# Corso di Linguaggi di Programmazione

## Lezione 23

Alberto Ceselli
alberto.ceselli@unimi.it

Dipartimento di Informatica
Università degli Studi di Milano

28 Maggio 2013

# Sincronizzazione in ambiente distribuito

- Monitor funzionano bene se ho *memoria condivisa* (code ecc.)
- In ambiente distribuito, i monitor si adattano male
- è invece naturale progettare un meccanismo basato su *scambio di messaggi*

# Guarded commands

- Proposito 'storico' (Dijkstra '75): programmazione che assicuri correttezza durante lo sviluppo anziché ricorrere a testing.
- Es: costrutto di selezione:

```
if <espr. booleana> -> <istruzione>
[] <espr. booleana> -> <istruzione>
[] ...
[] <espr. booleana> -> <istruzione>
fi
```

- [] è detto 'fatbar' e permette la concatenazione delle clausole
- semanticamente diverso dalla selezione multipla (case): *tutte* le espressioni booleane sono valutate simultaneamente durante l'esecuzione
- se più di una è vera, ne viene scelta una in modo *non-deterministico*, se *nessuna* è vera, c' è errore a runtime ed il programma è terminato

# Nota

- in modo analogo, possono essere definiti *cicli*

```
do <espr. booleana> -> <istruzione>
[] <espr. booleana> -> <istruzione>
[] ...
[] <espr. booleana> -> <istruzione>
od
```

- La verifica della correttezza di un programma è più semplice se si utilizzano solo cicli e selezione, oppure solo guarded commands.

# Scambio di messaggi sincrono

- *"... tasks are often busy and, when busy, are unwilling to be interrupted by other units."* (R. Sebesta)
- Ci sono due tipi di costrutti linguistici:
  - costrutti per l'*invio* di messaggi
  - costrutti per specificare che il ricevente è in *attesa* di ricevere messaggi
- lo scambio effettivo di informazioni è detto **rendezvous**, e può avvenire anche bidirezionalmente.

# Esempio:

Tasks in ADA (`main_tasks.adb`)
Actor e Server in ADA (`consumerpackage, producerpackage`)

# Accept con guardia

- Ogni `accept` può avere una clausola `when` associata (guardia).

```
when not Full ( Buffer ) =>
   accept Deposit ( New_Value ) do
```

- `accept` con `when` può essere **aperta** (se la condizione è vera) o **chiusa** (se la condizione è falsa)

# Cooperation synchronization

- cooperation synchronization: può essere ottenuta combinando `select` e `accept` con guardia, ed utilizzando un task per la sincronizzazione: es put() e get()

```
loop
   select
      when Filled < Bufsize =>
         accept Put( v : in Integer) do
            ...
         end Put;
   or
      when Filled > 0 =>
         accept Get( v : out Integer) do
            ...
         end Get;
   end select;
end loop;
```

# Competition synchronization

- competition synchronization:può essere ottenuta incapsulando le strutture dati a cui garantire l'accesso esclusivo all'interno di un task 'server':

```
task Buffer_Server is
   entry Put( v : in Integer);
   entry Get( v : out Integer );
end Buffer_Server;
task body Buffer_Server is
   size : constant Integer := 100;
   buffer : array (1 .. size) of Integer;
   filled : Integer range 0 .. size := 0;
   begin
      ...
   end Buffer_Server;
```

# Esempio

Produttore e consumatore in ADA (Sebesta pagg. 520 – 521)
`ada_prod_cons`

# Valutazione

- In ambiente distribuito, i monitor non possono essere gestiti in modo naturale, lo scambio di messaggi sì
- In ambiente non distribuito, lo scambio di messaggi è a metà strada tra monitor e semafori:
  - safe per competition synchronization
  - unsafe per cooperation synchronization (ma meno dipendente dal corretto utilizzo)

# Scambio di messaggi asincrono

Buona programmazione concorrente significa anche sfruttare il parallelismo:

