Modelos de memoria distribuida vs. memoria compartida

En la computación de alto rendimiento, los modelos de memoria desempeñan un papel crucial en la forma en que los sistemas gestionan y acceden a los datos. Dos modelos prominentes son la memoria compartida y la memoria distribuida. Ambos tienen sus ventajas y desafíos únicos y son adecuados para diferentes tipos de aplicaciones y arquitecturas de sistemas.

Memoria compartida

En el modelo de memoria compartida, múltiples procesadores tienen acceso a una región común de memoria. Este enfoque es característico de las arquitecturas multiprocesador simétrico (SMP), donde todos los procesadores comparten la misma memoria física y pueden leer y escribir en ella. Las principales características y desafíos del modelo de memoria compartida son:

- Acceso Uniforme: Los procesadores pueden acceder a cualquier ubicación de la memoria compartida con el mismo tiempo de acceso. Esto facilita la programación, ya que los desarrolladores pueden escribir programas sin preocuparse de la ubicación física de los datos.
- Coherencia de Caché: Uno de los mayores desafíos en los sistemas de memoria compartida es mantener la coherencia de caché. Dado que múltiples procesadores pueden almacenar en caché los mismos datos, es crucial asegurarse de que cualquier modificación se refleje en todas las cachés. Protocolos como MESI (Modificado, Exclusivo, Compartido, Inválido) y MOESI (Modificado, Exclusivo, Compartido, Inválido, Propietario) se utilizan para gestionar esta coherencia.
- Sincronización: La sincronización es vital en los sistemas de memoria compartida para prevenir condiciones de carrera y garantizar la consistencia de los datos. Los mecanismos comunes de sincronización incluyen semáforos, mutexes y barreras. Aunque estos mecanismos son efectivos, pueden introducir sobrecarga y complicar la programación.
- Escalabilidad: La escalabilidad es un desafío significativo en los sistemas de memoria compartida. A medida que se añaden más procesadores, el tráfico en el bus de memoria y la complejidad de mantener la coherencia de caché aumentan, lo que puede degradar el rendimiento.
- Programación: La programación en sistemas de memoria compartida es generalmente más sencilla debido a la uniformidad en el acceso a la memoria. Los desarrolladores pueden utilizar modelos de programación paralela como OpenMP, que simplifican la creación de aplicaciones paralelas.

Estrategias de memoria compartida

En sistemas multiprocesador donde múltiples procesadores acceden y comparten la misma memoria física, es crucial implementar estrategias efectivas para garantizar el rendimiento, la coherencia y la consistencia de los datos. A continuación, se explican las principales estrategias

de memoria compartida, incluyendo la localización de datos, la reducción de contención, y la minimización de la latencia de caché.

1 . Localización de datos

La localización de datos en sistemas de memoria compartida implica organizar y almacenar los datos de manera que se optimice el acceso a la memoria por parte de los procesadores, minimizando la latencia de acceso y mejorando el rendimiento general del sistema.

Ejemplo: En una aplicación de procesamiento de imágenes, almacenar los datos de píxeles que se procesan juntos en ubicaciones contiguas de memoria puede mejorar significativamente el rendimiento debido a un acceso más eficiente a la caché.

Técnicas:

- Localidad espacial: Los datos que se utilizan juntos deben almacenarse cerca unos de otros en la memoria. Por ejemplo, las estructuras de datos contiguas en matrices.
- Localidad temporal: Los datos que se utilizan con frecuencia deben almacenarse en cachés para accesos rápidos. Por ejemplo, variables locales y datos frecuentemente accedidos en cachés L1 o L2.

Ventajas:

- Acceso rápido: Los procesadores pueden acceder rápidamente a los datos que necesitan, mejorando la eficiencia del sistema.
- Mejor utilización de la caché: Almacenar datos relacionados cerca unos de otros maximiza la eficiencia del uso de la caché.

Desafíos:

- Diseño complejo: Organizar los datos de manera eficiente puede ser complejo y requerir un diseño cuidadoso de la memoria.

