Resumen

Hilos

- Los **hilos (threads)** son unidades básicas de ejecución dentro de un proceso.
- Permiten la ejecución concurrente al compartir el mismo espacio de direccionamiento, pero cada hilo tiene su propio **program counter**, registros y **stack**.

2. Concurrencia y Paralelismo

- **Concurrencia:**
  - Percepción de múltiples tareas ejecutándose al mismo tiempo en una CPU mediante multiprogramación.
  - Ejemplo: Un cajero alterna entre dos clientes.
- **Paralelismo:**
  - Ejecución real de múltiples tareas simultáneamente en varias CPUs.
  - Ejemplo: Dos cajeros atendiendo clientes al mismo tiempo.

3. Regiones de Memoria en Hilos

- Los hilos comparten las regiones de **código** y **heap**, pero cada hilo tiene su propio stack.
- Esto facilita la ejecución independiente de cada hilo sin interferir en las variables locales.

4. Problemas en Programación Concurrente

- **Condiciones de Carrera (Race Conditions):**
  - Ocurren cuando varios hilos acceden simultáneamente a una variable compartida sin sincronización.
  - Resultados inconsistentes y no deterministas.
  - Ejemplo: Incrementar una variable compartida (counter++) puede dar resultados incorrectos debido a interrupciones del sistema operativo.
- **Sección Crítica:**
  - Porción del código donde ocurre acceso a variables compartidas.
  - Necesita protección para garantizar que solo un hilo acceda a esta sección a la vez.
  - **Solución:** Exclusión mutua mediante mecanismos como:
    - **Mutexes:** Bloquean el acceso concurrente.
    - **Atomicidad:** Ejecutar instrucciones críticas sin interrupciones.

5. API pthread para Manejo de Hilos

- **pthread** es una librería estándar en sistemas POSIX (Linux, macOS) para manejar hilos.
- **Funciones principales:**
  - pthread_create: Crea hilos, asigna una función y pasa parámetros.
  - pthread_join: Espera la finalización de un hilo.
  - Los argumentos se pasan como punteros genéricos (void *) para mayor flexibilidad.

Ejemplo:

```
pthread_t thread_id;
pthread_create(&thread_id, NULL, my_function, (void *)&arguments);
pthread_join(thread_id, NULL);
```

6. Ventajas del Uso de Hilos

- **Eficiencia:** Menor sobrecarga que crear nuevos procesos.
- **Paralelismo:** Mejora el rendimiento al ejecutar tareas concurrentemente.
- **Escalabilidad:** Compatible con arquitecturas multi-núcleo modernas.

7. Casos Prácticos

1. **Procesadores de texto:**
   - Un hilo recibe entrada de teclado, otro verifica ortografía en tiempo real.
2. **Conexiones a Internet:**
   - Un hilo maneja la conexión, otro mantiene activa la interfaz gráfica.
3. **Venta de Tiquetes:**
   - Sin sincronización, múltiples hilos pueden generar problemas como **overbooking** al vender más tiquetes de los disponibles.

8. Soluciones a Problemas Concurrentes

1. **Identificar Secciones Críticas:**
   - Detectar dónde se accede a variables compartidas.
2. **Implementar Exclusión Mutua:**
   - Usar herramientas como mutexes o semáforos.
3. **Sincronización de Acceso:**
   - Proteger las operaciones críticas para evitar inconsistencias.

Conclusión

La programación concurrente y paralela, utilizando hilos, es esencial para aprovechar al máximo los sistemas modernos. Sin embargo, plantea desafíos como las condiciones de carrera y la necesidad de sincronización. Al identificar y proteger las secciones críticas mediante herramientas como **pthread**, los desarrolladores pueden garantizar aplicaciones más eficientes, confiables y escalables.

En sistemas concurrentes, los **locks** son mecanismos fundamentales para garantizar **exclusión mutua** y prevenir condiciones de carrera al acceder a recursos compartidos.

1. Problemas iniciales y técnicas simples

- **Regiones críticas:** Sin mecanismos de sincronización, múltiples hilos accediendo a una variable compartida pueden producir resultados inconsistentes debido a **condiciones de carrera**.
- **Deshabilitar interrupciones:**
  - Una de las primeras soluciones, deshabilita las interrupciones durante la región crítica para evitar cambios de contexto.
  - Limitaciones:
    - Ineficiente en sistemas multiprocesador.
    - Requiere confianza en las aplicaciones (un bucle infinito podría bloquear el sistema).