- Caso semplice di asincronia: accesso a procedure o funzioni non sincronizzate
- Altro caso di asincronia: un task lancia un altro task, che svolge un particolare calcolo indipendentemente
- Infine: l'esecuzione di un task può essere interrotta da un evento: **asynchronous select** in ADA
  Esempio pag. 524 Sebesta

# Semafori

- I task ADA possono essere utilizzati per implementare semafori ...
- Esercizio: implementare i semafori in ADA

# Oggetti protetti

- ADA ha un altro costrutto per gestire la concorrenza: gli oggetti `protected`
- ogni oggetto `protected` ha variabili, funzioni e procedure
- fornisce accesso *mutuamente esclusivo* alle procedure, che hanno la possibilità di effettuare letture e scritture sulle variabili dell'oggetto
- fornisce accesso *concorrente* alle funzioni, che hanno la possibilità di accedere solo in lettura alle variabili dell'oggetto
- Esempio: pag 523 Sebesta

# Priorità e terminazione

- Un task è *completato* quando il suo thread di controllo ha raggiunto la fine del corpo del codice
- I task ADA *terminano*
  - Quando sono completati, e tutti i sotto-task (i task *dipendenti*) sono terminati
  - Aspettando ad una clausola `terminate`
- è possibile associare una priorità ad ogni task, utilizzando la direttiva `pragma`:

```
pragma Priority(<espressione>);  -- ad esempio
pragma Priority(System.Priority'Last);
```

# Esercizi:

- Implementare un buffer condiviso, utilizzando monitor e garantendo cooperation e competition synchronization (pseudo-codice)
  - codice monitor ("server")
  - codice produttore
  - codice consumatore
- implementare i semafori in ADA

# Unità concorrenti e sincronizzazione in Java

VEDI Lez. 22

# Valutazione

- Semplice, ma efficace.
- Thread (lightweight) rispetto ai Task (heavyweight) di ADA: non adatti ad ambiente distribuito (librerie!)
- Si possono simulare monitor e semafori

# Unità concorrenti in C#

- Le unità concorrenti in C# possono essere metodi qualsiasi
- Le entità in cui girano i metodi concorrenti sono dei *thread*
- Creazione di un thread:

```
public void any_method() { ... }
...
Thread myThread = new Thread( new ThreadStart( any_method ) ); // delegate
...
myThread.start();
```

- `ThreadStart` è una classe *delegate* (funzione dei puntatori a procedure in C).
- non solleva nessuna eccezione (niente blocco `try`)

# Sincronizzazione

Ci sono vari modi per sincronizzare thread C#

- classe `Interlocked`: operazioni *atomiche*

  ```
  Interlocked.Increment(ref value); Interlocked.Decrement(ref value);
  ```

- istruzione `lock`: mutua esclusione

  ```
  lock(<espressione>) { <sezione critica> ... }
  ```

- classe `Monitor`: ha 4 metodi (`Enter`, `Wait`, `Pulse`, `Exit`)
  - `Monitor.Enter(<oggetto>)`: acquisisce i lock sull'oggetto
  - `Monitor.Wait()`: analogo a `wait()` Java
  - `Monitor.Pulse(<oggetto>)`: analogo a `notifyAll()` Java
  - `Monitor.Exit(<oggetto>)`: rilascia il lock sull'oggetto

# Valutazione

- Molto simile a Java.
- Ogni metodo può acquisire un thread di controllo
- più costrutti per la sincronizzazione

# Il modulo threading

- Il supporto alla programmazione concorrente in Python è fornito nel modulo `threading`

```
...
import queue
from threading import Thread
...
```

- threading: Thread, Lock, Semaphore, Condition
- esistono classi già costruite per molti problemi 'fondamentali' (Es. queue)
- NB: l'interprete Python *non è completamente thread-safe* (esiste un Global Interpreter Lock — GIL) che deve essere acquisito.