2 . Reducción de contención

La contención ocurre cuando múltiples procesadores intentan acceder simultáneamente a la misma ubicación de memoria, lo que puede causar conflictos y retrasos. Reducir la contención es crucial para mantener un rendimiento óptimo en sistemas de memoria compartida.

Ejemplo:

En una base de datos concurrente, múltiples transacciones pueden intentar acceder y modificar el mismo registro al mismo tiempo. La contención se puede reducir mediante el uso de particiones y técnicas de control de concurrencia.

Técnicas:

- Descomposición de tareas: Dividir las tareas en subtareas independientes que acceden a diferentes partes de la memoria, reduciendo la necesidad de acceso concurrente a las mismas ubicaciones.
- Bloqueo de granularidad fina: Utilizar bloqueos más específicos (por ejemplo, a nivel de fila en lugar de a nivel de tabla) para reducir la probabilidad de contención.

- Algoritmos concurrentes sin bloqueos: Implementar estructuras de datos y algoritmos que minimicen o eliminen el uso de bloqueos, como listas enlazadas sin bloqueo o pilas concurrentes.

Ventajas:

- Mejor concurrencia: Menos bloqueos y esperas entre procesadores, mejorando la eficiencia y el rendimiento.
- Mayor escalabilidad: Permite que el sistema escale mejor con el aumento del número de procesadores.

Desafíos:

- Complejidad de implementación: Implementar algoritmos concurrentes sin bloqueos puede ser técnicamente desafiante.

3 . Minimización de la latencia de caché

La latencia de caché se refiere al tiempo que tarda un procesador en acceder a los datos almacenados en caché. Minimizar esta latencia es esencial para maximizar el rendimiento en sistemas de memoria compartida.

Ejemplo:

En un sistema de procesamiento de transacciones financieras, es crucial acceder rápidamente a los datos de las cuentas. Mantener estos datos en caché reduce la latencia y mejora el rendimiento del sistema.

Técnicas:

- Prefetching de datos: Anticipar qué datos serán necesarios próximamente y cargarlos en caché antes de que se soliciten.
- Políticas de reemplazo de caché: Utilizar políticas eficientes de reemplazo de caché, como LRU (Least Recently Used) para mantener en caché los datos más relevantes.
- Agrupación de datos: Agrupar datos que se utilizan conjuntamente para que se almacenen juntos en caché y se accedan de manera más eficiente.
- Tamaño óptimo de caché: Determinar el tamaño óptimo de las cachés en diferentes niveles (L1, L2, L3) para equilibrar el espacio de almacenamiento y la latencia de acceso.

Ventajas:

- Mayor rendimiento: Acceso rápido a datos frecuentemente utilizados, mejorando la eficiencia del sistema.
- Reducción de latencia: Menos tiempo de espera para acceder a datos almacenados en caché.

Desafíos:

- Gestión de caché: Determinar qué datos almacenar y cuándo reemplazarlos puede ser complejo y requerir una gestión cuidadosa.
- Coherencia de caché: Mantener la coherencia de los datos en cachés múltiples es un desafío técnico significativo.

4 . Coherencia y consistencia de caché

La coherencia de caché asegura que todas las copias de un dato en diferentes cachés sean iguales. La consistencia de caché garantiza el orden en que las operaciones de memoria son vistas por diferentes procesadores.

Ejemplo:

En una aplicación de simulación científica, varios procesadores pueden acceder y actualizar los mismos datos. Es crucial que todos los procesadores vean los mismos datos actualizados para evitar errores en los cálculos.

Técnicas:

- Protocolos de coherencia (e.g., MESI): Utilizar protocolos de coherencia de caché como MESI (Modificado, Exclusivo, Compartido, Inválido) para asegurar que las actualizaciones de datos se propaguen correctamente a todas las cachés.
- Barreras de memoria: Implementar barreras de memoria para asegurar que todas las operaciones de lectura y escritura se completen antes de que se continúen otras operaciones.

