Programmazione concorrente

SCD

Anno accademico 2008/9
Sistemi Concorrenti e Distribuiti

Tullio Vardanega, tullio.vardanega@math.unipd.it

Premesse – 1

- L'espressività di un linguaggio – naturale o di programmazione – ne è anche il limite
  - Ciò che il linguaggio non prevede o non consente di "dire" non esiste
    - Questo è anche il caso della concorrenza
- Molti linguaggi "storici" sono sequenziali
  - Il flusso di programma è unico per ogni esecuzione
    - Anche se dati variabili ne possono modificare il cammino

Premesse – 2

- La realtà è intrinsecamente concorrente
- Tale è anche la maggior parte dei sistemi a controllo *software*
  - Esiste una varietà di tecniche per rappresentare il parallelismo inerente di molte attività
- Come progettare un linguaggio di programmazione per sistemi concorrenti?

Linguaggi concorrenti – 1

- Un linguaggio sequenziale può esprimere concorrenza tramite l'uso di librerie di sistema
  - Sia entro un singolo programma che tra programmi distinti
    - Per esempio, in ambiente Unix, usando fork()/exec() oppure *socket*
  - L'espressione e il controllo della concorrenza è al di fuori del linguaggio
    - Con ciò causando problemi semantici e di portabilità
- Un linguaggio concorrente può esprimere la compresenza di più flussi di controllo
  - Il compilatore crea un ambiente d'esecuzione capace di creare e gestire entità e azioni concorrenti
    - *Run-time environment* come macchina virtuale

Linguaggi concorrenti – 2

- Il progetto di un linguaggio concorrente si ispira a un dato modello di concorrenza di riferimento
  - Molte scelte possibili
  - La programmazione concorrente agevola la rappresentazione dell'attività di sistemi complessi
  - Ma è anche difficile ed esposta al rischio di importanti errori concettuali
  - Per questo occorre che il modello di riferimento sia valido
    - Espressivo ma anche verificabile

Forme di concorrenza – 1

- Chiameremo processo il singolo flusso di controllo all'interno di un programma
  - Cosa definisce lo "stato" di un processo?
- Concettualmente l'esecuzione di un processo può prevedere che
  a) Tutti i processi condividano uno stesso elaboratore
  b) Ciascun processo possieda un elaboratore proprio e tutti gli elaboratori condividano un banco di memoria comune
  c) Ciascun processo possa avere un elaboratore proprio e gli elaboratori, pur connessi, non condividano memoria

Forme di concorrenza – 2

- Ciascuna di queste 3 forme (e i loro ibridi) comportano tecniche realizzative diverse
  - Definiamo paralleli quei processi che, a ogni istante, sono simultaneamente in esecuzione → i casi b) e c)
  - Definiamo concorrenti quei processi che sono capaci di esecuzione parallela
- Parallelismo → concorrenza realizzata
- Concorrenza → parallelismo potenziale

Forme di concorrenza – 3

- La concorrenza è più generale del parallelismo
  - E di conseguenza concettualmente più importante

> **Programmazione concorrente è il nome dato a notazioni e tecniche usate per esprimere parallelismo potenziale e per risolvere i problemi di sincronizzazione e comunicazione correlati con tale espressione.**
>
> **Il vantaggio della programmazione concorrente è di consentire lo studio del parallelismo senza doversi confrontare con le relative problematiche di realizzazione.**
>
> **Ben-Ari, *Principles of Concurrent Programming*, 1982**

Forme di concorrenza – 4

- La programmazione concorrente non è il solo modo di sfruttare *hardware* parallelo
  - Parallelismo a grana grossa *vs.* parallelismo a grana fine
- Altri modelli di elaborazione sono più adatti ad ambiti di calcolo parallelo
  - Processori vettoriali (*vector processors*)
    - Per eseguire simultaneamente più operazioni matematiche su elementi di strutture dati
      - I processori convenzionali sono detti "scalari" perché trattano un elemento alla volta
  - Architetture *data-flow*
    - Operano simultaneamente su tutti i dati in ingresso che siano completamente disponibili
      - Non hanno bisogno di *program counter*

