Giorno 7
Programmazione Concorrente

Sommario

Si presentano gli aspetti e i problemi principali della concorrenza, utilizzando un modello che simula il comportamento a livello macchina e descrivendo le soluzioni ed i costrutti offerti da alcuni linguaggi di alto livello.

1 Concorrenza e parallelismo

1.1 Concorrenza vs parallelismo

Quando si parla di **concorrenza** ci si riferisce ad un insieme di task (processi o thread)[1] che competono e si alternano su un singolo processore.

Per **parallelismo** si intende invece un insieme di task che operano in parallelo, su diverse CPU.

1.2 Esecuzione non sequenziale

Se due processi sono eseguiti concorrentemente o parallelamente, vi sono diversi possibili ordini di esecuzione delle singole istruzioni.

Si definisce **atomica** un'operazione **non ulteriormente divisibile**, e.g. un'istruzione assembler.

1.2.1 Primo esempio

Assumendo per ora che si possa **assegnare atomicamente un valore ad una variabile**, qual è il valore di n al termine dell'esecuzione concorrente dei seguenti processi?

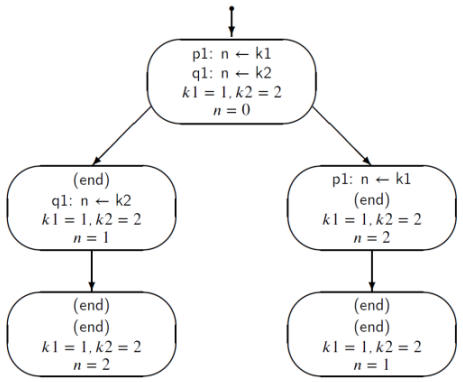
Sia `int` $n \leftarrow 0$ variabile condivisa:

```
1 // PROCESSO P
2       int k1 <- 1
3 p1: n <-k1
```

```
1 // PROCESSO Q
2     int k2 <- 2
3 q1: n <-k2
```

Il risultato sarà sicuramente 1 oppure 2.
Sulla destra è riportato un **sistema di transizioni** che descrive tutti i possibili passi di computazione che il programma può fare.
Tale sistema di transizioni è **non deterministico** (l'ordine dipende dalla tecnica di scheduling utilizzata).

Il problema di questo esempio è che, nella pratica, un assegnamento di questo tipo non è atomico...



M. Ben-Ari. Principles of Concurrent and Distributed Programming, Second edition © M. Ben-Ari 2006

...infatti un comando di assegnamento è generalmente trasformato in più di un'istruzione:

```
                  load  R1, k
n = k + 1  ⟹      add   R1, #1
                  store R1, n
```

[1] Differenza: il sistema operativo esegue più **processi**; ogni processo è composto da uno o più **thread**. vd AESO

Anche analizzando il primo esempio **senza l'assunzione di atomicità**, il comportamento del programma rimane però lo stesso, dato che gli unici interleaving possibili sono le permutazioni di:

`load R1, k1` → `store R1, n` → `load R1, k2` → `store R1, n`

Ed elencando tutti i casi si vede che il risultato è lo stesso.

E in caso di parallelismo? Anche se si provasse a scrivere la stessa variabile parallelamente, **l'hardware definisce un ordine**, quindi non ci sono problemi, ed il comportamento rimane lo stesso.

1.2.2 Secondo esempio

Sia `int` $n \leftarrow 0$ variabile condivisa:

```
// PROCESSO P
p1: n <- n + 1
```

```
// PROCESSO Q
q1: n <- n + 1
```

In questo caso c'è effettivamente una differenza tra il caso dell'assegnamento atomico e non:
In caso di assegnamento atomico, il valore di n alla fine del programma è necessariamente 2.

Se invece si fa un'**analisi a basso livello**:

Sia `int` $n \leftarrow 0$ variabile condivisa:

```
// PROCESSO P
p1: load  R1, n
p2: add   R1, #1
p2: store R1, n
```

```
// PROCESSO Q
q1: load  R1, n
q2: add   R1, #1
q2: store R1, n
```

Queste operazioni possono essere interleaved in modo che n alla fine sia 1.

