TANKERINO.com

Lezione 201

https://www.tankerino.com/it/corsi/10/tepi-4/lezioni/201/lock-nella-programmazione-concorrente

L’importanza dei lock nella programmazione concorrente

Quando parliamo di thread, ci riferiamo a delle entità di esecuzione che permettono ad un programma di svolgere più operazioni contemporaneamente. Non diamo nulla per scontato: un thread è come un “filo” di attività che il nostro programma avvia per fare qualcosa, mentre altre parti del codice possono procedere in parallelo.

La sincronizzazione è ciò che ci consente di far lavorare questi thread insieme senza che si creino conflitti. Senza un adeguato meccanismo di sincronizzazione, più thread potrebbero modificare simultaneamente gli stessi dati, causando comportamenti imprevedibili.

Immaginiamo di avere una risorsa condivisa, ad esempio una variabile che conta quanti utenti stanno utilizzando un’applicazione. Se due thread incrementano questa variabile allo stesso tempo senza coordinazione, potremmo perdere un conteggio o ottenere risultati non coerenti.

La risorsa condivisa può essere vista come una stanza a cui accedono più persone. Se non ci sono regole, due persone potrebbero spingere la porta allo stesso momento e bloccarla. La sincronizzazione serve proprio a introdurre queste regole, evitando che l’accesso alla risorsa condivisa avvenga in modo caotico.

In questo contesto, parole come sezione critica iniziano a comparire. Una sezione critica è un pezzo di codice che accede a una risorsa condivisa e che non dovrebbe essere eseguito da più thread simultaneamente. Qui entrano in gioco i meccanismi di lock e il semaforo binario.

Differenze principali

Caratteristica	lock	Semaforo binario
Utilizzo	È specifico per thread che lavorano in memoria condivisa in un unico processo.	Può essere usato per sincronizzare thread in processi diversi.
Sintassi e semplicità	`lock` è semplice da usare e automatico nella gestione delle eccezioni.	Richiede chiamate esplicite a metodi come `Wait` e `Release`.
Rilascio obbligatorio	Il lock viene rilasciato automaticamente quando si esce dal blocco `lock`.	Il rilascio del semaforo deve essere fatto esplicitamente con `Release()`.
Funzionalità avanzate	Non consente timeout o il controllo dello stato.	Può essere configurato con timeout e tentativi di acquisizione non bloccanti.
Multipli rilasci	Non permette di essere rilasciato più volte dallo stesso thread.	Può essere rilasciato più volte, il che potrebbe introdurre errori se non usato correttamente.

Cos’è un lock e perché è importante

Un lock è uno strumento che permette di impedire l’accesso simultaneo di più thread alla stessa risorsa condivisa. Possiamo vederlo come una chiave virtuale: se un thread entra nella stanza (la nostra sezione critica) utilizzando il lock, gli altri thread devono attendere finché quella chiave non viene restituita.

L’idea base è che, prima di entrare nella sezione critica, un thread “prende” il lock. Nessun altro thread potrà prendere lo stesso lock finché il primo non lo rilascia, garantendo così che non ci siano due accessi contemporanei.

Se vogliamo fare un paragone: immaginiamo di avere una sala computer con una sola postazione. Il lock è come una chiave fisica di quella stanza: chi la possiede è l’unico a poter usare il computer, gli altri devono aspettare fuori.

> Un lock garantisce che un solo thread alla volta esegua il codice racchiuso tra le parentesi protette dal lock.

Esempio in C#:

```
object _lockObject = new object();
void IncrementCounter()
{
  lock(_lockObject) // Qui applichiamo il lock
  {
     counter++;
  }
}
```

Le differenze tra lock e semaforo binario

Il lock, come visto, permette l’accesso esclusivo a una sezione critica. Un semaforo binario è simile, ma con qualche differenza sostanziale. Il semaforo binario può essere immaginato come un segnale che può essere “verde” o “rosso”. Se è “verde”, un thread può passare; se è “rosso”, deve attendere.

La differenza sta nel fatto che il semaforo binario è più flessibile: può essere inizializzato con uno stato libero o occupato e può essere rilasciato anche da un thread diverso da quello che lo ha

acquisito, cosa che il lock generalmente non consente.

Inoltre, il lock in C# è un costrutto specifico del linguaggio, molto semplice da usare, mentre il semaforo binario è una struttura più generale e può essere utilizzata in molti contesti diversi, non solo in C#.

> Il lock in C# è un blocco di codice strutturato, mentre il semaforo binario è un'entità più generale che può essere incrementata o decrementata, gestendo un numero di permessi, in questo caso uno solo (binario).

Esempio di utilizzo di un semaforo binario in C#:

```
using System.Threading;
SemaphoreSlim semaphore = new SemaphoreSlim(1,1); // Semaforo binario
void AccessResource()
{
  semaphore.Wait(); // attende che il semaforo sia libero
  try
  {
    // sezione critica
  }
  finally
  {
    semaphore.Release(); // rilascia il semaforo
  }
}
```

Utilizzo pratico del lock in C# su una risorsa condivisa

Quando decidiamo di usare un lock, di solito stiamo proteggendo qualcosa di molto concreto: ad esempio una lista condivisa, un contatore, un file, qualsiasi cosa che più thread potrebbero modificare contemporaneamente. Senza lock, l'accesso concorrente potrebbe generare errori o comportamenti imprevedibili.

> La risorsa condivisa è di solito una struttura dati o un oggetto in memoria che può essere letto e scritto da diversi thread in parallelo.

Facciamo un esempio: abbiamo un servizio web che utilizza un contatore per contare quante richieste

ha gestito. Se due thread incrementano il contatore nello stesso momento, potremmo avere risultati non coerenti. Ecco perché attorno a quell'operazione mettiamo un lock.

Esempio in C# per proteggere un contatore condiviso:

```
int requestCount = 0;
object lockObj = new object();
void HandleRequest()
{
  lock(lockObj)
  {
    requestCount++;
    Console.WriteLine("Richieste gestite: " + requestCount);
  }
}
```

Grazie a questo approccio, ogni thread dovrà attendere il turno per incrementare la variabile, assicurando che non si creino incoerenze nei dati. Senza lock questo tipo di problema sarebbe molto frequente.

Infine, comprendere bene il funzionamento dei lock aiuta a scrivere codice affidabile, prevenendo problemi difficili da individuare come race condition o deadlock. Questi ultimi sono scenari in cui i thread restano bloccati in attesa l'uno dell'altro, senza poter progredire.