Ventajas:

- Datos consistentes: Asegura que todos los procesadores trabajen con los mismos datos, evitando errores.
- Sincronización eficiente: Permite la sincronización eficiente entre múltiples procesadores.

Desafíos:

- Sobrecarga de comunicación: La propagación de actualizaciones entre cachés puede introducir sobrecarga de comunicación.
- Complejidad técnica: Implementar y gestionar protocolos de coherencia puede ser técnicamente desafiante.

Ejemplos

- Un sistema de memoria compartida típico podría ser un servidor de múltiples núcleos donde todos los núcleos pueden acceder a una base de datos en memoria. Si un núcleo actualiza un registro en la base de datos, todos los demás núcleos deben ver esta actualización inmediatamente, lo cual se gestiona a través de mecanismos de coherencia de caché.

- Consideremos una aplicación de simulación científica que corre en un servidor SMP. Los diferentes núcleos del servidor pueden trabajar en diferentes partes de la simulación, pero necesitan acceder y actualizar una memoria compartida donde se almacenan las variables globales y los resultados parciales de la simulación. Utilizando OpenMP, los desarrolladores pueden paralelizar el bucle principal de la simulación para que cada núcleo trabaje en diferentes iteraciones

Memoria distribuida

En contraste, el modelo de memoria distribuida implica que cada procesador tiene su propia memoria local. Los procesadores se comunican entre sí mediante un mecanismo de paso de mensajes para compartir datos. Este modelo es característico de los sistemas de procesamiento paralelo de memoria distribuida (DMPP) y las arquitecturas de clústeres.

- Autonomía de memoria: Cada procesador tiene acceso rápido a su propia memoria local, lo que reduce la latencia de acceso a los datos locales. Sin embargo, acceder a la memoria de otros procesadores requiere comunicación explícita, lo cual puede ser más lento.
- Paso de mensajes: La comunicación en sistemas de memoria distribuida se realiza mediante el paso de mensajes. Librerías como MPI (Message Passing Interface) son comunes y proporcionan las herramientas necesarias para la comunicación inter-procesador. Aunque el paso de mensajes introduce sobrecarga, es esencial para la coordinación y el intercambio de datos entre procesadores.
- Sincronización: La sincronización en sistemas de memoria distribuida es menos problemática en términos de coherencia de caché, pero sigue siendo crucial para coordinar la ejecución de tareas y el intercambio de datos. Los mensajes deben ser gestionados cuidadosamente para evitar bloqueos y garantizar la entrega correcta.
- Escalabilidad: Los sistemas de memoria distribuida generalmente escalan mejor que los de memoria compartida. A medida que se añaden más procesadores, cada uno con su propia memoria local, se reduce la contención por el acceso a la memoria, permitiendo que el sistema maneje un mayor número de procesadores de manera más eficiente.
- Programación: La programación en sistemas de memoria distribuida puede ser más compleja debido a la necesidad de gestionar explícitamente la comunicación y la sincronización. MPI es una herramienta comúnmente utilizada para este propósito, pero requiere una comprensión profunda de los patrones de comunicación y la estructura de datos distribuida.

Estrategias de memoria distribuida

En sistemas de memoria distribuida, donde cada procesador tiene su propia memoria local y los datos se reparten entre diferentes nodos, es crucial implementar estrategias eficientes para manejar y acceder a los datos. Aquí se explican tres estrategias clave para mejorar el rendimiento en sistemas de memoria distribuida: localización de datos, reducción de contención y minimización de la latencia de caché.

1. Localización de datos: La localización de datos se refiere a la estrategia de almacenar los datos lo más cerca posible de los procesadores que los utilizan con mayor frecuencia. Esto minimiza la necesidad de comunicación entre nodos y reduce la latencia de acceso a los datos.

