Appunti Concurrency

Diego Oniarti

Anno 2024-2025

Contents

1 Goals of the course

- get a general understnding of the issues
- develop the ability to reason about
  - what can go wrong under interference
  - what can be safely made mode efficient by concurrent cooperation

Più nello specifico vedremo algoritmi di sincronizzazione, tecniche per ragionare formalmente sulla concorrenza, e i fondamentali: cos'è fattibile, cosa non lo è, e perché.

Il libro che segue la Quaglia è "*The art of multiprocessor programming, revised print*".

1.1 Modalità d'esame

Orale obbligatorio e progetto opzionale. Il progetto rende l'orale più semplice. Il progetto andrebbe fatto prima di Natale.

2 Concurrent vs Sequential

Un programma sequenziale deve occiparsi solo della "**safety**". Quindi assicurarsi che l'output sia corretto per un dato input.

Un programma concorrente, invece, deve occuparsi sia della safety che della "**liveness**". Questo è un'aspetto che non ha eguali nella programmazione sequenziale. Con liveness definiamo la capacità di un programma di eseguire correttamente a tempo indeterminato senza supervisione dall'utente (credo).
Pensare alla safety nella programmazione concorrente è molto più complesso che nella programmazione sequenziale, e deve essere garantita a prescindere dall'ordine di esecuzione dei thread.

3 Underlying Model

Assumiamo:

- Thread Multipli (processi)
- Singola memoria condivisa
- Gli oggetti risiedono in memoria
- Delay asincroni inprevedibili

4 Road Map

Inizieremo parlando di principi e use-cases dopo.

5 Esempio numeri primi

Immaginiamo di voler usare 10 processi per calcolare i numeri primi da 0 a 10^10. L'intuizione naive sarebbe quella secondo cui questo è 10 volte più veloce rispetto a farlo con un processore singolo. Ma raramente abbiamo questo tipo di improvement.

Primo approccio Il promo approccio potrebbe essere quello di dividere il range in 10 e associare ad ogni thread uno dei 10 range in cui calcolare i numeri primi.

Questa opzione non è buona per il fatto che il task dell'ultimo thread è molto più complesso del task del primo. Questo è un problema di "load balancing".

Metodo più dinamico Usare un contatore condiviso. Ogni processo prende il primo numero libero che trova e testa se è primo.

Il problema ora è la gestione del counter condiviso. Se più thread tentano di accedere al counter in modo non atomico potrebbero sovrascriverlo con valori sbagliati o leggere valori non aggiornati.

6 Semafori

Questo non è safe, in quanto la lettura e la scrittura della variabile `temp` non sono sincrone.

```
public class Counter {
    private long value;

    public long getAndIncrement() {
        temp = value;
        value = temp+1;
        return temp;
    }
}
```

Affrontare la safety è diverso in diversi linguaggi. In Java possiamo usare la keywork "synchronized"

```
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized {
            temp = value;
            value = temp+1;
        }
        return temp;
    }
}
```

Safety and Liveness. Safety: Niente di brutto deve succedere.
Liveness: Prima o poi deve succedere qualcosa di bello.

7 Amdahl's Law

Questa formula ci indica quanto più veloce un programma può essere reso utilizzando concorrenza:

$$speedup = \frac{1}{1 - p + \frac{p}{n}}$$

dove p è la percentuale di codice che può essere parallelizzata e n è il numero di processi.

8 Mutual Exclusion

8.1 Time And Events

Un evento a_0 di un thread A è *istantaneo* e *non* esistono eventi simultanei.
Possiamo vedere un thread come una *macchina a stati*.

State Differenziamo lo stato di un thread dallo stato del sistema.
State:

- program counter
- local variables

System state:

- shared variables
- thread states

Intervalli Un intervallo $A_0 = (a_0, a_1)$ è il tempo tra gli eventi a_0 e a_1.