# Thread

... estendono la classe Thread, sovrascrivendo il metodo `run`

```
def main():
    for i in range(3):
        my_thread = MyThread(i)
my_thread.daemon = True
        my_thread.start()

class MyThread(Thread):

    def __init__(self, i):
        Thread.__init__(self)
        self.i = i

    def run(self):
        while True:
...
```

# Lock

Simili a Queue in Concurrent Pascal:

```
from threading import Thread, Lock
class Buffer(Thread):

    def __init__(self, i):
        self.lock = Lock()
        self.storage = 0

    def write(self, v):
        self.lock.acquire()
        self.storage = v
        self.lock.release()
```

# Semaphore

`acquire()` equivale a P(), e `release()` equivale a V()

```
semaphore = threading.BoundedSemaphore(3)
semaphore.acquire() # decrements the counter
... access the shared resource
semaphore.release() # increments the counter
```

# Event

event gestisce un flag booleano, che può essere modificato con `set()` e `get()`. Se il flag è `false`, `wait()` interrompe il thread corrente, ponendolo in stato waiting; `set()` risveglia eventuali thread waiting.

```
event = threading.Event()

# a client thread can wait for the flag to be set
event.wait()

# a server thread can set or reset it
event.set()
event.clear()
```

# Condition

Simile a `Event`, ma supporta operazioni `notify()` e `notifyAll()`

```
condition = threading.Condition()
... generate item
condition.acquire()
if (full) condition.wait()
... add item to resource
condition.notify() # signal that a new item is available
condition.release()
```

e diverse condition possono condividere lo stesso `lock`

```
lock = threading.Lock()
condition_1 = threading.Condition(lock)
condition_2 = threading.Condition(lock)
```

# High Performance Fortran

- Collezione di estensioni al Fortran 90 per permettere ai *programmatori* di fornire istruzioni al *compilatore* per generare codice ottimizzato su piattaforme multiprocessore
- Istruzioni di **specifica**, es. numero di processori o distribuzione dati ai processori

```
!HPF$ PROCESSORS procs (<n>)
!HPH$ DISTRIBUTE (<tipo_distr>) ONTO procs :: <lista id_variabili>
```

- `tipo_distr` può essere `BLOCK` oppure `CYCLIC`
- N.B. `!` introduce *commenti* in Fortran: facile da gestire per un compilatore / preprocessore.

# High Performance Fortran

- Istruzioni di **controllo**, es.

```
FORALL (index = 1:1000) list_1 (index) = list_2 (index)
```

  suggerisce di parallelizzare rispetto agli indici degli array
- Esistono anche librerie C / C++ che utilizzano la stessa filosofia: es. **openMP**.

# Esercizi:

- Implementare una simulazione della Cena dei Filosofi utilizzando:
  - semafori
  - monitor
  - scambio di messaggi
  - Java threads / C# threads

# Il concetto di eccezione

A *run time* si possono verificare

- eventi anomali o eccezionali
- individuabili via hardware o software
- che richiedono una gestione appropriata
- spesso richiedendo variazioni nel flusso di controllo principale del programma

... ovvero *eccezioni* che devono essere *gestite*.

# Input Output

**Esempio (individuato da hardware).**

- contesto: Input da file
- evento (eccezione): device failure
- gestione: messaggio a video e terminazione del programma

Fortran:

```
Read(Unit=5, Fmt=1000, Err=100, End=999) Weight
```

Se si verifica device failure `goto 100`, se si verifica EOF `goto 999`

# Array index out of bounds

**Esempio (individuato da software).**

- contesto: accesso a componenti di array
- evento (eccezione): indice fuori dai limiti
- gestione: messaggio a video e nessuna operazione.