Ejemplo: En un sistema de recomendación, los datos de un usuario pueden almacenarse en el nodo donde se realizan los cálculos de recomendación para ese usuario. De esta forma, las consultas y actualizaciones de datos son rápidas y eficientes.

Técnicas:

- Particionamiento de datos: Dividir los datos en particiones basadas en ciertos criterios (e.g., geográficos, demográficos) y asignar cada partición a un nodo específico.
- Replicación de datos: Mantener copias de los datos en múltiples nodos para asegurar que los datos estén disponibles localmente en varios nodos, reduciendo la latencia de acceso.
- Consistencia local: Asegurar que las operaciones de lectura/escritura en los datos locales se manejen de manera eficiente, y que las actualizaciones se propaguen a otros nodos según sea necesario.

Ventajas:

- Menor latencia: Acceso más rápido a los datos debido a su proximidad.
- Reducción de tráfico: Menos comunicación entre nodos, reduciendo el tráfico de red.

Desafíos:

- Equilibrio de carga: Asegurar que los datos y la carga de trabajo estén equilibrados entre los nodos.
- Consistencia de datos: Mantener la consistencia de los datos replicados entre diferentes nodos.

1. Reducción de contención: La contención ocurre cuando múltiples nodos intentan acceder a los mismos recursos simultáneamente, lo que puede causar conflictos y retrasos. La reducción de contención se centra en minimizar estos conflictos para mejorar el rendimiento del sistema.

Ejemplo: En una base de datos distribuida, múltiples nodos pueden intentar acceder y actualizar el mismo registro simultáneamente. La contención se puede reducir mediante técnicas de particionamiento y replicación.

Técnicas:

- Particionamiento horizontal: Dividir una base de datos en tablas más pequeñas (shards) y asignar cada shard a un nodo diferente, reduciendo la contención en tablas grandes.
- Bloqueo de granularidad fina: Utilizar bloqueos más específicos en lugar de bloquear grandes regiones de datos. Por ejemplo, en lugar de bloquear toda una tabla, bloquear solo las filas o columnas específicas que están siendo accedidas.
- Control optimista de concurrencia: Permitir que múltiples transacciones se procesen simultáneamente sin bloquearse entre sí y resolver los conflictos solo si ocurren (e.g., utilizando técnicas de versionado).

Ventajas:

- Mejor utilización del sistema: Menos tiempo de espera para los nodos, aumentando la eficiencia del sistema.
- Mayor concurrencia: Permite que más transacciones se procesen simultáneamente.

Desafíos:

- Complejidad de implementación: Estrategias como el control optimista de concurrencia pueden ser complejas de implementar y gestionar.
- Conflictos de datos: Aumenta la posibilidad de conflictos que deben resolverse eficientemente.

1. Minimización de la latencia de caché La latencia de caché se refiere al tiempo que tarda un procesador en acceder a los datos almacenados en caché. Minimizar esta latencia es crucial para mejorar el rendimiento del sistema.

Ejemplo:

En un sistema de procesamiento de datos en tiempo real, como una plataforma de análisis de eventos, es fundamental acceder rápidamente a los datos más recientes. Minimizar la latencia de caché asegura que los análisis se realicen lo más rápido posible.

Técnicas:

- Prefetching de datos: Anticipar qué datos serán necesarios próximamente y cargarlos en caché antes de que se soliciten, reduciendo el tiempo de espera.
- Cache Locality: Organizar los datos en memoria de manera que las operaciones accedan secuencialmente a áreas contiguas de memoria, aprovechando mejor la jerarquía de la memoria caché.
- Políticas de reemplazo de caché: Implementar políticas de reemplazo eficientes (e.g., LRU - Least Recently Used) para mantener en caché los datos que se acceden con mayor frecuencia.
- Caché distribuida: Implementar una caché distribuida que se extiende a través de múltiples nodos, permitiendo que los datos caché se almacenen en varios lugares para un acceso rápido desde cualquier nodo.

