TANKERINO.com

Lezione 203

https://www.tankerino.com/it/corsi/10/tepi-4/lezioni/203/produttore-consumatore

Il problema del produttore-consumatore

Il Problema del produttore-consumatore è un classico problema di sincronizzazione che si affronta nei sistemi concorrenti. Descrive uno scenario in cui dei produttori generano dati e li inseriscono in un buffer condiviso, mentre i consumatori prelevano questi dati per elaborarli.

Immaginiamo una fabbrica dove i produttori producono oggetti e li mettono su una catena di montaggio (il buffer). I consumatori, come operai o robot, prendono gli oggetti dalla catena per lavorarci. Il sistema deve assicurarsi che:

- I produttori non producano più oggetti di quanti il buffer possa contenere.
- I consumatori non prelevino oggetti se la catena è vuota.

Il problema del produttore-consumatore si risolve usando meccanismi di sincronizzazione come i semafori per controllare l'accesso al buffer condiviso.

In questa lezione, esploreremo una soluzione usando i semafori, che sono strumenti fondamentali per sincronizzare i thread.

Cos'è un semaforo?

Un semaforo è una struttura dati utilizzata per gestire l'accesso concorrente a una risorsa condivisa. Può essere immaginato come un contatore che tiene traccia di quante risorse sono disponibili in un sistema.

Un semaforo può essere incrementato o decrementato dai thread per indicare il rilascio o l'acquisizione di una risorsa.

Ad esempio, in un buffer con capacità di 5 elementi, un semaforo può contare quanti spazi liberi ci sono (per i produttori) o quanti elementi ci sono (per i consumatori). Quando un thread produttore aggiunge un elemento, il semaforo viene decrementato; quando un thread consumatore rimuove un elemento, il semaforo viene incrementato.

In C#, possiamo utilizzare la classe Semaphore per implementare un semaforo.

La logica del produttore-consumatore

La soluzione al problema prevede l'uso di due semafori:

- Full: Tiene traccia di quanti elementi ci sono nel buffer, inizializzato a 0 perché il buffer è vuoto all'inizio.
- Empty: Tiene traccia di quanti spazi vuoti ci sono nel buffer, inizializzato alla capacità massima del buffer.

Inoltre, utilizzeremo un lock (o mutex) per garantire che solo un thread alla volta acceda al buffer condiviso.

La sincronizzazione del produttore e del consumatore avviene coordinando l'uso di semafori e lock.

Implementazione pratica in C#

Ecco un esempio completo che mostra come implementare il problema del produttore-consumatore usando i semafori:

```
using System;
using System.Threading;

class ProduttoreConsumatore
{
   private const int BufferSize = 5;
   private static int[] buffer = new int[BufferSize];
   private static int inIndex = 0;
   private static int outIndex = 0;

   private static Semaphore empty = new Semaphore(BufferSize, BufferSize); //
Spazi vuoti
   private static Semaphore full = new Semaphore(0, BufferSize);          //
Elementi presenti
   private static object lockObject = new object(); // Per sincronizzare l'accesso
al buffer

   static void Main(string[] args)
   {
```

```
        // Creiamo thread produttori e consumatori
        for (int i = 0; i < 3; i++) new Thread(Produttore).Start(i);
        for (int i = 0; i < 2; i++) new Thread(Consumatore).Start(i);
    }

    static void Produttore(object id)
    {
        while (true)
        {
            int dato = new Random().Next(1, 100); // Genera un dato casuale
            empty.WaitOne(); // Aspetta che ci sia spazio libero nel buffer
            lock (lockObject)
            {
                buffer[inIndex] = dato;
                Console.WriteLine($"Produttore {id} ha prodotto: {dato} (Posizione
{inIndex})");
                inIndex = (inIndex + 1) % BufferSize;
            }
            full.Release(); // Incrementa il semaforo degli elementi pieni
            Thread.Sleep(new Random().Next(500, 1000)); // Simula il tempo di
produzione
        }
    }

    static void Consumatore(object id)
    {
        while (true)
        {
            full.WaitOne(); // Aspetta che ci siano elementi nel buffer
            int dato;
            lock (lockObject)
            {
                dato = buffer[outIndex];
                Console.WriteLine($"Consumatore {id} ha consumato: {dato}
(Posizione {outIndex})");
                outIndex = (outIndex + 1) % BufferSize;
            }
            empty.Release(); // Incrementa il semaforo degli spazi vuoti
            Thread.Sleep(new Random().Next(500, 1000)); // Simula il tempo di
consumo
        }
    }
}
```

Come funziona il codice?

Vediamo passo per passo come il codice risolve il problema:

- Produttore: Genera un dato, aspetta che ci sia spazio disponibile nel buffer (semaforo empty), accede in modo sicuro al buffer (lock) e poi aggiorna l'indice di inserimento (inIndex).
- Consumatore: Aspetta che ci sia un elemento disponibile nel buffer (semaforo full), preleva un dato in modo sicuro (lock) e aggiorna l'indice di rimozione (outIndex).
- Sincronizzazione: I semafori full ed empty assicurano che i produttori e i consumatori rispettino le regole di sincronizzazione, mentre il lock garantisce che solo un thread alla volta acceda al buffer.
