 y luego m1, creando una espera circular y deadlock.

5.4. Detección de deadlocks

- Grafo de asignación de recursos: nodos de procesos y recursos; aristas de asignación y petición.
- Algoritmo de detección: buscar ciclos en el grafo para identificar interbloqueos en tiempo de ejecución.

Este subtema sienta las bases para las soluciones de **deadlock** posteriores.

6. Soluciones y Prevención de Deadlocks

6.1. Prevención: anulación de una condición necesaria

La prevención de deadlock consiste en eliminar al menos una de las cuatro condiciones necesarias:

- Eliminar retención y espera: requerir que un proceso solicite todos los recursos de una vez, bloqueando hasta recibirlos todos.
- Permitir preempción: si un proceso no puede obtener un recurso adicional, libera los que ya tiene.
- Evitar espera circular: imponer un orden total a los recursos y obligar a los procesos a solicitar recursos en orden creciente.

Ejemplo en C (orden total de mutexes)

```
// Dos mutexes m1, m2 con orden global: m1 < m2
directives:
// Siempre adquirir m1 antes de m2
typedef enum {M1, M2} Resource;

pthread_mutex_t m1, m2;

void access_resources(Resource first, Resource second) {
    pthread_mutex_t *r1 = (first == M1 ? &m1 : &m2);
    pthread_mutex_t *r2 = (second == M1 ? &m1 : &m2);
    pthread_mutex_lock(r1);
    pthread_mutex_lock(r2);
    // Sección crítica
    pthread_mutex_unlock(r2);
    pthread_mutex_unlock(r1);
}
```

6.2. Evitación: algoritmo del banquero de Dijkstra

El algoritmo del banquero evita deadlock analizando si la asignación solicitada mantiene al sistema en un estado seguro. Mantiene tablas de Max[i], Allocation[i], Need[i] = Max[i] - Allocation[i] y Available.

```
// Pseudocódigo simplificado del chequeo de seguridad

typedef int Matrix[N][M];

int Available[M];

int Max[N][M];

int Allocation[N][M];

int Need[N][M];

bool is_safe() {

  bool Finish[N] = {false};

  int Work[M]; memcpy(Work, Available, sizeof(Work));

  while (true) {

    bool found = false;

    for (i = 0; i < N; i++) if (!Finish[i]) {

      bool can_allocate = true;

      for (j = 0; j < M; j++) if (Need[i][j] > Work[j]) { can_allocate = false; break;
}

      if (can_allocate) {

        for (j = 0; j < M; j++) Work[j] += Allocation[i][j];

        Finish[i] = true; found = true;

      }

    }

    if (!found) break;

  }

  for (i = 0; i < N; i++) if (!Finish[i]) return false;

  return true;

}
```

6.3. Detección y recuperación: abortos y retrocesos

- Detección: construir un grafo de asignación de recursos y detectar ciclos periódicamente.

- Recuperación: abortar procesos de forma selectiva o forzar liberación de recursos (rollback) hasta romper el ciclo.

6.4. Ejemplos de código en C para prevención y evasión

- Prevención (orden de recursos): ver ejemplo en 6.1.
- Evitación (banquero): implementar llamadas a is_safe() antes de conceder recursos:

```
bool request_resources(int pid, int request[]) {
    // Validar request <= Need[pid]
    for (j = 0; j < M; j++) if (request[j] > Need[pid][j]) return false;
    // Simular asignación
    for (j = 0; j < M; j++) {
        Available[j] -= request[j];
        Allocation[pid][j] += request[j];
        Need[pid][j] -= request[j];
    }
    if (!is_safe()) {
        // Deshacer simulación
        for (j = 0; j < M; j++) {
            Available[j] += request[j];
            Allocation[pid][j] -= request[j];
            Need[pid][j] += request[j];
        }
        return false;
    }
    return true;
}
```

7. Estrellación (Starvation) y Prioridad Invertida

7.1. Concepto de inanición (starvation)

La inanición ocurre cuando un proceso o hilo de baja prioridad nunca recibe acceso a un recurso compartido porque continuamente es adelantado por otros de mayor prioridad. Aunque no exista deadlock, el proceso «pobrecito» queda esperando indefinidamente.