Ventajas:

- Mayor velocidad de acceso: Reducción del tiempo necesario para acceder a datos frecuentemente utilizados.
- Mejor rendimiento global: Optimización del uso de la memoria caché y reducción de la latencia.

Desafíos:

- Administración de caché: Determinar qué datos almacenar y cuándo reemplazar datos en la caché puede ser complicado.
- Consistencia de caché: Mantener la coherencia de los datos almacenados en caché en un entorno distribuido puede ser difícil y costoso en términos de recursos.

Ejemplos

- En un sistema de memoria distribuida, podríamos tener un clúster de computadoras donde cada nodo del clúster tiene su propia memoria local. Para resolver un problema conjunto, como un análisis de grandes datos, los nodos deben intercambiar información mediante mensajes. Si un nodo realiza un cálculo que otros nodos necesitan, debe enviar los resultados a los nodos pertinentes.

- Un ejemplo práctico de memoria distribuida es un sistema de recomendación en un clúster de Hadoop. Cada nodo del clúster procesa una parte de los datos de usuario y elementos para generar recomendaciones. Los nodos deben intercambiar datos sobre usuarios y elementos similares para mejorar la precisión de las recomendaciones. Utilizando MPI, los desarrolladores pueden implementar un algoritmo que coordina la comunicación entre nodos para compartir información relevante.

Comparación de modelos

Latencia y rendimiento:

- Memoria compartida: Baja latencia de acceso a memoria compartida, pero alta latencia y complejidad en la coherencia de caché.
- Memoria distribuida: Baja latencia en acceso a memoria local, pero alta latencia en comunicación inter-procesador.

Escalabilidad:

- Memoria compartida: Escalabilidad limitada debido a la contención de memoria y la complejidad de mantener la coherencia.
- Memoria distribuida: Mejor escalabilidad, adecuada para grandes clústeres, pero requiere manejo explícito de comunicación y sincronización.

Complejidad de programación:

- Memoria compartida: Más sencilla debido a la uniformidad de acceso a memoria, pero requiere mecanismos de sincronización.
- Memoria distribuida: Más compleja debido a la necesidad de gestionar el paso de mensajes y la sincronización.

Aplicaciones adecuadas:

- Memoria compartida: Aplicaciones que requieren acceso rápido a datos compartidos y donde la coherencia puede ser gestionada eficientemente.
- Memoria distribuida: Aplicaciones que pueden ser descompuestas en tareas independientes que requieren poca comunicación, o donde la comunicación puede ser optimizada mediante el paso de mensajes.

Ejercicios

1 . Coherencia de caché: Explica cómo se mantiene la coherencia de caché en un sistema de memoria compartida utilizando el protocolo MESI. Incluya ejemplos de transiciones de estados de caché cuando dos procesadores acceden y modifican la misma línea de caché.

Respuesta Esperada:

- Debea describir los cuatro estados del protocolo MESI (Modificado, Exclusivo, Compartido, Inválido) y explicar las transiciones de estados cuando:
  - Un procesador lee una línea de caché por primera vez.
  - Un segundo procesador lee la misma línea de caché.

- El primer procesador modifica la línea de caché.
- El segundo procesador intenta leer la línea de caché modificada.

```
## Tu respuesta
```

2 . Implementa un programa en C utilizando POSIX threads (pthread) que demuestre el uso de mutexes para proteger una variable compartida. El programa debe crear varios hilos que incrementen una variable global compartida de manera segura.

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

#define NUM_THREADS 5
#define NUM_INCREMENTS 1000000

pthread_mutex_t mutex;
int shared_variable = 0;

void* increment(void* arg) {
    for (int i = 0; i < NUM_INCREMENTS; i++) {
        pthread_mutex_lock(&mutex);
        shared_variable++;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    pthread_mutex_init(&mutex, NULL);

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    printf("Final value of shared_variable: %d\n", shared_variable);
    return 0;
}

## Tu respuesta
```

3 . Describe los diferentes modelos de consistencia de memoria (consistencia estricta, consistencia secuencial, consistencia causal) y cómo afectan el comportamiento observable de los programas en un sistema de memoria compartida.

Respuesta Esperada:

Debes explicar:

- Consistencia estricta: Todas las operaciones de memoria son vistas por todos los procesadores en el orden exacto en que ocurren.
- Consistencia secuencial: Las operaciones de memoria de todos los procesadores se intercalan en un orden secuencial que es consistente con el orden de programa de cada procesador.
- Consistencia causal: Solo las operaciones de memoria que son causalmente relacionadas deben ser vistas en el mismo orden por todos los procesadores.