1.2.3 Simulare il comportamento a livello macchina

Introduciamo una simulazione che "agisce sul programma":
Invece di scrivere `n ← n + 1` scriveremo `n ← !n + 1`, che rende esplicita l'operazione di variabile.

$$\begin{bmatrix} n = 0 \\ n \leftarrow !n + 1 \end{bmatrix} \longrightarrow \begin{bmatrix} n = 0 \\ n \leftarrow 0 + 1 \end{bmatrix} \longrightarrow \begin{bmatrix} n = 1 \\ (end) \end{bmatrix}$$

1.3 Shared memory vs Message Passing

Ci sono due modi per far comunicare processi tra loro:

- Memoria condivisa
- Scambio di messaggi

1.3.1 Memoria condivisa

Sia $x = 0$ una variabile condivisa

```
// Thread 1
producer(){
    int k = 6;
    x = fattoriale(k);
}
```

```
//Thread 2
consumer(){
    while (x == 0) sleep(10);
    print(x);
}
```

In questo modo il thread 2 si sveglia ogni 10 secondi per controllare se il dato del producer è pronto.

Si potrebbero utilizare delle funzioni di **segnalazione** tra thread, fornite dal **runtime** del linguaggio:

```
// Thread 1
producer(){
    int k = 6;
    x = fattoriale(k);
    wakeup();
}
```

```
//Thread 2
consumer(){
    while (x == 0) sleep();
    print(x);
}
```

Dove con **sleep** il consumer si mette in attesa (non necessariamente attiva, a differenza della $sleep(t)$) e **wakeup** sveglia il consumer (il dato è pronto).

1.3.2 Message passing

Non utilizziamo una variabile globale, ma due funzioni **send** e **receive**:

```
// Thread 1
producer(){
    int k = 6;
    x = fattoriale(k);
    send(x);
}
```

```
//Thread 2
consumer(){
    int y = receive();
    print(y);
}

```

Il trasferimento del dato da un processo all'altro è a carico del sistema operativo.

1.3.3 Meccanismi di sincronizzazione

- Busy waiting
- sleep e wakeup
- send e receive
- ...

Sono tutti meccanismi di sincronizzazione usati da **thread** e **processi**, messi a disposizione dal runtime del linguaggio (per i thread) e dal sistema operativo (per i processi).

2 Modello della programmazione concorrente

2.1 Il linguaggio

2.1.1 Sintassi

$$e ::= n \mid e + e \mid !l \mid l := e \mid skip \mid e; e$$

Tipi: int, unit.
[**regole di tipo e semantica**: vedi slide, regole intuitive per linguaggio imperativo.]

2.1.2 Estensione concorrente

Estendiamo la sintassi con $e|e$, l'esecuzione concorrente di due espressioni. Chiameremo queste due espressioni **processi** (anche se si comportano in modo più simile ai **thread**), e definiamo il tipo **proc**, "supertipo di **unit**".

2.1.3 Operazioni atomiche

Assumiamo che +, !, := e *skip* siano atomiche.

2.1.4 Esempio

Eseguiamo $(l := 1+!l)|(l := 7+!l)$, nella memoria $\{l \mapsto 0\}$
Se andiamo a disegnare il sistema di transizioni che descrive tutti i comportamenti possibili (molto grande anche per questo piccolo esempio, vd slide) notiamo che ci sono, di nuovo, vari diversi outcome, alcuni dei quali non sono ciò che ci aspettiamo! Notiamo analizzando il grafo che in questo caso tutti e soli i cammini che portano a risultati sbagliati fanno prima tutte le letture e poi tutte le scritture.

Ci serve un meccanismo di **mutua esclusione** che ci permetta di "rendere atomiche" alcune parti del programma.

2.2 Mutua esclusione

Un **mutex** è un'entità astratta associata ad un'area di memoria da accedere in mutua esclusione.