7.2. Prioridad invertida: definición y mitigación (Priority Inheritance)

La prioridad invertida es un fenómeno particular donde un hilo de alta prioridad queda bloqueado esperando un recurso que posee un hilo de baja prioridad. Mientras tanto, un hilo de prioridad media puede estar ejecutándose, impidiendo al hilo de baja prioridad liberar el recurso, lo que retrasa al de alta prioridad.

- Priority Inheritance: protocolo que eleva temporalmente la prioridad del hilo que posee el recurso al nivel del hilo de mayor prioridad que lo está esperando. De esta forma, se le permite terminar rápidamente y liberar el recurso.

7.3. Ejemplos y soluciones en código C

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

pthread_mutex_t mutex;

void* low_priority(void* arg) {
    // Simula trabajo previo
    sleep(1);
    pthread_mutex_lock(&mutex);
    printf("[Low] bloqueado, enter sección crítica\n");
    sleep(3);
    printf("[Low] liberando recurso\n");
    pthread_mutex_unlock(&mutex);
    return NULL;
}

void* high_priority(void* arg) {
    sleep(2);
    printf("[High] intentando adquirir recurso\n");
    pthread_mutex_lock(&mutex);
    printf("[High] adquirido, entrar sección crítica\n");
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main() {
    pthread_t th_low, th_high, th_medium;
    pthread_mutexattr_t attr;

    // Configurar mutex con protocolo priority inheritance
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);
    pthread_mutex_init(&mutex, &attr);

    // Hilo de prioridad media consume CPU
    pthread_create(&th_medium, NULL, (void*)sleep, (void*)1);
    pthread_create(&th_low, NULL, low_priority, NULL);
    pthread_create(&th_high, NULL, high_priority, NULL);

    pthread_join(th_low, NULL);
    pthread_join(th_high, NULL);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

En este ejemplo, al habilitar PTHREAD_PRIO_INHERIT, el hilo de baja prioridad hereda la prioridad alta, evitando que el hilo de prioridad media lo atrase y resolviendo la prioridad invertida.

8. Mecanismos de sincronización

En sistemas concurrentes se emplean diversos mecanismos para coordinar el acceso a recursos compartidos y garantizar la correcta ejecución de secciones críticas. A continuación, se describen los principales.

8.1. Mutex y Spinlocks

Mutex (mutual exclusion lock): bloqueo que pone a dormir al hilo si no puede adquirirlo.

```
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
pthread_mutex_lock(&m);
// sección crítica
pthread_mutex_unlock(&m);
pthread_mutex_destroy(&m);
```

Spinlock: bucle activo que espera liberación; útil en esperas muy cortas o en entornos de baja latencia.

```
pthread_spinlock_t s;
pthread_spin_init(&s, PTHREAD_PROCESS_PRIVATE);
pthread_spin_lock(&s);
// sección crítica
pthread_spin_unlock(&s);
pthread_spin_destroy(&s);
```

8.2. Semáforos (binarios y de conteo)

Binario (mutex ligero) y conteo (control de n recursos).

```
sem_t sem;
sem_init(&sem, 0, 1); // binario
sem_wait(&sem);
// crítica
sem_post(&sem);
sem_destroy(&sem);
```

8.3. Monitores e variables de condición

- Monitor: encapsula mutex + variables de condición.
- Condición: permite esperar/despertar según predicado.

```
pthread_mutex_t m;
pthread_cond_t cv;
// esperar
pthread_mutex_lock(&m);
while (!predicate) pthread_cond_wait(&cv, &m);
pthread_mutex_unlock(&m);
// señalizar
pthread_mutex_lock(&m);
// actualizar estado
pthread_cond_signal(&cv);
pthread_mutex_unlock(&m);
```

```
pthread_barrier_t b;
pthread_barrier_init(&b, NULL, N);
// cada hilo
pthread_barrier_wait(&b);
```

8.4. Barreras y Latches

- Barrera: sincroniza n hilos en un punto.
- Latch (conteo descendente): similar a semáforo inverso.

