Programmazione parallela e concorrente

Jacob Sparre Andersen

JSA Research & Innovation

Ottobre 2017

Programmazione parallela e concorrente

Che cosa è, e come farlo con il supporto del linguaggio di programmazione.

Using Ada as the demonstration language, I will give a quick course in how to write concurrent and parallel programs. Both language constructs and conceptual issues will be covered.

Terminologia

- Programmazione parallela
- Programmazione concorrente

Terminologia: Programmazione parallela

Programmazione per un'**esecuzione effettivamente simultanea**.

(Singolo | Multipli) Istruzioni (Singolo | Multipli) Dati:

- SISD - come impara programmare
- SIMD - processori di vettore
- MIMD - processori multi-core
- MISD - il software per il "Space Shuttle" americano

Tipicamente visto come un modo per fare un programma correre più veloce:

$$t' = t \cdot (1 - p + \frac{p}{n})$$

Terminologia: Programmazione concorrente

Programmazione per un'esecuzione **logicamente simultanea**.

Un modo per astrarre considerazione su quanti core sono in realtà disponibili per l'esecuzione simultanea.

L'esecuzione attuale può essere sequenziale (per esempio su un computer single-core).

Questo è un estensione di come linguaggi di programmazione (tipicamente) astratti via inutili dettagli su i processori usati per i nostri programmi.

Concetti basali

- **Processo:** Una sequenza di istruzioni che vengono eseguite contemporaneamente ad altre sequenze di istruzioni.
- **Risorsa condivisa:** Qualsiasi tipo di risorsa (area di memoria, porta I/O, ecc.) che può essere utilizzato da più di un *processo*.
- **Esclusione reciproca:** Impedire l'accesso contemporaneamente a una risorsa da due processi.
- **Azione indivisibile:** Un'operazione su una *risorsa condivisa* che non può essere interrotto par altre operazioni sulla stessa risorsa.
- **Sincronizzazione:** Coordinamento dei processi per raggiungere un obiettivo comune.

Tipi di errori

- **Deadlock:** Due o più processi si bloccano a vicenda (tipicamente a causa di competizione per risorse condivise).
- **Starvation:** Rinvio indefinito di processi (tipicamente a causa di altri processi bloccando l'accesso ad un risorsa condivisa).
- **Race condition:** Quando il comportamento del programma dipende dal sequenza o temporizzazione degli eventi esterni (in modo non intenzionale).

Tipi di errori: Deadlock

Due processi (p_1 e p_2) e due risorse (r_1 e r_2):

Passo	p_1	p_2
1	Prende r_1.	Prende r_2.
2	Prende r_2.	Prende r_1.
3	Manipola r_1 e r_2.	Manipola r_1 e r_2.
3	Ritorna r_1.	Ritorna r_2.
4	Ritorna r_2.	Ritorna r_1.

Se i due processi completano passo 1 prima che ognuno di loro iniziano sul passo 2, saranno bloccati nel passo 2 perché l'altro processo ha il mancante risorsa.

Tipi di errori: Starvation

Rinvio indefinito di un processo.

Se un processo attende l'accesso a una risorsa condivisa e altri processi continua a bloccarlo, diciamo che il processo è **starved**.

Tipi di errori: Race condition

Generalmente si verificano *race conditions*, quando il programmatore presume una sequenza di operazioni essere atomico, mentre questo non è il caso della realtà.

```
if [ ! -d "${directory}" ]; then
    mkdir "${directory}"
fi
```

Esercizio:

Come può questo fallire?

task

In Ada un processo si chiama un “task”.

Una dichiarazione di un “task” banale:

```
task Multiply;
```

E un implementazione corrispondente:

```
task body Multiply is
begin
   Product := 1.0;
   for Element of Values loop
      Product := Product * Element;
   end loop;
end Multiply;
```

Creato in modo statico o dinamico

I “task” possono esistere come entità autonome o come tipi.

```
task Multiply;
```

```
task type Receiver;
```

You create an actual task from a task type either by declaring an object of the type, or by allocating a task object on the heap:

```
One : Receiver; --  object
```

```
Two := new Receiver; --  heap
```

```
procedure Vector_Math (Values  : in      Vector;
                       Sum     :    out  Scalar;
                       Product :    out  Scalar) is
   task Multiply;

   task body Multiply is
   begin
      Product := 1.0;
      for Element of Values loop
         Product := Product * Element;
      end loop;
   end Multiply;
begin
   Sum := 0.0;
   for Element of Values loop
      Sum := Sum + Element;
   end loop;
end Vector_Math;
```

Protected object

In Ada l'implementazione di esclusione reciproca è fatto con un "protected object".

