Requisiti di programmazione concorrente

SCD

Anno accademico 2018/19
Sistemi Concorrenti e Distribuiti

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

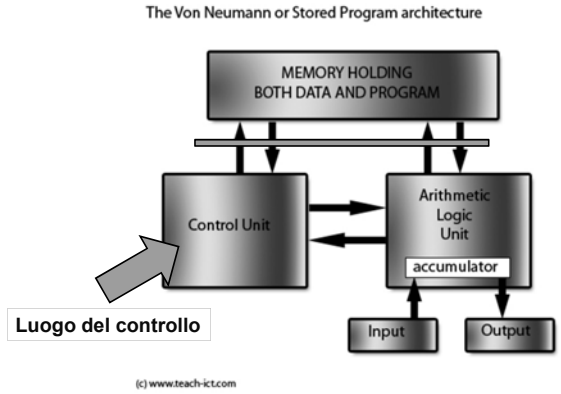
Premesse – 1

- L'espressività di un linguaggio ne è il potere ma anche il limite
  - Ciò che il linguaggio non sa dire, 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
  - Prevedono un unico luogo del controllo
- Questo riflette l'architettura 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

Architettura «von Neumann»



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 supporta più luoghi del controllo**
  - **Il compilatore crea un ambiente d'esecuzione (aka *runtime*) capace di creare e gestire le astrazioni di concorrenza offerte dal linguaggio**
  - ***Runtime* come macchina astratta**

Linguaggi concorrenti – 2

- **Il progetto di un linguaggio concorrente si deve ispirare a un modello di concorrenza di riferimento coerente e consistente**
  - **Più scelte possibili per astrazioni 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**
  - **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 espresso all'interno di un programma**
  - **Perché possiamo farlo?**
  - **Cosa definisce lo "stato" di un processo?**
  - **Perché ci importa saperlo definire?**
- **La modalità di esecuzione 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**

Concorrenza vs. parallelismo – 1

- **La concorrenza è problema di diverso ordine rispetto al parallelismo**

> **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**

Concorrenza vs. parallelismo – 2

- **Concorrenza**

> ***Concurrent programming allows simplifying the program by using multiple logical threads of control to reflect cohesively the natural concurrency structure in the solution***

  - **I costrutti di programmazione usati per ottenerla possono essere relativamente costosi perché il loro costo è (molto) inferiore al loro valore**

- **Parallelismo**

> ***Parallel programming allows using a divide-and-conquer style to solve a problem, with multiple threads that work in parallel on independent parts of the problem space***

  - **I costrutti di programmazione devono essere a basso costo perché sono usati molte volte per unità di lavoro potenzialmente molto piccole**

Osservazioni

- **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**

Precursori della concorrenza – 1

- ***Coroutine* : la storia**
  - **Una delle prime e più basiche 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**

Precursori della 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 **!**
  - **Dall'esecuzione di una stessa coroutine si può ritornare più volte**
    - Mentre da quella di 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 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

Alternative a confronto

- **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 – 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 è azione interrompibile in presenza di prerilascio**
- **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 distanza temporale che separa due successive invocazioni della stessa attività periodica
    - **Inevitabile**, può essere migliorata solo mediante maggiore accuratezza nell'espressione del tempo e controllo più fine e veloce dell'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

Drift locale

- The software clock resolution is an important RTOS design parameter
  - The finer the resolution the better the clock accuracy but the larger the time-service interrupt overhead
- There is delicate balance between the clock accuracy needed by the application and the clock resolution that can be afforded by the system
  - Latency is intrinsic in any query made by a task to the software clock
    - E.g., 439 clock cycles in ORK for the Leon microprocessor
- The resolution cannot be finer-grained than the maximum latency incurred in accessing the clock (!)