8.5. Pipes, Colas y Paso de Mensajes

Pipes Unix: comunicación unidireccional.

Queues: colas en memoria compartida o colas de mensajes.

// ejemplo simplificado: cola protegida por mutex y semáforos

8.6. Comparativa de rendimiento y escalabilidad

Mecanismo	Latencia de acquire	Overhead CPU	Escalabilidad multicore	Notas
Mutex	Media	Media	Buena	sleep-wake cost
Spinlock	Baja (cortas esperas)	Alta	Limitada	alto consumo CPU
Semáforo	Alta (sleep)	Media	Variable	dependiente SO
Condición	Media	Media	Buena	ideal para eventos
Barrera	Alta (espera colectiva)	Media	Buena	sin paso de datos
Pipe/Queue	Alta (context switch)	Alta	Depende de implementación	I/O-bound

9. Patrones de diseño concurrente

Los patrones de diseño concurrente ofrecen soluciones reutilizables a problemas clásicos de sincronización y comunicación entre hilos o procesos. A continuación, se describen los más relevantes.

9.1. Productor–Consumidor

Coordina productores que generan datos y consumidores que los procesan mediante un buffer limitado.

- **Primitivas**: 3 semáforos (mutex, empty, full).
- **Pseudocódigo**:

```
semaphore mutex = 1
semaphore empty = N
semaphore full = 0

Producer():
  while true:
    item = produce()
    P(empty); P(mutex)
    buffer[in++] = item
    V(mutex); V(full)

Consumer():
  while true:
    P(full); P(mutex)
    item = buffer[out++]
    V(mutex); V(empty)
    consume(item)
```

Ejemplo C: // Véase sección 7.2: implementación completa con pthreads y semáforos

9.2. Lectores–Escritores

Permite múltiples lectores concurrentes o un único escritor.

- Primitivas: 2 semáforos (mutex, rw_mutex) y contador read_count.
- Pseudocódigo:

```
Reader():
  P(mutex)
  read_count++
  if read_count == 1: P(rw_mutex)
  V(mutex)
  // lectura
  P(mutex)
  read_count--
  if read_count == 0: V(rw_mutex)
  V(mutex)

Writer():
  P(rw_mutex)
  // escritura
  V(rw_mutex)
```

Comparativa de cobertura: tabla resumida: | Patrón | Lectores concurrentes | Escritor exclusivo | Bloqueo mutuo | |---|---|---|---| | Sin prioridad | Sí | Sí | Writers bloquean lectores | | Con prioridad | Configurable | Configurable | Se puede dar preferencia |

9.3. Cena de Filósofos y Barbero Dormilón

- **Cena de Filósofos:** semáforos por tenedor + semáforo de sala para evitar deadlock.
- **Barbero Dormilón**: semáforos chairs, customer, barber, mutex para gestionar sala de espera y señalización.

Patrón	Recursos	Problema principal	Solución semáforos
Filósofos	Tenedores (5)	Deadlock e inanición	Limitar filósofos (N-1) o numerar tenedores
Barbero Dormilón	Sillas, barbero	Coordinación cliente-barb	Semáforos de conteo y señalización mutua

9.4. Actor Model y Cilk

- **Actor Model**: cada actor es un proceso ligero que solo comunica por envío de mensajes inmutables. No hay memoria compartida.