Estendiamo ulteriormente il linguaggio:

$$e ::= n \mid e + e \mid !l \mid l := e \mid skip \mid e; e \mid e|e \mid \textbf{lock}\ m \mid \textbf{unlock}\ m$$

dove $m \in \mathbb{M}$ (insieme dei **mutex**);

Nelle regole semantiche del linguaggio aggiungiamo l'insieme dei Mutex [vd slide]; la semantica intuitiva è come segue:

- Se nessuno sta accedendo all'area di memoria protetta dal mutex (i.e. se **nessuno ha ancora interagito con il mutex** oppure **l'ultima interazione con il mutex è stata una unlock**) allora il thread che chiama la **lock** procede;
- Se qualcuno sta accedendo all'area di memoria (i.e. **ultima interazione = lock**) allora il thread che chiama la lock si mette in attesa che chi ha chiamato la **lock** chiami l'**unlock**.

2.2.1 Uso dei mutex

Il programma di prima può essere riscritto come segue:

$$e = (\textbf{lock}\ m; l := 1+!l; \textbf{unlock}\ m)|(e = \textbf{lock}\ m; l := 7+!l; \textbf{unlock}\ m)$$

In questo modo eliminiamo la possibilità di fare tutte le letture prima delle scritture, ed otteniamo il risultato corretto: **gli assegnamenti sono eseguiti come se fossero atomici**.

2.2.2 Coarse-grained vs Fine-grained locking

- Strategia **coarse-grained**: un mutex per tutte le locazioni di memoria, e.g.

$$\textbf{lock}\ m\ ;\ l_1 := 1 + !l_2\ ;\ \textbf{unlock}\ m$$

  Caratteristiche: Meno lavoro al singolo thread (una sola lock), ma concorrenzza ridotta

- Strategia **fine-grained**: un mutex diverso per ogni locazione di memoria

$$\textbf{lock}\ m_1; \textbf{lock}\ m_2\ ;\ l_1 := 1 + !l_2\ ;\ \textbf{unlock}\ m_1; \textbf{unlock}\ m_2$$

  Caratteristiche: Preferibili per la maggiore concorrenza, ma **hanno un problema**...

$$\begin{aligned} e = &(\textbf{lock}\ m_1; \textbf{lock}\ m_2\ ;\ l_1 := !l_2\ ;\ \textbf{unlock}\ m_1; \textbf{unlock}\ m_2) \\ &\mid (\textbf{lock}\ m_1; \textbf{lock}\ m_2\ ;\ l_2 := !l_1\ ;\ \textbf{unlock}\ m_1; \textbf{unlock}\ m_2) \end{aligned}$$

Un'analisi del sistema di transizioni ci porta a scoprire che **esistono ordini di esecuzione tali che il programma si blocca**, perché i mutex si aspettano a vicenda.
Questa situazione prende il nome di **deadlock**.

2.2.3 Metodi per evitare il deadlock

- **Deadlock prevention**: si stabiliscono regole controllabili staticamente che garantiscano che i deadlock non possano verificarsi.
- **Deadlock avoidance**: Il supporto a runtime si accorge che il programma sta per andare in deadlock ed interviene cambiando l'ordine di esecuzione dei thread.
- **Deadlock recovery**: Il programma viene lasciato libero di andare in deadlock, ma se ciò avviene il runtime del linguaggio se ne accorge ed interviene per ripristinare uno stato senza deadlock.

2.2.4 Esempio di deadlock prevention: Two-Phase Locking

Il locking avviene in due fasi:

- Una fase "ascendente" in cui si eseguono le lock
- Una fase "discendente" in cui si eseguono le unlock, nell'ordine **opposto** rispetto a quello delle lock.

Applicando tale disciplina al programma di prima si ottiene:

$$\begin{aligned} e = &(\textbf{lock}\ m_1; \textbf{lock}\ m_2\ ;\ l_1 := !l_2\ ;\ \textbf{unlock}\ m_2; \textbf{unlock}\ m_1) \\ &\mid (\textbf{lock}\ m_1; \textbf{lock}\ m_2\ ;\ l_2 := !l_1\ ;\ \textbf{unlock}\ m_2; \textbf{unlock}\ m_1) \end{aligned}$$

Che non può andare in deadlock.

3 Costrutti di concorrenza nei linguaggi di alto livello

3.1 Java

In Java i thread hanno ognuno il proprio stack ma lo **heap condiviso** (shared memory).

