Comunicazione con memoria condivisa

In un sistema informatico, la comunicazione tra più processi o thread avviene spesso attraverso uno spazio di memoria condivisa. Questa memoria condivisa consente a diversi elementi del sistema di scambiarsi informazioni in modo efficiente, senza dover utilizzare altre forme di comunicazione. Tuttavia, la memoria condivisa porta con sé una serie di problematiche, poiché ogni processo o thread che accede a questi dati può alterarne lo stato.

Questa lezione esplorerà i concetti alla base della memoria condivisa, con particolare attenzione al problema della corsa critica. Tratteremo il significato della corsa critica, il motivo per cui è importante gestirla e alcune delle tecniche fondamentali per evitare conflitti quando più processi accedono alla stessa risorsa.

Un esempio semplice per introdurre il concetto è quello di un conto bancario. Immaginiamo che più persone possano accedere a un conto contemporaneamente per effettuare operazioni di deposito e prelievo. Se queste operazioni non sono ben coordinate, il risultato finale del saldo potrebbe non riflettere accuratamente le azioni effettuate da ciascun utente.

Un esempio di corsa critica è rappresentato da due persone che tentano di prelevare denaro dallo stesso conto contemporaneamente. Senza sincronizzazione, i due processi potrebbero leggere il saldo allo stesso tempo e sottrarre denaro, portando a un errore nel calcolo finale.

Impostazione dell'Esempio

Immaginiamo di avere una semplice variabile condivisa che rappresenta il saldo di un conto bancario. Supponiamo che due thread, A e B, accedano a questa variabile allo stesso tempo per eseguire delle operazioni di prelievo. Il saldo iniziale è di 100 euro e ogni thread vuole prelevare un certo importo. Questo esempio dimostrerà come la mancanza di sincronizzazione possa causare un errore nel calcolo finale del saldo.

Supponiamo che:

- Thread A voglia prelevare 30 euro
- Thread B voglia prelevare 50 euro

Passo 1: Thread A Legge il Saldo

Il primo passo è quando il Thread A accede alla variabile saldo e legge il valore corrente, che è di 100 euro. Thread A ha intenzione di prelevare 30 euro, quindi calcola che il nuovo saldo dovrebbe essere di 70 euro (100 - 30). Tuttavia, in questo preciso momento, Thread A non ha ancora aggiornato la variabile saldo nel sistema.

Thread A legge il saldo e vede che è di 100 euro. Decide che il nuovo saldo sarà 70 euro, ma non aggiorna ancora la variabile.

Passo 2: Thread B Legge lo Stesso Saldo

Nel frattempo, prima che Thread A possa aggiornare la variabile saldo, Thread B accede alla stessa variabile e legge anch'esso un saldo di 100 euro. Anche Thread B ha intenzione di prelevare 50 euro, quindi calcola che il nuovo saldo dovrebbe essere di 50 euro (100 - 50). Ancora, però, Thread B non ha ancora aggiornato il valore della variabile saldo.

Thread B legge il saldo e vede anch’esso che è di 100 euro. Decide che il nuovo saldo sarà 50 euro, ma non aggiorna ancora la variabile.

Passo 3: Thread A Aggiorna il Saldo

Ora, il Thread A procede con l'aggiornamento della variabile saldo e la imposta a 70 euro. Questo significa che il sistema ora riflette il risultato della sottrazione di 30 euro effettuata da Thread A, ma senza tener conto dell'intenzione di Thread B di prelevare altri 50 euro.

Thread A aggiorna la variabile saldo, e ora il saldo è di 70 euro nel sistema.

Passo 4: Thread B Aggiorna il Saldo

Dopo l'aggiornamento di Thread A, anche il Thread B procede a impostare la variabile saldo, sovrascrivendo il valore esistente con il suo risultato, 50 euro. Il problema è che il nuovo valore di saldo di 50 euro non tiene conto del prelievo precedente di Thread A.

Thread B aggiorna la variabile saldo, e ora il saldo è di 50 euro nel sistema, senza considerare che

Thread A aveva già prelevato 30 euro.

Risultato Finale Errato

Il risultato finale nel sistema è di 50 euro, ma se consideriamo entrambi i prelievi (30 euro da Thread A e 50 euro da Thread B), il saldo corretto dovrebbe essere 20 euro. A causa della mancanza di sincronizzazione, il sistema non ha rilevato il conflitto e ha sovrascritto il valore del saldo più volte, portando a un errore.