Forme di concorrenza – 5

- Principio base di ingegneria del *software*
  - Selezione e uso di strumenti e metodi di sviluppo adatti alle caratteristiche del dominio applicativo
- Ambito: applicazioni inerentemente concorrenti
  - Tutte quelle il cui *software* interagisce direttamente con componenti *hardware* interne ed esterne
  - Sistemi *embedded*

Un modello di concorrenza – 1

- Entità attive
  - Capaci di intraprendere azioni di propria iniziativa (se forniti delle necessarie risorse di elaborazione)
- Entità reattive
  - Eseguono azioni solo in risposta a richieste esplicite
  - Risorse → hanno stato interno e impongono (pre-, post-) condizioni di accesso (p.es. mutua esclusione)
  - Entità passive → non impongono condizioni di accesso (p.es. senza stato interno)

Un modello di concorrenza – 2

- La realizzazione di entità risorse richiede capacità di controllo sulle condizioni di accesso
  - Agente di controllo
- Agente di controllo come entità passiva
  - Risorsa protetta (p.es. un semaforo)
- Agente di controllo come entità attiva
  - *Server* (uno speciale processo)

Un modello di concorrenza – 3

Tipo Entità		Realizzabile da
Attiva		Processo
Reattiva	Risorsa protetta	Modulo con agente di controllo (attivo o passivo)
	Server	Processo
	Passiva	Modulo senza agente di controllo

Il progetto di un programma concorrente che usi questo modello richiede il riconoscimento di queste entità nel problema

Un modello di concorrenza – 4

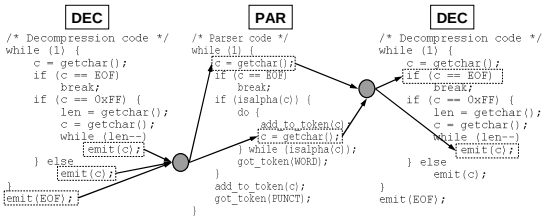
- **La realizzazione di questo modello richiede fino a 3 categorie di primitive**
  - **Processo (entità attiva)**
  - **Agente di controllo passivo per risorse protette**
    - Basso livello di astrazione
    - Efficienza d'esecuzione
    - Inflessibilità
  - **Agente di controllo attivo**
    - Alto livello d'astrazione
    - Proliferazione di processi, con elevati costi di gestione e d'esecuzione
    - Flessibilità

Espressione di concorrenza – 1

- ***Coroutine* – 1 : la storia**
  - **Una delle prime e più rudimentali modalità espressive di concorrenza espressa a programma**
    - Melvin E. Conway, *Design of a separable transition-diagram compiler*, Communications of the ACM, 6(7), July 1963
  - **Esplicita alternanza d'esecuzione tra strutture concorrenti**
    - Tramite comando `resume` oppure `yield-to`
  - **Modella una tecnica di rappresentazione della simulazione discreta → SIMULA 67**
  - **Presente in Modula-2, linguaggio concorrente "storico"**
    - Ma inadeguata alla programmazione concorrente
  - **Ora anche in Ruby 1.9.0 (http://www.ruby-lang.org/)**