2. Soluciones

- **Uso de variables bandera (flags):**
  - Implementan locks verificando una variable que indica el estado de la región crítica.
  - Problemas:
    - Condiciones de carrera al leer y escribir el flag.
    - Alta ineficiencia en sistemas multiprocesador.
- **Técnicas avanzadas (Peterson, Dekker):**
  - Introducen turnos y reglas para garantizar exclusión mutua, pero no son prácticas en hardware moderno.

3. Avances con soporte de hardware

- **Instrucciones atómicas:**
  - **Test and Set:** Combina lectura y escritura en una única operación no interrumpible.
    - Solución efectiva para eliminar condiciones de carrera sobre el flag.
    - Introduce **espera activa**, que consume CPU mientras los hilos esperan.
  - **Compare and Swap (CAS):**
    - Más flexible, permite verificar y actualizar valores de memoria simultáneamente.
  - **Load Linked/Store Conditional (LL/SC):**
    - Garantiza atomicidad mediante operaciones combinadas de lectura y escritura.
- **Ticket locks:**
  - Implementan equidad asignando turnos a los hilos.
  - Usan **Fetch and Add** para distribuir turnos y asegurar que los hilos accedan en orden FIFO.

4. Optimización con espera activa y bloqueo

- **Problemas con espera activa:**
  - Consume CPU innecesariamente, especialmente en sistemas monoprocesador.
- **Uso de colas:**
  - Introduce funciones como park (bloqueo) y unpark (desbloqueo) para evitar el consumo de CPU durante la espera.
  - Limitaciones:
    - Las señales pueden perderse si los hilos intentan bloquearse mientras otros liberan el lock.
  - **Futex en Linux:**
    - Implementa mecanismos eficientes para gestionar hilos bloqueados y despertarlos cuando sea necesario.

5. Locks de dos fases

- Combina espera activa breve con bloqueo en hilos.
- **Fase 1:** Espera activa durante un período corto para verificar si el lock se libera rápidamente.
- **Fase 2:** Si la espera se prolonga, el hilo entra en estado bloqueado para liberar CPU.
- **Ventajas:**
  - Ideal en sistemas multiprocesador donde múltiples núcleos pueden procesar tareas en paralelo.
  - Reduce el costo de cambio de contexto al equilibrar espera activa y bloqueo.

Conclusión

La evolución de los locks refleja el progreso en la gestión de concurrencia en sistemas operativos, desde soluciones simples basadas en software hasta técnicas avanzadas soportadas por hardware y optimizadas para sistemas modernos. Las implementaciones actuales buscan equilibrar eficiencia, equidad y desempeño, utilizando herramientas como instrucciones atómicas y técnicas de dos fases para maximizar el uso de recursos y garantizar acceso seguro a regiones críticas.

desarrolladores pueden garantizar aplicaciones más eficientes, confiables y escalables.

1. Introducción a estructuras concurrentes y *locks*
Las estructuras de datos concurrentes, como contadores, listas enlazadas, colas y tablas hash, utilizan mecanismos de sincronización (*locks*) para evitar conflictos en entornos multihilo. Estas protecciones aseguran acceso seguro a los datos compartidos, pero pueden introducir problemas de desempeño debido a los tiempos de espera y cuellos de botella.

2. Contadores Concurrentes:
- **Preciso:** Usa un *lock* global para proteger la sección crítica. Es seguro pero no escala bien, ya que múltiples hilos compiten por el acceso.
- **Aproximado:** Divide el contador en locales (uno por núcleo) y un contador global que se actualiza periódicamente. Mejora la escalabilidad sacrificando precisión temporal. Es útil cuando el desempeño es más importante que la exactitud inmediata.

3. Listas Enlazadas Concurrentes:
- **Primera versión:** Usa un *lock* global para proteger las operaciones. Si bien es sencillo, limita la concurrencia y puede ser propenso a errores en funciones con múltiples puntos de entrada/salida.
- **Mejoras:** Solo proteger las regiones críticas (actualización de punteros) con *locks*. Esto reduce el tiempo de bloqueo, mejora el desempeño y simplifica el mantenimiento del código.

4. Colas Concurrentes:
- Implementan encolado y desencolado simultáneo mediante dos *locks* separados (uno para la cabeza y otro para la cola) y un nodo *dummy*. Este diseño permite operaciones concurrentes sin interferencia, maximizando la eficiencia y escalabilidad en aplicaciones multihilo.

