Programmazione concorrente

Anno accademico 2012/13
Sistemi Concorrenti e Distribuiti

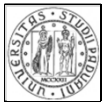
Tullio Vardanega, tullio.vardanega@math.unipd.it

Premesse – 1

- L'espressività di un linguaggio ne è il potere ma anche il limite
  - Ciò che il linguaggio non prevede o non consente di esprimere non esiste
  - *The limits of my language are the limits of my mind. All I know is what I have words for*
    Ludwig Wittgenstein, Tractatus Logico-Philosophicus, 1922
- Questo è anche il caso della concorrenza nei linguaggi di programmazione

Premesse – 2

- Molti linguaggi "storici" sono sequenziali
  - Un unico luogo del controllo
- Questo riflette l'architettura della macchina di von Neumann
  - Il modello concettuale del primo *stored-program computer* per l'esecuzione automatica di un programma
  - Una singola memoria per dati e istruzioni e una CPU che esegue un ciclo infinito *fetch-decode-read-execute-write*
- Un modello di esecuzione intrinsecamente sequenziale



Premesse – 3

- Ma la realtà è intrinsecamente concorrente quando non parallela
  - Che relazione c'è tra concorrenza e parallelismo?

- Concorrente è anche la maggior parte dei sistemi a controllo SW
  - Esiste una varietà di tecniche per rappresentare il parallelismo inerente di molte attività
- Come progettare un linguaggio di programmazione per sistemi concorrenti?
  - Proveremo a rispondere a questa domanda …

Linguaggi concorrenti – 1

- **Un linguaggio sequenziale può generare 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()
  - **L'espressione e il controllo della concorrenza sono in questo caso posti al di fuori del linguaggio**
    - Con ciò causando problemi semantici e di portabilità
- **Un linguaggio concorrente può esprimere la compresenza di più luoghi del 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 deve ispirare a un modello di concorrenza di riferimento coerente e consistente**
  - **Molte scelte possibili per livello di astrazione e potere espressivo**
  - **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 in modo economico

Forme di concorrenza – 1

- **Chiameremo processo il singolo flusso di controllo all'interno di un programma**
  - **Cosa definisce lo "stato" di un processo?**
  - **Perché ci importa saperlo definire?**
- **Concettualmente la modalità di esecuzione di un processo può essere tale che**
  a) **Tutti i processi condividano lo stesso elaboratore**
  b) **Ciascun processo possieda un elaboratore proprio e tutti gli elaboratori condividano memoria**
  c) **Ciascun processo possa avere un elaboratore proprio e gli elaboratori, pur interconnessi, non condividano memoria**

Forme di concorrenza – 2

- **Ciascuna delle forme a) – c) e i loro ibridi comportano tecniche realizzative diverse**
  - **Definiamo paralleli quei processi che, a un dato istante, sono simultaneamente in esecuzione**
    - I casi b) e c), particolarmente nel caso dei nuovi processori *multicore*
  - **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
  - Quindi 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.
>
> M. Ben-Ari, *Principles of Concurrent Programming*, 1982

Importante inciso

- Dati n processi e m processori
  - Per $1 = \mathrm{m} < n$ un buon modello concorrente di soluzione a un problema ha buone virtù architetturali e consente massimo utilizzo della CPU
  - Per $1 < n \leq m$ l'esecuzione di quel sistema ha *speed-up* $\leq n$ che dipende dal grado di parallelismo della soluzione concorrente
  - Per $1 < n \ll m$ serve una progettazione esplicitamente parallela per la quale i modelli di concorrenza classici non sono adeguati

Forme di concorrenza – 4

- La programmazione concorrente non è l'unico modo di sfruttare HW parallelo
  - Da parallelismo a grana grossa a parallelismo a grana fine
- Vi sono modelli di elaborazione più adatti ad ambiti di calcolo parallelo
  - Processori vettoriali
    - Modello di esecuzione SIMD (*single instruction multiple data*)
  - Architetture *data-flow*
    - Non von Neumann (concettualmente senza *program counter*)

Forme di concorrenza – 5

- Criterio: applicazione di un principio base di ingegneria del *software*
  - Selezione e uso di strumenti e metodi di sviluppo adatti alle caratteristiche del dominio del problema
- Ambito: applicazioni inerentemente concorrenti
  - Tutte quelle il cui SW specializzato interagisce direttamente con componenti HW 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- e 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**
  - Semaforo o monitor

- **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 passivo
	Server	Processo
	Passiva	Modulo senza agente di controllo

Il progetto di un programma concorrente che usi questo modello di concorrenza 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**
  - Per processi (entità attive)
  - Per agenti di controllo passivi (semaforo o monitor)
    - Basso livello di astrazione
    - Efficienza d'esecuzione
    - Inflessibilità
  - Per agenti di controllo attivi (server)
    - Maggior livello d'astrazione
    - Maggiori costi di gestione e d'esecuzione
    - Maggiore flessibilità algoritmica