8.1.1 Precedenza

Possiamo dire che un intervallo *precede* un'altro intervallo $(a_1 < b_0)$

Irreflexive Never true that $I \to I$

Antisymmetric $I \to J \implies I \not\to J$

Transitive $I \to J$ and $J \to K \implies I \to K$

Partial

8.1.2 Eventi Ripetuti

a_0^k indica la k-esima occorrenza dell'evento a_0, qual'ora esso si ripeta.
Lo stesso si può fare con l'intervallo A_0^k.

8.2 Locks

Definiamo due metodi *lock* e *unlock* che incapsulreanno la sezione critica.

```
public class Counter {
    private long value;
    private Lock mylock;
    public long getAndIncrement() {
        // Acquisisci il lock
```

```
        mylock.lock();
        try {
            // Sezione critica
            int temp = value;
            value = value+1;
        } finally {
            // Rilascia il lock a ogni costo, anche in caso di errore
            mylock.unlock();
        }
        return temp;
    }
}
```

java. Usiamo Java perché è il linguaggio usato nel libro di testo. Non per altri motivi.

8.3 Deadlock-Free

Se un qualche thread chiama *lock()*, primo a poi un qualsiasi thread deve acquisire il lock.

8.4 Starvation-Free

Se un qualche thread chiama *lock()*, prima o poi deve acquisire il lock.

8.5 Soluzioni per due/n thread

Inizieremo con soluzioni che risolvono il problema con due thread prima di muoverci a soluzioni più generali e complesse.
Indicheremo con i il thread corrente e j l'altro.

8.5.1 LockOne

```
class LockOne implements Lock {
    private boolean[] flag = new boolean[2];
    public void lock() {
        // Setta la flag e aspetta che l'altra chiuda
        flag[i] = true;
        while (flag[j]) {}
    }
    public void unlock() {
        flag[i] = false;
    }
}
```

Questa prima implementazione primitiva garantisce mutual exclusion ma non è deadlock-free.
Se i due thread settano la flag allo stesso momento entrano in deadlock.

8.5.2 LockTwo

```
class LockTwo implements Lock {
    private int victim;
    public void lock() {
        victim = i;
        while (victim==i) {}
    }
    public void unlock() {}
}
```

8.5.3 Peterson's Algorithm

```
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim==i) {};
}
public void unlock() {
    flag[i] = false;
}
```

Questo algoritmo risolve sia mutual exclusion che starvation.

L'ordine delle operazioni è importante. Se cambiassimo l'ordine delle attribuzioni di `flag[i]` e `victim` rompiamo già la mutua esclusione.

L'algoritmo può essere visto come un "filtro". Dove settare la flag indica il desiderio di un thread di entrare nella CS, e la variabile victim funge da discriminatore.

8.5.4 Filter Algorithm

L'algoritmo di Peterson può essere generalizzato a più thread con una soluzione a "livelli"

```
int[] level;
int[] victim;

public Filter(int n) {
    public void lock() {
        for (int L=1, L<n; L++) {
            level[i] = L;
            victim[L] = i;
```

```
            while (((exists k!= i) level[k]>=L) && victim[L]==i) {};
        }
    }
    public void unlock() {

    }
}
```

Questo algoritmo è starbation free e garantisce mutua esclusione. Non è buono a livello di fairness però.

8.6 Bounded Waiting

Possiamo dividere un metodo *lock* in due parti. Il "*doorway interval*" e il "*waiting intervall*". Per garantire un qualche livello di fairnes dobbiamo imporre un upper bound alla durata dell'intervallo di waiting.

Un thread non può "superare" un altro più di r volte. r è un superparametro imposto dallo sviluppatore.

superare. Diciamo che un thread A ne "supera" uno B quando A entra ed esce dalla sezione critica mentre B rimane bloccato nel waiting della lock.