Un "protected object" consiste di qualche **operazione**, e qualche **dati** privati.

To the extent that the operations only operate on the private data of the protected object, they are *atomic*.

Esempio (senza esclusione reciproca)

```
type Population_Size is range 0 .. 100;
--  Defined by fire regulations.

package Population is
   procedure Increment;
   procedure Decrement;
   function Current return Population_Size;
private
   Count : Population_Size := 0;
end Population;
```

Esercizio:
Che cosa può andare male, se più processi (“tasks”) accedono il pacchetto `Population`?

Esempio

Un esempio d'un "protected object" con propria esclusione reciproca:

```
type Population_Size is range 0 .. 100;
--  Defined by fire regulations.

protected Population is
   entry Increment;
   procedure Decrement;
   function Current return Population_Size;
private
   Count : Population_Size := 0;
end Population;
```

Spiegazioni

```
function Current return Population_Size;
```

Molti **functions** possono leggere i dati privati al stesso tempo, ma bloccano i **procedures** e **entries**.

```
procedure Decrement;
```

Un **procedure** ha accesso esclusivo di lettura/scrittura ai dati privati.

```
entry Increment;
```

Un **entry** ha accesso esclusivo di lettura/scrittura ai dati privati, male chiamate possono essere bloccate in base allo stato interno.

Esempio (implementazione)

```
protected body Population is
   entry Increment
     when Count < Population_Size'Last
   is
   begin
     Count := Count + 1;
   end Increment;

   procedure Decrement is
   begin
      Count := Count - 1;
   end Decrement;

   function Current return Population_Size is (Count);
end Population;
```

Deadlock: Aiuto pratico

Trovare i deadlock è un problema NP completo.

Ma un gruppo di ricerca a Vienna ha sviluppato una tecnica che utilizza Kronecker algebra per dimostrare l'assenza di deadlocks[1].

There can still be deadlocks hidden in dead code, which are not detected, but as the code will never be executed, it is not a problem in practice.

[1] L'unica cosa veramente interessante per noi sviluppatori.

Deadlocks: Un protocollo semplice

A basic procedure for avoiding potential deadlocks due to competition for shared resources is that all processes takes/locks the shared resources in the same order.

Swapping the order of taking the resources in one of the two processes in the deadlock example earlier would prevent a deadlock from ever occurring there.

Sincronizzazione

In Ada we can implement synchronisation between tasks either indirectly using *protected objects* or directly using a “rendezvous”.

The rendezvous is a client-server message passing construction:

- The server task declares a number of *entries*, which it may accept and execute at its leisure. (There is no guarantee that the server task will ever accept its published entries.)
- The client task can then call any of these entries (it looks like a plain procedure call).

Note that any task can function both as a client and a server.

Sincronizzazione (continuato)

If a server is ready to accept an entry, but no client has called it yet, the server task will be blocked (waiting) until somebody calls the entry[2].

If a client calls an entry, but the server isn't ready to accept it, then the client task will be blocked (waiting) until the server is ready to accept the entry call[3].

[2] We skip selective accepts for now.

[3] The client-side too has a bit more options than this.

Sincronizzazione (continuato)

Specification of a task with a single entry:

```
task Worker is
   entry Set_ID (Value : in     IDs);
end Worker;
```

Implementation of the same task:

```
task body Worker is
   ID : IDs;
begin
   accept Set_ID (Value : in     IDs) do
      ID := Value;
   end Set_ID;

   Do_Something (ID);
end Worker;
```

Sincronizzazione: Semaforo

Un “protected object” con un contatore e due operazioni:

- **Wait**: Attendere che il contatore sia maggiore di zero, quindi diminuisci il contatore.
- **Signal**: Incrementa il contatore.

Semaphores are used to provide mutual exclusion in cases where you can’t put the whole protected block into a *protected object*.