3.1.1 Creare thread in Java

```
class Hello extends Thread{
   public void run() {
       System.out.println("Hello Thread!");
   }
}

class Main{
   public static void main(String[] args){
       Thread t = new Hello();
       t.start();                          // NON RUN!!
   }
}
```

Si noti che si chiama il metodo `start`, e non il metodo `run`! Chiamare quest'ultimo accederebbe al metodo della classe, facendo esecuzione sequenziale, mentre chiamare `start` crea il nuovo thread.

Un altro modo di creare un thread è il seguente:

```
class HelloRunnable implements Runnable{
   public void run() {
       System.out.println("Hello Thread!");
   }
}

class Main{
   public static void main(String[] args){
       Runnable r = new HelloRunnable();
       Thread t = new Thread(r);
       t.start();
   }
}

```

In questo modo ad uno stesso thread si possono passare diversi runnable da eseguire.

Altro modo: classe anonima:

```
class Main{
   public static void main(String[] args){
       Runnable r = new Runnable(){
           public void run() {
               System.out.println("Hello Thread!");
           }
       };
       Thread t = new Thread(r);
       t.start();
   }
}
```

Inoltre, si può usare una **lambda espressione**, utile per creare oggetti con un solo metodo:

```
class Main{
   public static void main(String[] args){
       Runnable r =
           () -> {System.out.println("Hello Thread!"); };

       Thread t = new Thread(r);
       t.start();
   }
}
```

3.1.2 Meccanismi di sincronizzazione

Un primo meccanismo di sincronizzazione può essere un metodo della classe che implementa Thread, chiamato dal main (l'oggetto di quella classe è comunque visibile al main, shared memory) che scrive su una variabile (e.g. un booleano) usata dal thread per comunicare un evento
[vd esempio Timer delle slide].

Questo può andar bene quando tutte le interazioni tra Thread vanno in una sola direzione, o quando non ci sono interazioni, ma non sempre.

Per **eseguire due metodi della stessa classe in mutua esclusione** si utilizza il modificatore **synchronized**:

```
...
public synchronized void incr() {n=n+1;}
public synchronized void decr() {n=n-1;}
...
```

Problema: il modificatore synchronized è **coarse grained**, cioè esegue in mutua esclusione gli interi metodi! Per un approccio più fine-grained, si usa:

```
...
 if (val%2==0) { synchronized (this) { pari = pari+1; } }
...
```

Dove **this** è usato come "oggetto che contiene la mutex".

In Java ogni oggetto contiene una mutex, quindi possiamo utilizzare un qualsiasi oggetto, anche **this**.

Bisogna però stare attenti, quando si usano strategie di locking coarse-grained, a non finire in deadlock (usando tecniche come il two phase locking).

3.1.3 Concorrenza nelle Collections

Alcune collections hanno metodi synchronized: in particolare **Vector** e **HashTable**. Gli altri non ne hanno, ma la classe **Collections** mette a disposizione metodi che trasformano classi non synchronized in synchronized, come `Collections.synchronizedList`, che prende come argomento un oggetto di una qualsiasi classe che implementa `List`.

3.2 Javascript

Javascript usa `async` per definire una funzione che è eseguita senza interrompere il resto del codice (eseguita quindi su un nuovo thread), ed `await` per far attendere la fine dell'esecuzione di una funzione `async`. Se si passa alla funzione asincrona una funzione `callback`, la si può far chiamare alla fine dell'esecuzione.

3.3 Go

Prevede un costrutto `go` per definire funzioni asincrone (**goroutines**); le funzioni si sincronizzano **scambiandosi messaggi su canali**: `c <- 1` manda il segnale 1 sul canale, e `<-c` attende di ricevere.

3.4 Erlang

In Erlang `tutto è un processo`, ossia una **componente in esecuzione concorrente** con `memoria separata` dagli altri: **no memoria condivisa**, solo **scambio di messaggi**.

- `spawn` avvia un nuovo processo
- `dest ! msg` invia il messaggio **msg** al processo **dest** (invio non bloccante)
- `receive` mette il programma in ascolto per uno o più messaggi. (receive bloccante)