Quando più thread accedono e modificano una variabile condivisa senza sincronizzazione, il risultato finale può essere impreciso o errato.

Questo esempio dimostra come la mancanza di sincronizzazione possa portare a errori di concorrenza e risultati imprevisti in programmi che utilizzano variabili condivise. Per evitare questo tipo di errore, è necessario implementare meccanismi di sincronizzazione, come il lock, che impediscono l'accesso simultaneo non autorizzato alla stessa risorsa.

Il Problema della Corsa Critica

La corsa critica è una situazione che si verifica quando due o più processi cercano di accedere e modificare una risorsa condivisa allo stesso tempo. In una corsa critica, il risultato finale dipende dall'ordine in cui i processi accedono alla risorsa, e questo può portare a risultati imprevedibili. Questo problema è particolarmente rilevante nei sistemi multithreading e multiprocessing, dove è comune che più processi operino contemporaneamente.

Per fare un esempio, immaginiamo un sistema di gestione di biglietti per eventi, in cui più persone possono prenotare un posto. Se due persone tentano di prenotare lo stesso posto esattamente nello stesso momento, senza una gestione accurata della memoria condivisa, entrambi potrebbero riuscire ad acquistare lo stesso biglietto.

Il problema della corsa critica si presenta quando due processi cercano di accedere a una risorsa condivisa senza un controllo dell'ordine di accesso.

Il concetto di corsa critica è fondamentale per chi sviluppa software che utilizza memoria condivisa, poiché l'impatto di un accesso non regolato alle risorse può essere devastante, portando a errori di calcolo, conflitti nei dati e instabilità dell'applicazione.

L'Importanza della Sincronizzazione

Per evitare i problemi causati dalla corsa critica, è necessario introdurre un concetto chiave: la sincronizzazione. La sincronizzazione consente di controllare l'accesso alle risorse condivise, assicurando che solo un processo o thread alla volta possa accedere alla risorsa in una sezione critica.

Un esempio semplice di sincronizzazione può essere immaginato come una porta che consente solo a una persona di entrare alla volta in una stanza. Allo stesso modo, la sincronizzazione nel software assicura che solo un processo o thread per volta possa accedere alla risorsa condivisa.

La sincronizzazione è essenziale per garantire l'integrità dei dati e prevenire conflitti. Nei sistemi software, ci sono vari metodi per implementare la sincronizzazione, che esploreremo in lezioni successive.

La sincronizzazione permette di evitare conflitti tra processi, garantendo che l'accesso alle risorse condivise avvenga in modo controllato.

Sezioni Critiche

Una sezione critica è una parte del codice in cui un processo accede a una risorsa condivisa. Durante l'esecuzione di una sezione critica, nessun altro processo dovrebbe essere in grado di accedere alla stessa risorsa per evitare conflitti. Il controllo delle sezioni critiche è un elemento fondamentale nella gestione della corsa critica.

Per immaginare cosa significhi una sezione critica, pensiamo a una cassa in un supermercato. Quando una persona è alla cassa, nessun altro può effettuare il pagamento in quel preciso istante. In modo simile, quando un thread è in una sezione critica, nessun altro thread dovrebbe poter modificare la stessa risorsa.

Immagina che il codice di aggiornamento di un contatore sia una sezione critica: se due processi modificano il contatore contemporaneamente, il valore finale potrebbe non essere corretto. La sezione critica è quindi quella parte di codice che modifica la risorsa in modo sicuro.

Sfide e Benefici della Sincronizzazione

Sebbene la sincronizzazione sia fondamentale per evitare la corsa critica, essa può introdurre altre sfide. Una sincronizzazione troppo rigida può rallentare le prestazioni del sistema, poiché solo un

thread alla volta può accedere alla risorsa condivisa. D’altra parte, se la sincronizzazione è troppo permissiva, si corre il rischio di non proteggere correttamente i dati.

I principali vantaggi della sincronizzazione sono la stabilità e la consistenza dei dati. Essa consente di mantenere i dati in uno stato prevedibile e affidabile, anche quando più processi o thread lavorano contemporaneamente. Tuttavia, è fondamentale trovare un bilanciamento tra la protezione dei dati e l'efficienza del sistema.

La sincronizzazione è essenziale per garantire che i dati condivisi siano sempre accurati, ma deve essere implementata con attenzione per evitare rallentamenti nel sistema.

Questa lezione e' stata realizzata grazie al contributo di:

Risorse per la scuola
https://www.baobab.school

Siti web a Varese
https://www.francescobelloni.it