actor A: on receive(msg): // procesar msg send(B, response)

```
- **Cilk (Cilk Plus)**: extensión de C para paralelismo estructurado.
  ```c
#include <cilk/cilk.h>
void fib(int n) {
  if (n < 2) return n;
  int x = cilk_spawn fib(n-1);
  int y = fib(n-2);
  cilk_sync;
  return x + y;
}
```

Estos patrones ayudan a diseñar sistemas concurrentes robustos y fáciles de razonar.

10. Modelo de memoria y consistencia

10.1. Arquitectura Von Neumann y modelos de memoria

La arquitectura Von Neumann define un único espacio de direcciones para instrucciones y datos, conectados por un bus común. En sistemas modernos con múltiples núcleos y cachés, los modelos de memoria especifican cómo y cuándo las escrituras de un hilo son visibles a otros:

- UMA (Uniform Memory Access): misma latencia de acceso para todos los núcleos.
- NUMA (Non-Uniform Memory Access): latencias variables según la cercanía física de la memoria.
- Coherencia de caché: protocolos (MESI, MOESI) mantienen copias consistentes en cachés de cada núcleo.
```

10.2. Consistencia secuencial vs débil

Modelo	Descripción	Ventajas	Desventajas
Secuencial	Cada operación de lectura/escritura aparece en orden global único, respetando programa por hilo.	Intuitivo, fácil de razonar	Poco rendimiento en HW moderno
Débil (weak)	Permite reordenamientos y visibilidad diferida salvo barreras explícitas.	Alto rendimiento y paralelismo	Difícil de razonar, requiere barreras

- Release consistency: las escrituras se hacen visibles a otros tras una operación de release.
- Acquire consistency: lecturas posteriores a una operación acquire ven todos los efectos previos.

10.3. Barreras de memoria y operaciones atómicas en C11/C++11

C11/C++11 introducen primitivas atómicas con órdenes de memoria:

- atomic_thread_fence(memory_order_seq_cst): barrera global secuencial.
- atomic_thread_fence(memory_order_acq_rel): acquire on load, release on store.
- Operaciones atómicas: atomic_load_explicit, atomic_store_explicit, atomic_fetch_add_explicit con memory orders:

```c
#include <stdatomic.h>
atomic_int x = 0;

// Escritor
void writer() {
    atomic_store_explicit(&x, 42, memory_order_release);
}

// Lector
void reader() {
    int v = atomic_load_explicit(&x, memory_order_acquire);
    // tras acquire, ve todas las escrituras previas
}

// Barrera secuencial
atomic_thread_fence(memory_order_seq_cst);
```

Estas barreras y órdenes permiten construir algoritmos concurrentes correctos en arquitecturas con reordenamientos y caches múltiples, garantizando visibilidad y orden cuando es necesario.

11. Herramientas y metodologías de prueba

Para garantizar la fiabilidad y corrección de sistemas concurrentes, se emplean diversas metodologías y herramientas específicas:

11.1. Pruebas unitarias e integración en entornos multihilo

- Frameworks:
  - Google Test con extensiones para hilos (std::thread) y fixtures.
  - CUnit o Check combinado con mocks de pthread o semáforos.
- Estrategia:
  - Aislar funciones críticas usando mocks de primitivas de sincronización.
  - Introducir barreras o barrier()s para coordinar puntos de sincronización en el test.

11.2. Testing exploratorio y error guessing

- Testing exploratorio: diseñar y ejecutar pruebas ad hoc, adaptando los escenarios sobre la marcha.
- Error guessing: basarse en la experiencia para crear casos propensos a condiciones de carrera o deadlock.
- Ejemplo:
  - Forzar delays (usleep) en secciones clave para alterar el orden de ejecución.

11.3. Simulación y model checking (SPIN, TLA+)

Herramienta	Lenguaje/Modelo	Uso principal
SPIN	PROMELA	Modelado de protocolos y detección de deadlocks
TLA+	TLA+ + TLC	Especificación formal de algoritmos y verificación LTL

Ejemplo SPIN:

```
semaphore mutex = 1;
proctype P() {
  do
  :: mutex > 0 -> mutex--;
     /* sección crítica */
     mutex++;
  od
}
init { run P(); run P(); }
```

11.4. Benchmarking y profiling de aplicaciones concurrentes

- Herramientas: perf, gprof, Intel VTune para medir latencia de lock/unlock, contención.
- Métricas clave:
  - Throughput (operaciones/s)
  - Latencia de adquisición de bloqueo (µs)
  - Tiempo de espera promedio por hilo
- Metodología:
  - Diseñar benchmarks variando número de hilos y tamaño de sección crítica.
  - Graficar escalabilidad: speedup vs núcleos.