```
## Tu respuesta
```

4 . Explica cómo el paso de mensajes en un sistema de memoria distribuida permite la comunicación entre nodos. Describa las ventajas y desventajas del paso de mensajes comparado con la memoria compartida.

Respuesta Esperada:

Debes explicar:

- Cómo los nodos envían y reciben mensajes para compartir datos.
- Ventajas: No requiere coherencia de caché, escalabilidad mejorada, adecuado para sistemas distribuidos.
- Desventajas: Latencia en la comunicación, mayor complejidad en la programación, sobrecarga de comunicación.

```
## Tu respuesta
```

5 . Compare los modelos de consistencia eventual y consistencia fuerte en sistemas de memoria distribuida. Proporcione ejemplos de aplicaciones donde cada modelo sería más adecuado.

Respuesta Esperada:

Debes describir:

- Consistencia fuerte: Las actualizaciones son visibles instantáneamente a todos los nodos, proporcionando una vista consistente de los datos en todo momento.
- Consistencia eventual: Las actualizaciones se propagan gradualmente y todos los nodos eventualmente alcanzan una consistencia, pero puede haber inconsistencias temporales.
- Ejemplos: Consistencia fuerte es crucial para aplicaciones financieras, mientras que la consistencia eventual es adecuada para redes sociales y sistemas de caching distribuido.

```
## Tu respuesta
```

6 . Implementa un programa en Python que utilice el módulo multiprocessing para demostrar la memoria compartida. Crea varios procesos que incrementen una variable compartida de manera segura utilizando un Value y un Lock.

```
import multiprocessing

def increment(shared_value, lock):
    for _ in range(10000):
        with lock:
            shared_value.value += 1

def main():
    shared_value = multiprocessing.Value('i', 0)
    lock = multiprocessing.Lock()
    processes = []

    for _ in range(4):
        p = multiprocessing.Process(target=increment,
args=(shared_value, lock))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

    print(f"Final value: {shared_value.value}")

if __name__ == "__main__":
    main()

## Tu respuesta
```

7 . Explica la diferencia entre coherencia de caché y consistencia de caché. Proporciona ejemplos de cómo estos conceptos afectan el rendimiento de un sistema multiprocesador.

Respuesta esperada:

Debes explicar:

- Coherencia de Caché: Garantiza que todas las copias de un dato en diferentes cachés sean iguales.
- Consistencia de Caché: Garantiza el orden en que las operaciones de memoria son vistas por diferentes procesadores.
- Ejemplos: Condiciones de carrera debido a la falta de coherencia, problemas de sincronización debido a la falta de consistencia.

```
## Tu respuesta
```

8 . Implementa un programa en Python que simule la coherencia de caché utilizando threading. Crea un sistema donde múltiples hilos modifiquen una variable compartida y utilice bloqueos para garantizar la coherencia.

```
import threading
```

```
shared_value = 0
lock = threading.Lock()

def modify_shared_value():
    global shared_value
    for _ in range(10000):
        with lock:
            temp = shared_value
            temp += 1
            shared_value = temp

def main():
    threads = []

    for _ in range(4):
        t = threading.Thread(target=modify_shared_value)
        threads.append(t)
        t.start()

    for t in threads:
        t.join()

    print(f"Final value: {shared_value}")

if __name__ == "__main__":
    main()

## Tu respuesta
```

9 .Describe cómo funciona un sistema de snoop bus para mantener la coherencia de caché en un sistema multiprocesador. ¿Cuáles son los desafíos asociados con el snoop bus?

Respuesta Esperada:

Debes explicar:

- Funcionamiento del snoop bus: Todas las cachés observan (snooping) el bus de datos para detectar operaciones relevantes y mantener la coherencia.
- Desafíos: Escalabilidad limitada debido al tráfico en el bus, latencia, complejidad de implementación.