5. Tablas Hash Concurrentes:
- Utilizan listas enlazadas concurrentes en cada *bucket* de la tabla. Cada *bucket* tiene su propio *lock*, lo que permite que múltiples hilos operen en paralelo en diferentes partes de la tabla. Esto mejora la concurrencia y el desempeño comparado con una lista enlazada simple, especialmente en aplicaciones con un alto volumen de inserciones y búsquedas.

6. Consideraciones Generales:
- **Escalabilidad:** Las estructuras deben soportar un alto número de hilos sin degradar significativamente el desempeño.
- **Optimización de *locks*:** Proteger únicamente las regiones críticas para reducir el tiempo de espera y minimizar la competencia entre hilos.
- **Balance entre precisión y desempeño:** Algunas aplicaciones permiten sacrificar exactitud inmediata (como en contadores aproximados) para lograr mejor escalabilidad.

Conclusión:
La implementación de estructuras de datos concurrentes es fundamental para aplicaciones modernas que requieren alta concurrencia. Las soluciones discutidas combinan seguridad en el acceso con mejoras de desempeño, adaptándose a las necesidades específicas de cada aplicación. El diseño eficiente de estas estructuras maximiza la concurrencia y minimiza los costos de sincronización, siendo esencial para el desarrollo de sistemas robustos y escalables.

eficiencia, equidad y desempeño, utilizando herramientas como instrucciones atómicas y técnicas de dos fases para maximizar el uso de recursos y garantizar acceso seguro a regiones críticas.

Variables de Condición
Las **variables de condición** son herramientas de sincronización que permiten que los hilos esperen a que se cumpla una condición específica antes de continuar su ejecución. Funcionan en conjunto con bloqueos (*mutex*) para garantizar el acceso exclusivo a recursos compartidos.

1. **Operaciones principales:**
   - **wait**: Pone a un hilo en espera hasta que reciba una señal, liberando el bloqueo asociado mientras espera.
   - **signal**: Notifica a un hilo en espera que puede continuar.
2. **Recomendaciones:**
   - Utilizar estructuras while en lugar de if para volver a verificar la condición después de despertar, mitigando problemas como "despertares perdidos".
3. **Problemas frecuentes:**
   - **Condiciones de carrera:** Acceso desordenado a recursos compartidos puede generar inconsistencias.
   - **Señales perdidas:** Si el hilo que envía la señal lo hace antes de que otro hilo se duerma, la señal puede perderse, dejando hilos bloqueados indefinidamente.

Problema del Productor-Consumidor
El problema involucra dos tipos de hilos:
- **Productor:** Genera ítems y los coloca en un búfer compartido.
- **Consumidor:** Extrae ítems del búfer para procesarlos.

La sincronización es clave para evitar que:
- El productor agregue ítems a un búfer lleno.
- El consumidor intente consumir de un búfer vacío.

Soluciones al Problema
1. **Implementación básica con variables de condición:**
   - Un búfer de capacidad 1 y un contador para rastrear si está lleno o vacío.
   - El productor usa wait si el búfer está lleno y signal para notificar al consumidor cuando produce un ítem.
   - El consumidor usa wait si el búfer está vacío y signal para notificar al productor cuando consume un ítem.
2. **Problemas identificados:**
   - **Despertares perdidos:** Si múltiples hilos comparten una sola variable de condición, las señales pueden no despertar al hilo correcto.
   - **Señales incorrectas:** Un hilo puede despertar pero encontrar una condición que ya no es válida debido al acceso de otro hilo.
3. **Solución mejorada:**
   - Uso de **dos variables de condición**:
     - **empty**: Para manejar la condición de búfer vacío.
     - **full**: Para manejar la condición de búfer lleno.
   - Esto garantiza que:
     - Los productores solo despierten a consumidores.
     - Los consumidores solo despierten a productores.
4. **Extensión a búferes de múltiples posiciones:**
   - Permite la producción y el consumo concurrentes siempre que el contador del búfer esté entre 0 (vacío) y su capacidad máxima (lleno).
   - Mejora la eficiencia al manejar varios ítems simultáneamente.