Sincronizzazione: Semaforo in Ada

The specification of a semaphore type in Ada:

```
package Semaphore is
   protected type Instance (Initial_Value : Natural) is
      entry Wait;
      procedure Signal;
   private
      Count : Natural := Initial_Value;
   end Instance;
end Semaphore;
```

Sincronizzazione: Semaforo in Ada

The implementation of the semaphore type:

```
package body Semaphore is
   protected body Instance is
      procedure Signal is
      begin
         Count := Count + 1;
      end Signal;

      entry Wait when Count > 0 is
      begin
         Count := Count - 1;
      end Wait;
   end Instance;
end Semaphore;
```

Sincronizzazione: Uso del semaforo

```
Lock : Semaphore.Instance (Initial_Value => 1);

procedure Put_Line (Item : in      String) is
begin
   Lock.Wait;
   Ada.Text_IO.Put_Line (Item);
   Lock.Signal;
end Put_Line;
```

The procedure `Ada.Text_IO.Put_Line` is potentially blocking, so we are not allowed to call it from inside a protected object. Instead we use the semaphore (`Lock`) to ensure that only one task at a time is calling `Ada.Text_IO.Put_Line` (through this package).

Synchronisation: Finalised lock (specification)

```
with Ada.Finalization;

generic
package Finalized_Lock is
   --  An object of this type will automatically wait on
   --  the semaphore declared inside the package when it
   --  is created, and signal the semaphore when it goes
   --  out of scope.
   --
   type Instance is new Ada.Finalization.
      Limited_Controlled with null record;
private
   overriding
   procedure Initialize (Item : in out Instance);
   overriding
   procedure Finalize (Item : in out Instance);
end Finalized_Lock;
```

Synchronisation: Finalised lock (implementation)

```
package body Finalized_Lock is
   Lock : Semaphore.Instance (Initial_Value => 1);

   overriding
   procedure Initialize (Item : in out Instance) is
      pragma Unreferenced (Item);
   begin
      Lock.Wait;
   end Initialize;

   overriding
   procedure Finalize (Item : in out Instance) is
      pragma Unreferenced (Item);
   begin
      Lock.Signal;
   end Finalize;
end Finalized_Lock;
```

Synchronisation: Finalised lock (use)

```
task body GULCh is
   Key : Lock.Instance with Unreferenced;
begin
   Ada.Text_IO.Put ("Linux ");
   delay 0.2;
   Ada.Text_IO.Put ("Day ");
   delay 0.2;
   Ada.Text_IO.Put ("2017 ");
   delay 0.2;
   Ada.Text_IO.Put ("a ");
   delay 0.2;
   Ada.Text_IO.Put_Line ("Cagliari");
end GULCh;
```

Sincronizzazione: Futures

Futures is a popular concept in non-concurrent programming languages.

Basically your apparently non-concurrent program hands off some processing to an invisible process to get the result back sometime in the future.

The equivalent Ada construct is to use concurrency explicitly, and exchange results using a protected object or a direct rendezvous.

Synchronisation: Interrupts

Interrupts represent events detected by the hardware.

In Ada you use protected procedures as interrupt handlers:

```
protected Converter is
   entry Read (Value :      out Voltage);
private
   procedure Handler;
   pragma Attach_Handler (Handler, SIGINT);

   entry Wait_For_Completion (Value :      out Voltage);

   Busy                : Boolean := False;
   Conversion_Complete : Boolean := False;
end Converter;
```

Tips and tricks

Some Ada specific tips for parallel and concurrent programming.

Task attributes

The package `Ada.Task_Attributes` allows the developer to attach an attribute (i.e. a variable) to all tasks in a system.

This can for example be useful if you want to let each task have its own database connection, but you don't want to make that visible in the source text for individual tasks.

Task termination

The package `Ada.Task_Termination` allows the developer to attach a termination handler to all dependent tasks (or to a specific task).

In one application I work on, we are using it to ensure that it is logged if a task terminates in an unexpected manner.

CPU

Ada defines the aspect `CPU` for tasks and task types.

You can use it to assign specific tasks to run on specific CPU's.
The problem with this is that you end up hard-coding hardware architecture in the source code, thus writing concrete, parallel programs instead of abstracted, concurrent programs.
But there are certainly cases where it is the correct approach.

Bibliografia

Se vuoi saperne di più, suggerisco il libro “Building Parallel, Embedded, and Real-Time Applications with Ada” [1].

Alcuni degli esempi in questa presentazione sono stati ispirati da esso.


John W. McCormick, Frank Singhoff, and Jérôme Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada*.
Cambridge, 2011.

Informazioni di contatto

Jacob Sparre Andersen
JSA Research & Innovation
jacob@jacob-sparre.dk
http://www.jacob-sparre.dk/

Esempi:

http://www.jacob-sparre.dk/programming/parallel_programming/linux-day-2017-examples.zip