Java:

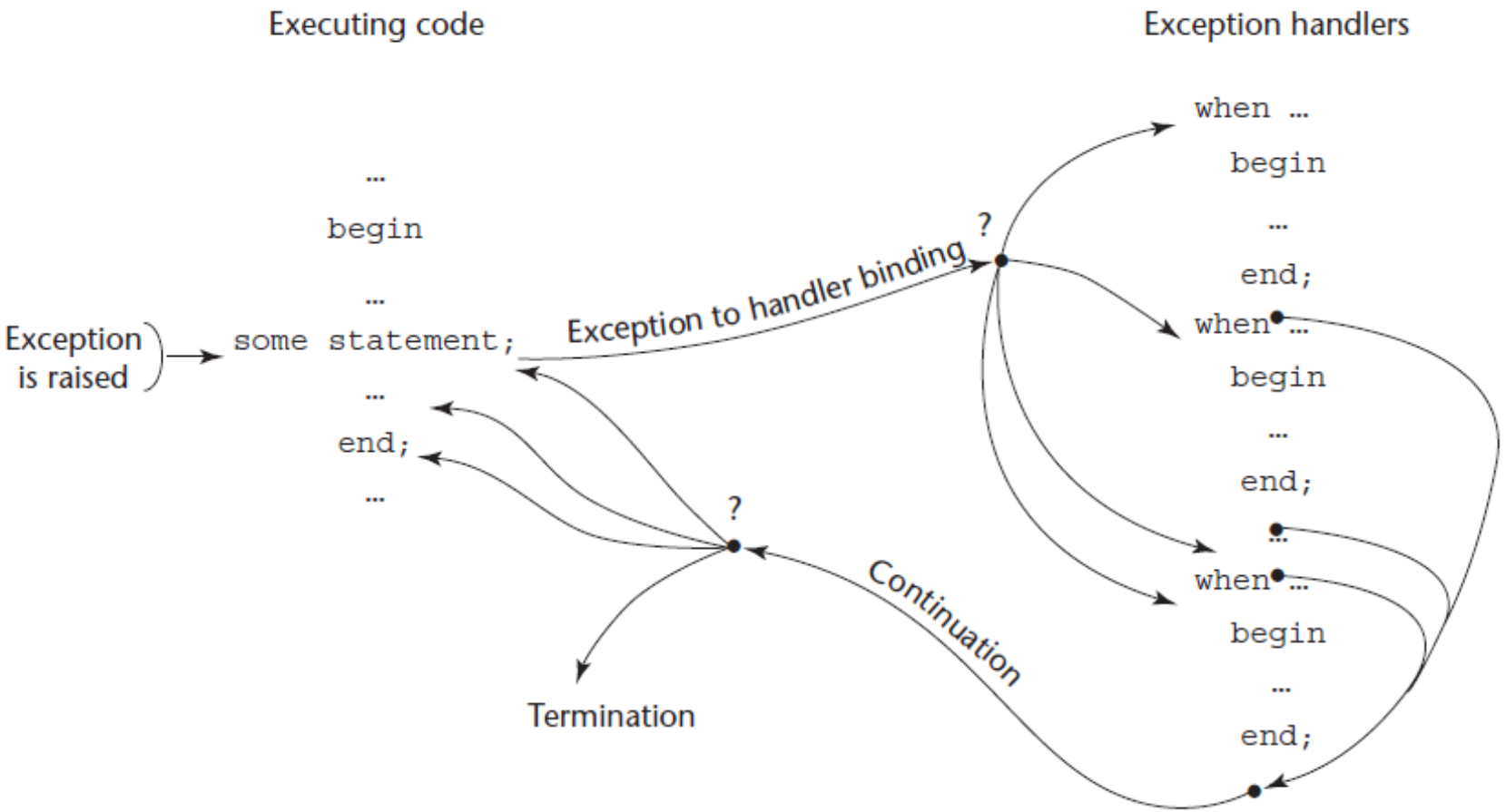
```
for (i = 0; i<10; i++) {
   ...
   try {
       v[i] = 100;
   } catch( ArrayIndexOutOfBoundsException e ) {
       System.out.println( "Invalid Index " + i);
   }
   ...
}
```

# Nomenclatura

Formalmente:

- elaborazione particolare a fronte di eccezione: **exception handling**
- codice associato: **exception handler**
- ... quando l'evento associato occorre, l'eccezione è **sollevata** (raised o *thrown*)

# Continuation, termination, resumption e finalization



# Handlers

```
try{
  // codice che potrebbe sollevare l'eccezione
} catch (parametro1) {
  // HANDLER 1 (Codice C++)
} catch (parametro2) {
  // HANDLER 2 (Codice C++)
} catch (...) {
  // HANDLER generico
}
```

# Binding di Eccezioni ad Handlers

```
throw [espressione di tipo T];
```

- avviene binding al *primo handler* il cui *tipo* è T (o const T o T& o const T&) o una sua *superclasse*
- se non ci sono handlers opportuni, l'eccezione viene *propagata* (al chiamante).

# Continuation e design issues

- Continuation: dopo che l'handler ha completato la propria esecuzione, il controllo *passa alla prima istruzione dopo il blocco try-catch*.
- In C++ ci sono *solo eccezioni user-defined*.
- Le eccezioni non possono essere disabilitate.
- Una funzione C++ può elencare le eccezioni che propaga: `int fun() throw (int, char *) ...`
- Un handler può risollevare un'eccezione con `throw` senza argomenti.

# Le eccezioni in Java sono oggetti

Le Eccezioni sono discendenti di `Throwable`.

- `Error` per errori individuati da JVM (es. out of memory)
- `Exception` usato come "capostipite" delle eccezioni user-defined.

# Handlers

```
try{
  // codice che potrebbe sollevare l'eccezione
} catch (parametro1) {
  // HANDLER 1 (Codice Java)
} catch (parametro2) {
  // HANDLER 2 (Codice Java)
...
} finally { // (*) }
```

`finally` eseguito

- dopo `try` se non si verificano eccezioni
- dopo handler se l'eccezione è gestita
- prima di propagare se l'eccezione non è gestita

# Binding di Eccezioni ad Handlers

```
class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) { super(msg); }
}
...
MyException e = new MyException();
throw e;
throw new MyException("in a compact way");
```

- avviene binding al *primo handler* il cui *tipo* è T o una sua *superclasse*
- se non ci sono handlers opportuni, l'eccezione viene *propagata* (al chiamante).

# L'idea della programmazione ad eventi

In estrema sintesi, un programma ad eventi è

- un main che si occupa solo della prima inizializzazione
- una serie di handler, uno per ogni possibile evento che si può verificare durante l'esecuzione

quindi l'esecuzione del programma è essenzialmente una iterativa gestione (**event handling**) di eccezioni (**event**).

# Esempio

Tipico esempio: codice associato a web form o GUI

- il main fa partire la GUI
- ad ogni azione dell'utente è associato un evento
- ad ogni evento è associato un handler

# Java Swing

Esempio di supporto alla programmazione ad eventi: Java Swing

```
ButtonGroup payment = new ButtonGroup();
JRadioButton box1 = new JRadioButton("Visa", true);
JRadioButton box2 = new JRadioButton("Master C", false);
JRadioButton box3 = new JRadioButton("Am Expr", false);
payment.add(box1);
payment.add(box2);
payment.add(box3);
```

# Java Swing: creazione della GUI

```
public class RdB extends JApplet implements ItemListener {
   private Container contentPane = getContentPane();
   ...
   private JPanel myPanel = new JPanel();
   ...
   public void init() {
      mypanel.setBackground(Color.cyan);
      ...
      mypanel.add(payment);
      ...
      contentPane.add(myPanel);
      ...
   }
}
```

# Java Swing: event handling

```
public class RdB extends JApplet implements ItemListener {
   ...
   public void init() {
      ...
      box1.addItemListener(this);
      box2.addItemListener(this);
      box3.addItemListener(this);
      ...
   }
   public void itemStateChanged (ItemEvent e) {
      if (box1.isSelected()) ...
      else if (box2.isSelected() ) ...
      else if (box3.isSelected() ) ...
   }
```