Conclusión
El uso de variables de condición y bloqueos es fundamental para resolver problemas de sincronización en programación concurrente. El problema del productor-consumidor ilustra cómo aplicar estas herramientas para garantizar un acceso seguro y eficiente a recursos compartidos, mitigando errores como condiciones de carrera, despertares perdidos y señales incorrectas. La solución final, que incluye múltiples variables de condición y búferes escalables, asegura un comportamiento robusto y eficiente en escenarios complejos.

1. Conceptos Fundamentales de Semáforos
- **Definición**:
  - Un semáforo es una estructura de sincronización basada en un contador interno.
  - Se utiliza para controlar el acceso concurrente a recursos compartidos.
- **Operaciones Principales**:
  - **wait (decrementar)**:
    - Reduce el contador y bloquea al hilo si el valor resulta negativo.
  - **post (incrementar)**:
    - Aumenta el contador y despierta a un hilo bloqueado, si lo hay.
- **Tipos de Semáforos**:
  - **Binarios**: Solo permiten valores 0 y 1, equivalentes a un

La concurrencia es fundamental en sistemas operativos modernos, permitiendo que múltiples hilos se ejecuten simultáneamente para mejorar el rendimiento y la fluidez de las aplicaciones. Sin embargo, trabajar con memoria compartida genera desafíos como condiciones de carrera y problemas de sincronización. Los **problemas clásicos de concurrencia** como **Productor-Consumidor**, **Lectores-Escritores** y **Filósofos Comensales** han sido ampliamente estudiados y proporcionan modelos útiles para solucionar situaciones reales. Además, los sistemas concurrentes enfrentan errores que van más allá de los problemas clásicos, como interbloqueos y otros tipos de bugs específicos de la programación concurrente.

Problemas Principales en Concurrencia
1. **Violación de Atomicidad**:
   - Ocurre cuando una operación crítica no es indivisible y

- **post (incrementar)**:
  - Aumenta el contador y despierta a un hilo bloqueado, si lo hay.
- **Tipos de Semáforos**:
  - **Binarios**: Solo permiten valores 0 y 1, equivalentes a un lock.
  - **Generales**: Contadores que pueden tomar valores enteros mayores.

2. Implementación de Semáforos en Linux
- **Diferencias Clave**:
  - En Linux, la actualización del contador en wait se realiza después del bloqueo, manteniendo el valor mínimo del contador en 0.
  - Los semáforos son protegidos con locks y variables de condición para garantizar exclusión mutua en operaciones simultáneas.

3. Problemas Clásicos Resueltos con Semáforos

3.1. Productor-Consumidor
- **Escenario**:
  - Un productor genera ítems que un consumidor consume, usando un búfer compartido.
- **Restricciones**:
  - El productor no puede producir si el búfer está lleno.
  - El consumidor no puede consumir si el búfer está vacío.
- **Solución**:
  - Semáforo empty: Representa espacios vacíos.
  - Semáforo full: Representa espacios llenos.
  - Semáforo mutex: Protege el acceso al búfer para evitar condiciones de carrera.
- **Problemas Adicionales**:
  - **Deadlock**: Resuelto ajustando el orden de adquisición de semáforos.
  - **Múltiples Productores/Consumidores**: Se requieren semáforos adicionales para garantizar exclusión mutua.

3.2. Lectores y Escritores
- **Escenario**:
  - Lectores pueden acceder simultáneamente si no hay escritores.
  - Escritores requieren acceso exclusivo.
- **Restricciones**:
  - Un escritor bloquea el acceso a lectores y otros escritores.
  - Múltiples lectores pueden acceder simultáneamente si no hay escritores.
- **Solución**:
  - Semáforo lock: Protege la estructura de datos.
  - Semáforo writelock: Permite acceso exclusivo a escritores.
  - Contador readers: Rastrea la cantidad de lectores activos.
- **Problema de Starvation**:
  - Escritores pueden sufrir inanición si hay muchos lectores.
  - Solución sugerida: Bloquear la entrada de nuevos lectores si hay escritores esperando.

3.3. Cena de Filósofos
- **Escenario**:
  - Cinco filósofos alternan entre pensar y comer, compartiendo cinco palillos.
  - Cada filósofo necesita dos palillos para comer.
- **Problemas Clásicos**:
  - **Deadlock**: Ocurre si todos los filósofos toman un palillo y esperan el segundo.
  - **Starvation**: Un filósofo puede quedarse sin comer si no adquiere los palillos necesarios.
- **Soluciones**:
  - Cambiar el orden de toma de palillos para un filósofo, evitando la espera circular.
  - Limitar el número de filósofos que intentan comer simultáneamente.
  - Introducir un árbitro que controle el acceso a los recursos.