8.6.1 Bakery Algorithm

L'algoritmo del fornaio assicura un ordine FIFO per i thread in attesa.

```
boolean[] flage;
Lavel[] lavel;
public void lock() {
    // doorway
    flag[i] = true;
    label[i] = max(label[0],...,label[n-1])+1;
    // waiting
    while (forall k != i:
        (flag[k]
        &&
        (label[k],k) << (label[i],i)
        )
    ) {};
}
```

$(lavel[a], a) << (label[b], b)$ ordine lessicografico.

L'algoritmo a livello teorico funziona perfettamente. L'implementazione è però difficile o impossibile.

1. Le lable esplodono a numeri infiniti, causando overflow
2. Ci possono essere conflitti durante il calcolo di max

3. Ci possono essere conflitti durante il calcolo del while

Gli ultimi due punti possono essere risolti dall'ordine lessicografico ma il primo no.

9 Precedence Graph

One can construct a

- wait-free (no mutual exclusion)
- concurrent
- timestamping system
- that never overflows

Un grafo dove an edge from x to y:

- x is later timestamp
- missing
- missing

Quando usiamo *grafi di precedenza* non facciamo ricorso alla transitività per capire "chi viene prima", ma sono gli archi.

10 Numero minimo di variabili

Non esiste un algoritmo che assicuri mutua esclusione tra n thread con meno di n variabili.

Assumiamo per assurdo che esista un algoritmo di questo tipo. Possiamo poi dimostrare che un algoritmo di questo tipo rompe la mutua esclusione (provando la tesi per assurdo).

Definiamo dei termini.

- S è *idle* se tutti i thread sono nella loro reminder region.
- $S \sim_A S'$ due stati sono *indistinguibili* se
  - Le variabili locali sono uguali
  - Le variabili globali sono uguali

Ora dei fatti:

1. Any thread A tunning solo from S, with either S idle or $S \sim I$ and I idle, can reach CS
2. Any thread A that from an idle state reaches CS running solo must write something in shared memory before getting in.

11 Tight bound: n bite for n threads

12 Hoare Logic

La *Hoare Logic* era inizialmente utilizzata per ragionare sulla correttezza di programmi sequenziali. È un tipo di analisi sintattica (quindi statica).

- At the basis of all deductive verification techniques
  Anche PVS usato dall'NSA si basa sulla Hoare Logic
- Used to specify properties of squential imperative programs by means of triples

  $$\{p\}S\{q\}$$

  where
  - S is a statement of the language
  - p is the *pre-condition*, an assertion about a relevant property which the state holds before S is exectuted
  - q is a *post-condition*, an assertion that holds after S

  In altre parole
  - if S starts execution in a state satisfying p
  - and if the execution of S terminates
  - Then the state reached after the execution of S satisfyes q

12.1 Language of assertions

Assertions are formulas in a propositional loguc

$$\begin{aligned} A ::= & A \; and \; A \\ & | A \; or \; A \\ & | true \\ & | false \\ & \ldots \\ S ::= & skip \; | x := E | S; S | if B then S else S | while(B) S \end{aligned}$$

Evert state sarisfies $true$ and no state sarisfies $false$. Henche fpr every p and for every S

$$\{p\}S\{true\}$$

is a valid triple. and $\{false\}S\{p\}$ is a valid triple.

es:

$$\begin{array}{cc} \{x = 5\}x := x + 1\{x = 6\} & \text{è valido} \\ \{x = 5 \ and \ y = 2\}x := x + 1\{x = 6 \ and \ y = 3\} & \text{non è valido} \\ \{x = 5 \ and \ y = 2\}x := x + 1\{x = 6 \ or \ y = 3\} & \text{è valido} \end{array}$$

12.2 Proof System

A proof system is defines so that: if the proof system accepts $\{p\}S\{q\}$, $\{p\}S\{q\}$ is valid.