Espressione di concorrenza – 2

- ***Coroutine* – 2 : il problema**

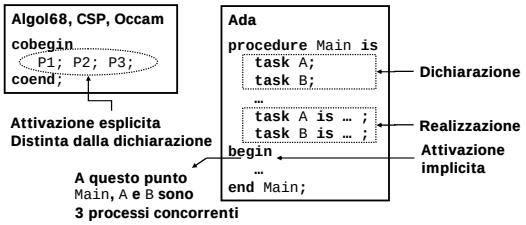


Espressione di concorrenza – 3

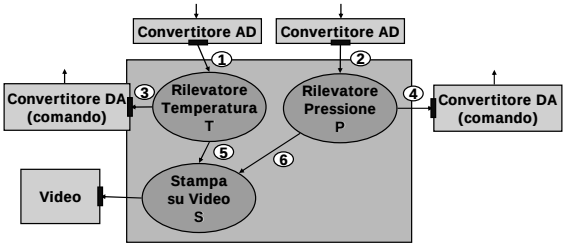
- ***Coroutine* – 2 : la soluzione**
  - **L'invocazione di `emit(c)` in DEC deve**
    - Salvare il valore di 'c' in uscita in un luogo noto
    - Salvare l'indirizzo dell'istruzione di ritorno
    - Saltare all'istruzione corrente del programma PAR
      - PAR ha più punti di ingresso → non è un sottoprogramma
  - **L'invocazione di `getchar()` in PAR deve**
    - Salvare l'indirizzo dell'istruzione di ritorno
    - Saltare all'istruzione corrente del programma DEC
      - DEC ha più punti di ingresso → non è un sottoprogramma
  - **Serve una istruzione speciale di invocazione**
    - Appunto `resume` oppure `yield-to` che svolga le azioni richieste
    - L'istruzione di salto è il punto di ritorno più recentemente salvato dal chiamato

Espressione di concorrenza – 4

- **Dichiarazione e attivazione di processo**



Un esempio – 1



T e P devono mantenere temperatura e pressione del sistema entro limiti di sicurezza e stampare a video i valori rilevati

Un esempio – 2

- **T e P sono entità attive**
- **S è una entità passiva**
  - ***Server* o risorsa protetta**
- **Almeno 3 possibili realizzazioni**
  - **Completamente sequenziale**
    - Ignorando il parallelismo potenziale di T, P, S
  - **T, P, S scritti in linguaggio sequenziale ma trattati come processi distinti**
    - Tramite chiamate al sistema operativo
  - **Usando un linguaggio concorrente**

Un esempio – 3

- **Studiamo le 3 possibili realizzazioni ...**
  - **Soluzione completamente sequenziale**
  - **Soluzione con primitive di sistema operativo**
  - **Soluzione in linguaggio concorrente**

Un esempio – 4

- **Soluzione completamente sequenziale**
  - **Forza un ordinamento artificioso tra moduli indipendenti**
    - P.es.: prima controllo temperatura, poi controllo pressione
  - **Può ritardare o impedire del tutto l'esecuzione di azioni indipendenti ma programmate come successive**
  - **Non tiene conto di possibili differenze nel ciclo operativo di produzione dei dati**
    - P.es.: controllo temperatura 2 secondi, controllo pressione ogni 5 secondi
      - Frequenze non armoniche sono un gran problema
    - Tenerne conto a programma comporterebbe gravi oneri di attesa attiva (*busy wait*)

Un esempio – 5

- **Soluzione con primitive di sistema operativo**
  - **Migliore rispetto alla soluzione sequenziale**
    - Non comporta attese attive
    - Attua un minimo di separazione logica tra moduli tra loro indipendenti
  - **Il controllo dell'esecuzione concorrente è demandato al sistema operativo**
    - Leggere il programma non ci aiuta a capirne il funzionamento
      - L'invocazione di servizi di sistema operativo ne ostacola la comprensione
    - Il comportamento del programma dipende dalle scelte del SO sottostante
      - Portabilità?
      - Verifica?

Un esempio – 6

- **Soluzione in linguaggio concorrente**
  - **La logica dell'applicazione è fissata completamente nel e dal programma**
    - Interpretazione garantita dal linguaggio di programmazione
  - **Operazioni di lettura valori dai dispositivi (1, 2) di tipo non bloccante permettono a ciascun processo di operare a frequenza autonoma**
  - **Ipotesi troppo semplicistiche sulla risorsa S**
    - Attualmente modellata come entità passiva
      - Non è realistico

