Lezione 197

https://www.tankerino.com/it/corsi/10/tepi-4/lezioni/197/semafori

Semafori

I semafori sono uno strumento fondamentale per la gestione della concorrenza nei sistemi operativi e nei programmi multithreading. Servono a coordinare l'accesso a risorse condivise tra più thread o processi, garantendo che non si verifichino conflitti.

Immagina di avere un bagno pubblico con una sola porta. Se una persona è già dentro, la porta viene chiusa e nessun altro può entrare finché quella persona non esce. Questo è, in sostanza, il funzionamento di un semaforo binario.

Un semaforo è una variabile o struttura che tiene traccia dello stato di una risorsa e permette o nega l'accesso a più processi o thread.

Ci sono due tipi principali di semafori:

- Semaforo binario: ha solo due stati, 0 (risorsa occupata) e 1 (risorsa libera).
- Semaforo contatore: tiene traccia del numero di risorse disponibili, ad esempio in un sistema con più accessi simultanei.

In un sistema operativo, un semaforo può essere usato per gestire una stampante condivisa: solo un processo alla volta può stampare, mentre gli altri devono aspettare.

Ora esploreremo come funzionano i semafori in dettaglio, vedremo alcuni esempi pratici e discuteremo i loro vantaggi e limiti.

Operazioni principali: Wait e Signal

I semafori utilizzano due operazioni principali:

1. Wait (o P): controlla se la risorsa è disponibile. Se lo è, decrementa il valore del semaforo e consente al processo di accedere alla risorsa. Se non è disponibile, il processo viene bloccato.

2. Signal (o V): incrementa il valore del semaforo per indicare che la risorsa è stata liberata e sblocca eventuali processi in attesa.

Wait e Signal sono operazioni atomiche che garantiscono che i cambiamenti al valore del semaforo siano eseguiti senza interruzioni.

Ecco come potrebbero apparire queste operazioni in pseudocodice:

```
Wait(S):
   if S > 0:
       S = S - 1
   else:
       blocca il processo

Signal(S):
   S = S + 1
```

Nel linguaggio C#, queste operazioni sono implementate utilizzando la classe Semaphore, che vedremo nei prossimi esempi.

Esempio di semaforo binario in C#

Supponiamo di avere due thread che vogliono accedere a una risorsa condivisa, ma solo uno alla volta può farlo. Ecco un esempio di utilizzo di un semaforo binario in C#:

```
using System;
using System.Threading;

class Program
{
   static Semaphore semaphore = new Semaphore(1, 1); // Semaforo binario

   static void AccessResource(string threadName)
   {
       Console.WriteLine($"{threadName} sta tentando di accedere alla
risorsa...");
       semaphore.WaitOne(); // Operazione Wait
       try
       {
           Console.WriteLine($"{threadName} ha ottenuto l'accesso alla risorsa.");
           Thread.Sleep(2000); // Simula l'uso della risorsa
```

```
        }
        finally
        {
            Console.WriteLine($"{threadName} ha rilasciato la risorsa.");
            semaphore.Release(); // Operazione Signal
        }
    }

    static void Main()
    {
        Thread t1 = new Thread(() => AccessResource("Thread 1"));
        Thread t2 = new Thread(() => AccessResource("Thread 2"));

        t1.Start();
        t2.Start();

        t1.Join();
        t2.Join();
    }
}
```

In questo esempio:

- Il semaforo inizia con il valore 1 (risorsa libera).
- Quando un thread chiama `WaitOne()`, il valore viene decrementato a 0, indicando che la risorsa è occupata.
- Quando il thread termina il suo lavoro, chiama `Release()`, incrementando il valore a 1 e liberando la risorsa.

Esempio di semaforo contatore in C#

Ora vediamo un esempio più complesso: gestire più thread con un semaforo contatore. Supponiamo di avere 3 stanze di un hotel e 5 ospiti che vogliono prenotare una stanza.

```
using System;
using System.Threading;

class Program
{
    static Semaphore semaphore = new Semaphore(3, 3); // 3 risorse disponibili

    static void BookRoom(string guestName)
```

```
    {
        Console.WriteLine($"{guestName} sta tentando di prenotare una stanza...");
        semaphore.WaitOne();
        try
        {
            Console.WriteLine($"{guestName} ha prenotato una stanza.");
            Thread.Sleep(2000); // Simula il soggiorno
        }
        finally
        {
            Console.WriteLine($"{guestName} ha lasciato la stanza.");
            semaphore.Release();
        }
    }

    static void Main()
    {
        string[] guests = { "Ospite 1", "Ospite 2", "Ospite 3", "Ospite 4", "Ospite
5" };
        Thread[] threads = new Thread[guests.Length];

        for (int i = 0; i < guests.Length; i++)
        {
            threads[i] = new Thread(() => BookRoom(guests[i]));
            threads[i].Start();
        }

        foreach (var thread in threads)
        {
            thread.Join();
        }
    }
}
```

In questo esempio:

- Il semaforo inizia con un valore di 3, poiché ci sono 3 stanze disponibili.
- Ogni thread che chiama `WaitOne()` occupa una stanza, decrementando il valore del semaforo.
- Quando un ospite lascia la stanza, chiama `Release()`, incrementando il valore del semaforo e liberando una stanza.

Vantaggi e limiti dei semafori

I semafori sono uno strumento potente, ma presentano vantaggi e svantaggi:

Vantaggi:

- Permettono una gestione sicura delle risorse condivise.
- Possono controllare l'accesso a più risorse simultaneamente.
- Supportano sia il controllo binario che quello contatore.

Limiti:

- Possono causare deadlock se non vengono usati correttamente.
- Non prevengono automaticamente il problema della starvation (processi che aspettano troppo a lungo).
- Richiedono attenzione nell'implementazione per evitare bug difficili da individuare.

I semafori devono essere utilizzati con attenzione per garantire che non si verifichino problemi di deadlock o starvation.

Con una corretta implementazione, però, i semafori sono uno strumento indispensabile per la programmazione concorrente.

Questa lezione e' stata realizzata grazie al contributo di:

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

FrancescoBelloni.it
software development
Siti web a Varese
https://www.francescobelloni.it