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
  - **Ma anche in Ruby v1.9.0 (http://www.ruby-lang.org/) e altri**

Espressione di concorrenza – 2

```
var q := new queue

coroutine produce
 loop
  while (q not full)
   <create item>
   <add item to q>
   yield to consume
   <point of resumption>

coroutine consume
 loop
  while (q not empty)
  <remove item from q>
  <use item>
  yield to produce
  <point of resumption>
```

- **Le coroutine che ritornano conservano il loro *record* di attivazione**
  - **I sottoprogrammi lo perdono**
  - **Le coroutine sono *continuations***
- **Di conseguenza**
  - **Le coroutine hanno più punti di attivazione**
    - I sottoprogrammi solo uno
  - **Da una stessa coroutine si può ritornare più volte**
    - Mentre da un sottoprogramma una sola

Espressione di concorrenza – 3

- **Dichiarazione e attivazione di processo**

Algol68, CSP, Occam

```
cobegin
  P1; P2; P3;
coend;
```

Attivazione esplicita
Distinta dalla dichiarazione

Ada

```
procedure Main is
   task A;
   task B;
   …
   task A is … ;
   task B is … ;
begin
   …
end Main;
```

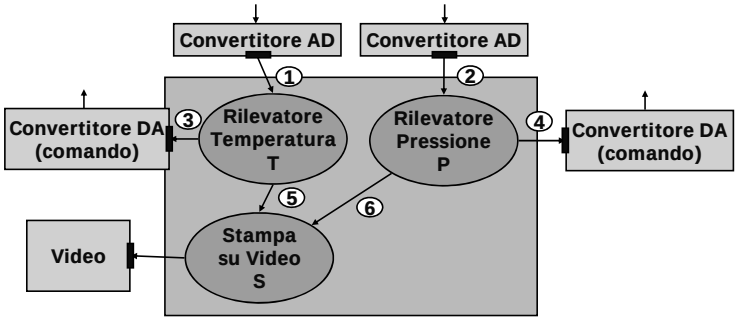
Dichiarazione

Realizzazione

Attivazione implicita

A questo punto `Main`**,** `A` **e** `B` **sono 3 processi concorrenti**

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 ma con uso concorrente
  - Meglio vederla come *server* o risorsa protetta
- Almeno 3 possibili realizzazioni
  - Completamente sequenziale, ignorando il parallelismo potentiale di T, P, S
  - Scrivendo T, P, S in linguaggio sequenziale ma «promuovendoli» a entità concorrenti tramite chiamate al sistema operativo
  - Usando un linguaggio concorrente

Un esempio – 3

- Studiamo le 3 possibili realizzazioni ...
  - Soluzione sequenziale
  - Soluzione con primitive di S/O
  - 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 ogni 2 secondi, controllo pressione ogni 5 secondi
    - Tenerne conto a programma comporta attesa attiva (*busy wait*)

Un esempio – 5

- Soluzione con primitive di S/O
  - Migliore rispetto alla soluzione sequenziale
    - Non richiede attesa attiva
    - Attua un minimo di separazione logica tra moduli tra loro indipendenti
  - Il controllo sulla concorrenza è demandato al S/O
    - Leggere il programma non ci aiuta a capirne il funzionamento
      - L'invocazione di servizi di S/O ne ostacola la comprensione
    - Il comportamento del programma dipende dalle scelte del S/O sottostante
      - Portabilità?
      - Verifica?

Un esempio – 6

- **Soluzione in linguaggio concorrente**
  - **La logica e la semantica dell'applicazione sono fissate completamente 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**
  - **Ma nel codice di esempio abbiamo fatto ipotesi semplicistiche sulla risorsa S**
    - Realizzarla come entità passiva non è sufficientemente generale

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

- **Alcune classi di sistemi richiedono controllo su e garanzie che certe uscite vengano sempre prodotte quando atteso**
  - **Questo requisito collide con il desiderio di massimizzare l'uso delle risorse di calcolo (*throughput*)**
  - **Il sistema va dimensionato per soddisfare i requisiti nel caso peggiore**
- **In questo tipo di sistemi occorre**
  - **Specificare quando certe azioni devono essere iniziate**
  - **Specificare quando certe azioni devono essere completate**
  - **Rispondere a situazioni in cui i requisiti temporali non possono essere soddisfatti**
  - **Rispondere a situazioni in cui i requisiti temporali cambiano dinamicamente**

La dimensione temporale – 3

```
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 – 4

- **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 – 5

```
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 – 6

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;
```

La dimensione temporale – 7

- **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

La dimensione temporale – 8