Un primo raffronto

- **Fattori a favore della concorrenza espressa a linguaggio**
  - **Programmi più leggibili → maggiore manutenibilità**
  - **Indipendenza dal sistema operativo → maggiore portabilità e idoneità all'uso in ambienti a risorse ristrette**
- **Fattori contrari all'espressione di concorrenza a linguaggio**
  - **Il linguaggio deve assumere uno specifico modello di concorrenza → perdita di generalità**
  - **La realizzazione del modello può confliggere con il sistema operativo sottostante → grande sforzo e rischi di distorsione semantica**

La dimensione temporale – 1

- **In molti sistemi l'esecuzione deve relazionarsi con il tempo di sistema**
  - **Un orologio fisico approssima il trascorrere del "tempo"**
  - **L'orologio diventa la sorgente del valore "tempo"**
  - **Varie scelte per rappresentare questo "tempo"**
    - Ora del giorno espressa in secondi e frazioni nell'arco di 24 ore
      - Oppure: maggiore granularità (e quindi maggiore accuratezza)
    - Tempo monotono crescente
      - Relazionato o meno con il valore "umano"
    - Misurazione di intervalli

La dimensione temporale – 2

```
declare
   Start, Finish : Ada.Calendar.Time;
   Interval : Ada.Calendar.Duration := 2.5;
begin
   Start := Ada.Calendar.Clock;
   … -- actual work
   Finish := Ada.Calendar.Clock;
   if Finish - Start > Interval then
      raise Overrun_Error;
   end if;
end;
```

Questa tecnica assume implicitamente un solo flusso di controllo!

```
Ada.Real_Time.Time;
Ada.Real_Time.Time_Span := Ada.Real_Time.To_Time_Span(2.5);
Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(2500);
```


La dimensione temporale – 3

- **L'orologio può anche essere usato a fini di sincronizzazione temporale**
  - **Sospensione relativa**

```
delay 10.0; -- valore di tipo Ada.Calendar.Duration
```

    - Dal tempo in cui viene presa in considerazione
    - La durata della sospensione **non è inferiore** alla richiesta
      - Nessun limite superiore garantito
      - Il tempo esatto di sospensione è **ignoto**
  - **Sospensione assoluta**

```
delay until A_Time; -- valore di tipo Ada.Real_Time.Time
```

    - Riferisce un valore temporale indipendente dal tempo di esecuzione della richiesta
      - Il tempo di risveglio richiesto è garantito entro una latenza data

La dimensione temporale – 4

```
Start := Clock;
First_Action;
delay until (Start + 10.0);
Second_Action;
```

A

```
Start := Clock;
First_Action;
delay (Start + 10.0 - Clock);
Second_Action;
```

B

A e B non hanno lo stesso effetto perché ●, in presenza di prerilascio, è azione interrompibile
il valore assunto da `Clock` quando valutato non è noto a priori

L'effetto del prerilascio nel calcolo di (`Start + 10.0`) non ne influenza il valore assoluto e dunque non perturba l'effetto di sospensione assoluta

La dimensione temporale – 5

Avendo accesso all'orologio e usando sospensioni assolute possiamo facilmente programmare vere attività periodiche

```
with Ada.Real_Time; use Ada.Real_Time;
…
task body Periodic_Task is
   Interval : constant Time_Span := Millisecond(10_000);
   Next_Time : Time := <System_Start_Time>;
begin
   loop
      delay until Next_Time;
      Periodic_Action;
      Next_Time := Next_Time + Interval;
   end loop;
end Periodic_Task;
```


Programmazione concorrente
La dimensione temporale – 6
La regolarità temporale delle attività periodiche è esposta a 2 fattori di rischio
Deviazione locale (local drift)
La vera distanza temporale che separa due successive invocazioni della stessa attività periodica
Inevitabile, può essere migliorata solo mediante migliore realizzazione del linguaggio (granularità di accesso all'orologio)
Deviazione cumulativa (cumulative drift)
L'effetto a catena causato dalla possibile varianza nel completamento delle attività precedenti
Evitabile tramite l'uso di sospensioni assolute, come nel riquadro precedente
Corso di Laurea Magistrale in Informatica, Università di Padova
31 / 31