12.2.1 Proof outline

Un modo più compatto di scrivere derivazioni nel proof system per non avere alberi enormi.

es:

$$\begin{array}{c} \{x + 1\} \\ \{x + 1 = y + 1\} \\ x := x + 1 \\ \{x = y + 1\} \\ y := y + 1 \\ \{x = y\} \end{array}$$

es:

$$\begin{array}{c} \{x = 5 \ and \ y = 2\} \\ \{x = 5\} \\ \{x + 1 = 6\} \\ x := x + 1 \\ \{x = 6\} \\ \{x = 6 \ or \ y = 3\} \end{array}$$

es:

$$\begin{array}{c} \{x = 3\} \\ \{x > 2\} \\ \{x + 1 > 3\} \\ x := x + 1; \\ \{x > 3\} \\ \{x + 2 > 5\} \\ x := x + 2; \\ \{x > 5\} \end{array}$$

13 2024-10-31

- test and set: Molti cache miss
- test and test and set: Meno cache miss
- exponential backoff: Meno chase miss ancora ma dipende molto da parametri dati

14 Linearizability

Each method should take effect instantaneously between invasion an response evenest.

A linearisable object is one whose possible executions are linearisable.
Split method Calls into two events:

- invocation
  q.enq(x)
- response
  q.enq(x) returns void / x / empty

15 Metodi di sincronizzazione

Il modo più becero per parallelizzare un set è aggiungere un lock che blocca l'intera data structure e la sblocca.
Questo approccio non è un granché, in quanto vorremmo che più thread possano svolgere operazioni in simultaneo.

15.1 Course grained sync

Un lock blocca l'intera struttura dati.

15.2 Fine Grained Synchronization

Una soluzione è quella di proteggere con dei lock porzioni più piccole della data structure.
Idealmente la suddivisione deve essere la più alta possibile.

15.3 Optimistic Synchronization

Search without locking. If you find it, lock and check. (Ok: we are done, NOK: start over)

Questo approccio è più veloce in quanto non richiede un lock, ma gli errori diventano molto costosi.

15.4 Lazy Synchronization

Postpone hard work.
Prendiamo per esempio la rimozione di un nodo. Iniziamo facendo una rimozione "logica" del nodo (marcarlo come cancellato) e più avanti facciamo la rimozione fisica più costosa.

15.5 Lock-Free Synchronization

Non usare lock ma istruzioni come CAS.
Questo approccio è molto più difficile da implementare e alcune volte aggiunge un grande overhead. Ma non fa alcuna assunzione sullo scheduler.

16 Implementazione Set

I set avranno 3 metodi. (add, remove, has) Possiamo implementare un set come una linked list. Ogni nodo ha

1. una key (hash di solito)
2. un valore
3. un puntatore a un nodo

Nella lista terremo le key ordinate e 2 nodi aggiunti all'inizio e la fine (∞ e $-\infty$)

16.1 Invarianti

1. la tail è raggiungibile dalla head.
2. il set non ha duplicati
3. la lista rimane ordinata (in funzione delle chiavi)

16.2 remove(x) add(x) fine-grained

Possiamo implementare un fine-grained sync approach usando 2 lock per scorrere la lista fino al punto desiderato. Poi sostituiamo il puntatore da $x-1$ a x con quello a $x+1$
In maniera analoga, per aggiungere un nodo alla lista dobbiamo prendere il lock sul predecessore di x e il successore. O posso fare con un solo lock? Non ho capito

16.3 Optimistic

Cerchiamo il punto nella lista e ci prendiamo il lock senza prenderlo sugli elementi prima durante lo scrolling. Dopo aver preso i lock facciamo dei controlli:

- che la head raggiunga il primo nodo

- che il secondo possa raggiungere la tail
- che il primo e il secondo siano ancora adiacenti

Se uno di questi controlli fallisce dobbiamo lasciare i lock e iniziare di nuovo

Questi controlli ci costringono a fare almeno un altro traversing completo della lista, che può diventare costoso su set molto grandi.

16.4 Lazy

Like optimistic, expect:

- scan once
- contains(x) never locks...

We add a flag to the cells, which tells us whether the value is inside the set or not.

17 Wrap Up

- Hoare logic
- simulations between I/O automata ciao

18 Presentazione