4. Lecciones Clave
1. **Evitar Deadlocks**:
   - Cambiar el orden de adquisición de recursos o utilizar semáforos adicionales.
2. **Garantizar Equidad (Fairness)**:
   - Priorizar el acceso para evitar inanición en casos de alta concurrencia.
3. **Maximizar Concurrencia**:
   - Permitir múltiples accesos simultáneos cuando no haya conflictos.
4. **Diseño Robusto**:
   - Usar semáforos junto con locks y variables de condición para asegurar sincronización y exclusión mutua.

Conclusión:
Los semáforos son herramientas poderosas y versátiles para resolver problemas de sincronización y exclusión mutua en sistemas concurrentes. Su correcta implementación y uso permiten gestionar recursos compartidos de manera eficiente, evitando problemas críticos como interbloqueos, inanición y condiciones de carrera.

bugs específicos de la programación concurrente.

Problemas Principales en Concurrencia
1. **Violación de Atomicidad**:
   - Ocurre cuando una operación crítica no es indivisible y es interrumpida, permitiendo que otro hilo modifique el estado compartido.
   - Ejemplo: Acceder a un valor nulo tras una validación no atómica.
   - Solución: Implementar exclusión mutua con **locks** para garantizar que sólo un hilo ejecute la operación a la vez.
2. **Violación de Orden**:
   - Se presenta cuando el orden esperado de ejecución entre hilos no se respeta, como cuando un hilo accede a una variable que aún no ha sido inicializada por otro.
   - Solución: Usar herramientas de sincronización como **variables de condición** o **semáforos** para asegurar que las operaciones se ejecuten en el orden correcto.

Interbloqueos:
Un **interbloqueo (deadlock)** ocurre cuando dos o más hilos quedan bloqueados indefinidamente esperando recursos retenidos por los demás.
1. **Condiciones necesarias para un interbloqueo**:
   - **Exclusión Mutua**: Un recurso sólo puede ser utilizado por un hilo a la vez.
   - **Retención y Espera**: Un hilo puede retener recursos mientras solicita otros.
   - **No Expropiación**: Los recursos no pueden ser forzadamente quitados a un hilo.
   - **Espera Circular**: Existe un ciclo de hilos, cada uno esperando un recurso retenido por el siguiente.
2. **Ejemplo práctico**:
   - Hilos adquiriendo recursos en un orden no controlado pueden formar una espera circular, resultando en un interbloqueo.

Manejo de Interbloqueos
Existen diversas estrategias para prevenir, evitar o manejar interbloqueos:
1. **Prevención de Interbloqueos**:
   - Negar al menos una de las condiciones necesarias para que ocurra un interbloqueo:
     - Romper la exclusión mutua mediante recursos compartidos (*lock-free*).
     - Eliminar la retención y espera haciendo que los hilos adquieran todos los recursos necesarios de manera atómica.
     - Permitir la expropiación de recursos.
     - Establecer un orden global de adquisición de recursos para evitar la espera circular.
2. **Evitar Interbloqueos**:
   - Usar planificadores inteligentes que controlen el acceso a recursos y garanticen un estado seguro, como el **algoritmo del banquero**.
   - Limitación: Estas técnicas son costosas y pueden afectar el rendimiento del sistema.
3. **Detección y Recuperación**:
   - Permitir que ocurran interbloqueos y detectarlos mediante un grafo de recursos e hilos.
   - Soluciones:
     - Finalizar hilos específicos.
     - Reiniciar el sistema.
     - Usado en sistemas donde la alta confiabilidad es crucial, como motores de bases de datos.
4. **El algoritmo del avestruz**:
   - Ignorar los interbloqueos y reiniciar el sistema cuando ocurran.
   - Apropiado para sistemas no críticos donde el impacto es mínimo.

Conclusión General
El manejo de concurrencia y los interbloqueos es una parte esencial en el diseño de sistemas concurrentes. Mientras los problemas clásicos de sincronización tienen soluciones bien definidas, los interbloqueos requieren un enfoque equilibrado entre prevención, detección y recuperación. La elección de la técnica adecuada depende del contexto del sistema, su criticidad y los recursos disponibles, asegurando siempre un balance entre rendimiento y confiabilidad.