```
## Tu respuesta
```

10 . Implementa un programa en Python que simule un sistema de snoop bus utilizando hilos. Cada hilo representa un núcleo con su propia caché y observa una lista compartida de operaciones de memoria.

```
import threading
import time

shared_memory = [0] * 10
```

```
bus_operations = []
bus_lock = threading.Lock()

class Cache:
    def __init__(self, id):
        self.id = id
        self.cache = [0] * 10

    def read(self, index):
        with bus_lock:
            bus_operations.append((self.id, 'read', index))
        return self.cache[index]

    def write(self, index, value):
        with bus_lock:
            bus_operations.append((self.id, 'write', index, value))
        self.cache[index] = value

    def snoop(self):
        while True:
            with bus_lock:
                if bus_operations:
                    op = bus_operations.pop(0)
                    if op[1] == 'write':
                        self.cache[op[2]] = op[3]
            time.sleep(0.01)

def cpu_task(cache, index, value):
    cache.write(index, value)
    print(f"CPU {cache.id} wrote {value} at index {index}")
    time.sleep(1)
    read_value = cache.read(index)
    print(f"CPU {cache.id} read {read_value} from index {index}")

def main():
    caches = [Cache(i) for i in range(4)]
    threads = []

    for cache in caches:
        t = threading.Thread(target=cache.snoop)
        t.daemon = True
        t.start()

    for i, cache in enumerate(caches):
        t = threading.Thread(target=cpu_task, args=(cache, i % 10, i))
        threads.append(t)
        t.start()

    for t in threads:
        t.join()
```

```
if __name__ == "__main__":
    main()

## Tu respuesta
```

11 . Describe el protocolo MESI para la coherencia de caché. Explica los cuatro estados posibles y cómo las transiciones de estado aseguran la coherencia de los datos.

Respuesta Esperada:

Debes explicar:

- Estados del Protocolo MESI: Modificado (M), Exclusivo (E), Compartido (S), Inválido (I).
- Transiciones: Cómo y cuándo ocurren las transiciones entre estos estados basadas en las operaciones de lectura y escritura de los procesadores.

```
## Tu respuesta
```

12 . Implementa un programa en Python que simule el protocolo MESI. Crea una clase CacheLine con los cuatro estados y simule operaciones de lectura y escritura que provoquen transiciones de estado.

```
class CacheLine:
    def __init__(self):
        self.state = 'I'  # Initial state is Invalid
        self.value = None

    def read(self):
        if self.state == 'I':
            self.state = 'S'
            print("Transition to Shared")
        return self.value

    def write(self, value):
        if self.state in ('I', 'S'):
            self.state = 'M'
            print("Transition to Modified")
        self.value = value

    def get_state(self):
        return self.state

def main():
    cache_line = CacheLine()

    # Simulate write operation
    print("Writing value 42")
    cache_line.write(42)
    print(f"State after write: {cache_line.get_state()}")
```

```
    # Simulate read operation
    print("Reading value")
    value = cache_line.read()
    print(f"State after read: {cache_line.get_state()}")

if __name__ == "__main__":
    main()
```

13 . Implementa un programa en C usando OpenMP que utilice una barrera para sincronizar los hilos en diferentes fases de un cálculo. El programa debe dividir un cálculo en dos fases y asegurarse de que todos los hilos completen la primera fase antes de pasar a la segunda.