Esta sección aborda cómo validar y medir el comportamiento de sistemas concurrentes, desde pruebas básicas hasta análisis formal y métricas de rendimiento.

12. Aplicaciones y Casos de Estudio

12.1. Sistemas embebidos y RTOS

- Contexto: microcontroladores y microprocesadores en automoción, IoT, robótica.
- Sincronización: tareas de sensor, actuador y comunicación coordinadas con semáforos y mutex POSIX.
- Ejemplo en C (FreeRTOS):

```
// Task1: Lectura de sensor
void vTaskSensor(void *pvParameters) {
    for (;;) {
        xSemaphoreTake(xMutex, portMAX_DELAY);
        readSensor();
        xSemaphoreGive(xMutex);
        vTaskDelay(pdMS_TO_TICKS(10));
    }
}
// Task2: transmisión CAN
void vTaskCAN(void *pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xDataReady, portMAX_DELAY) == pdTRUE) {
            transmitCAN();
        }
    }
}
```

12.2. Servidores web y bases de datos multihilo

- Modelos: thread-per-request (Apache), thread pool (Nginx).
- Sincronización: caché compartido, pools de conexiones a base de datos.
- Ejemplo en C (pseudocódigo):

```
pthread_t worker;
while (accept(server_fd, ...)) {
    pthread_create(&worker, NULL, handle_request, (void*)&client_fd);
}
```

- Uso de pthread_mutex_t cache_mutex para proteger estructura de caché.

12.3. GPU y aceleradores de hardware

- Contexto: CUDA, OpenCL con miles de hilos ligeros.
- Sincronización: barreras de bloque (__syncthreads()), memoria atómica global.
- Ejemplo CUDA:

```
__global__ void vectorAdd(int *A, int *B, int *C) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    C[idx] = A[idx] + B[idx];
    __syncthreads();
}
```

12.4. Cloud y microservicios

- Modelo: procesos ligeros comunicados por REST/gRPC en contenedores.
- Sincronización: colas de mensajes (RabbitMQ, Kafka), locks distribuidos (Redis RedLock).
- Caso de estudio: escalado de microservicio con Redis para asegurar solo una instancia procesando un trabajo cron.

```python
import redis
lock = redis.Redis().lock('cron_lock', timeout=60)
if lock.acquire(blocking=False):
    run_cron_job()
    lock.release()
```

13. Conclusiones y perspectivas

13.1. Resumen de retos y soluciones en concurrencia

La concurrencia es fundamental para maximizar el rendimiento y la responsividad de los sistemas modernos. Sin embargo, introduce problemas complejos como condiciones de carrera, violaciones de atomicidad y orden, y deadlocks. Para abordarlos, se disponen de mecanismos de sincronización (mutexes, semáforos, monitores), técnicas de prevención y evitación de interbloqueos (algoritmo del banquero), y protocolos de prioridad (priority inheritance).

13.2. Tendencias: programación reactiva, actor model, lock-free

- Programación reactiva: flujos de datos y propagación de cambios para gestionar eventos asincrónicos.
- Actor model: elimina la memoria compartida, usa solo mensajes inmutables, facilitando el escalado en distribuidos.
- Lock-free y wait-free: estructuras y algoritmos sin bloqueos que minimizan latencias y evitán deadlocks.

13.3. Líneas de investigación futura

- Concurrencia en sistemas heterogéneos: CPU, GPU y aceleradores conjuntando paradigmas y coherencia.
- Semánticas de memoria en lenguajes de alto nivel: facilitar garantías de orden y visibilidad al programador.
- Algoritmos de sincronización distribuida: integrar semáforos y locks en entornos sin memoria compartida, con tolerancia a fallos.
- Verificación formal escalable: aplicar model checking y técnicas basadas en SMT para sistemas con miles de hilos.