```
#include <stdio.h>
#include <omp.h>

#define NUM_THREADS 4
#define ARRAY_SIZE 16

int main() {
    int array[ARRAY_SIZE];
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = i;
    }

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int nthreads = omp_get_num_threads();

        // Fase 1: Sumar 10 a cada elemento del array
        for (int i = id; i < ARRAY_SIZE; i += nthreads) {
            array[i] += 10;
        }

        // Sincronización de barrera
        #pragma omp barrier

        // Fase 2: Multiplicar cada elemento por 2
        for (int i = id; i < ARRAY_SIZE; i += nthreads) {
            array[i] *= 2;
        }
    }

    printf("Array final:\n");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
```

```
    return 0;
}

## Tu respuesta
```

14 . Implementa un programa en C utilizando MPI (Message Passing Interface) para sumar los elementos de un array distribuido entre varios procesos. Cada proceso calcula la suma parcial de su porción y luego los resultados parciales se combinan en el proceso raíz.

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE 100
#define ROOT 0

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int local_size = ARRAY_SIZE / size;
    int* array = NULL;
    int* local_array = (int*)malloc(local_size * sizeof(int));

    if (rank == ROOT) {
        array = (int*)malloc(ARRAY_SIZE * sizeof(int));
        for (int i = 0; i < ARRAY_SIZE; i++) {
            array[i] = i + 1;
        }
    }

    MPI_Scatter(array, local_size, MPI_INT, local_array, local_size,
MPI_INT, ROOT, MPI_COMM_WORLD);

    int local_sum = 0;
    for (int i = 0; i < local_size; i++) {
        local_sum += local_array[i];
    }

    int global_sum;
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, ROOT,
MPI_COMM_WORLD);

    if (rank == ROOT) {
        printf("Total sum: %d\n", global_sum);
        free(array);
```

```
    }

    free(local_array);
    MPI_Finalize();

    return 0;
}

# Tu respuesta
```

15 . Implementa un programa en C utilizando POSIX threads (pthread) que implemente el algoritmo de productor-consumidor con un búfer compartido. Usa mutexes y condiciones para sincronizar los accesos al búfer.

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

#define BUFFER_SIZE 10

int buffer[BUFFER_SIZE];
int count = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_producer, cond_consumer;

void* producer(void* arg) {
    for (int i = 0; i < 20; i++) {
        pthread_mutex_lock(&mutex);

        while (count == BUFFER_SIZE) {
            pthread_cond_wait(&cond_producer, &mutex);
        }

        buffer[count++] = i;
        printf("Produced: %d\n", i);

        pthread_cond_signal(&cond_consumer);
        pthread_mutex_unlock(&mutex);

        sleep(rand() % 2);
    }
    pthread_exit(NULL);
}

void* consumer(void* arg) {
    for (int i = 0; i < 20; i++) {
        pthread_mutex_lock(&mutex);

        while (count == 0) {
```

```
            pthread_cond_wait(&cond_consumer, &mutex);
        }

        int item = buffer[--count];
        printf("Consumed: %d\n", item);

        pthread_cond_signal(&cond_producer);
        pthread_mutex_unlock(&mutex);

        sleep(rand() % 3);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t prod_thread, cons_thread;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond_producer, NULL);
    pthread_cond_init(&cond_consumer, NULL);

    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond_producer);
    pthread_cond_destroy(&cond_consumer);

    return 0;
}

## Tu respuesta
```

1. Discute cómo los algoritmos pueden ser optimizados para sistemas de memoria compartida. Incluye estrategias como la localización de datos, la reducción de contención y la minimización de la latencia de caché.

Respuesta Esperada:

Debea explicar:

- Localización de datos: Mantener los datos que son accedidos frecuentemente juntos en memoria para aprovechar la caché.
- Reducción de contención: Usar estructuras de datos concurrentes optimizadas y particionar tareas para minimizar el acceso simultáneo a la misma memoria.
- Minimización de la latencia de caché: Acceder a los datos en patrones que maximicen la eficiencia de la caché, evitando el "cache thrashing".

```
## Tu respuesta
```