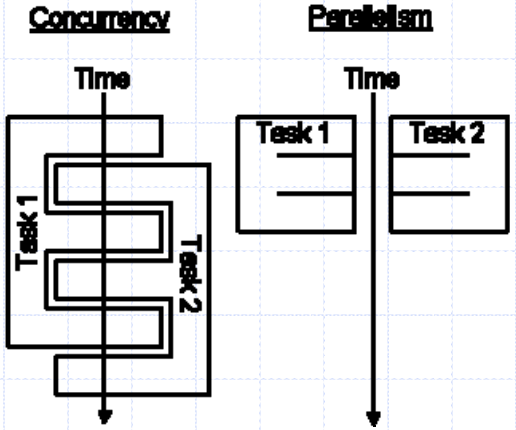
Programmazione concorrente: Java e Ada

Paolo Costa

paolo.costa@polimi.it

Concurrency

- Concurrency is an important area of computer science studied in different contexts: machine architectures, operating systems, distributed systems, database, etc
- Objects provide a way to divide a program into independent sections. Often, you also need to turn a program into separate, independently running subtasks
- Concurrency may be *physical* (parallelism) if each unit is executed on a dedicated processor or *logical* if the CPU is able to switch from one to another so that all units appear to progress simultaneously



Process & Thread

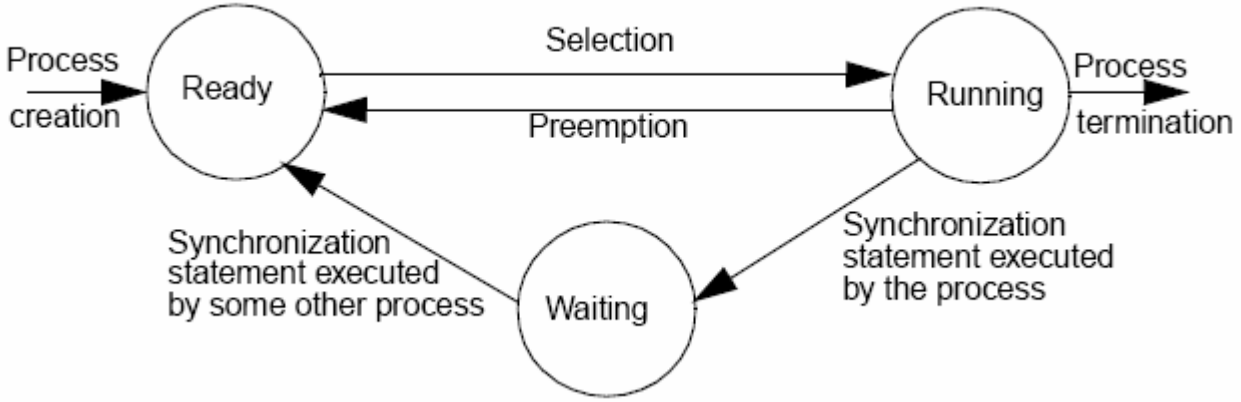
- A *process* is a self-contained running program with its own address space
- A *multitasking* operating system is capable of running more than one process at a time by periodically switching the CPU from one task to another
- The term *thread* (a.k.a. lightweight process) instead is used when the concurrent units share a single address space

Cooperation

- If the abstract machine does not support concurrency, it can be simulated by transferring control explicitly from one unit to another (*coroutines*) according to a cooperative model
- Runtime support is simpler to implement but it is more error prone:
  - the programmer has to handle cooperation: bugs in code may lock the systems

Preemption

- Modern systems adopts a *preemptive* model
- Preemption is an action performed by the underlying implementation: it forces a process to abandon its running state even if it could safely execute
- Usually a time slicing mechanism is employed where a process is suspend when a specified amount of time has expired



Concurrency in Java

- Java supports concurrency at *language level* rather than through run time libraries (like POSIX threads for C/C++)
- It employs a *preemptive* model
- If time-slicing is available (implementation dependent), Java ensures equal priority threads execute in round-robin fashion otherwise they runs to completion
- Priority is handled differently according to the specific implementation

Thread Class

- The simplest way to create a thread is to inherit from **`java.lang.Thread`**, which has all the wiring necessary to create and run threads. The most important method for **`Thread`** is **`run()`**

The thread object is defined by extending **Thread** class

You must override **run()**, to make the thread do your bidding.

Thread object is instantiated as usual through a **new()**

To run the thread you must invoke the **start()** method on it

```
class MyThread extends Thread {
  private String message;
  public MyThread(String m) {message = m;}
  public void run() {
    for(int r=0; r<20; r++)
      System.out.println(message);
  }
}

public class ProvaThread {
  public static void main(String[] args) {
    MyThread t1,t2;
    t1=new MyThread("primo thread");
    t2=new MyThread("secondo thread");
    t1.start();
    t2.start();
  }
}
```

Runnable Interface

- You can use the alternative approach of implementing the **Runnable** interface. **Runnable** specifies only that there be a **run()** method implemented, and **Thread** also implements **Runnable**

Your class must implement **Runnable** interfaces

run() must be overridden

To produce a thread from a **Runnable** object, you must create a separate Thread object

start() method is invoked to execute the thread

```
class MyThread implements Runnable {
  private String message;
  public MyThread(String m) {message = m;}
  public void run() {
    for(int r=0; r<20; r++)
      System.out.println(message);
  }
}

public class ProvaThread {
  public static void main(String[] args) {
    Thread t1, t2;
    MyThread r1, r2;
    r1 = new MyThread("primo thread");
    r2 = new MyThread("secondo thread");
    t1 = new Thread(r1);
    t2 = new Thread(r2);
    t1.start();
    t2.start();
  }
}
```

sleep()

- Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- The thread does not lose ownership of any monitors (see next).
- Useful for:
  - Periodic actions
    - Need to wait for some period of time before doing the action again.
  - Allow other threads to run

join()

- The **`join()`** method used to wait until thread is done.
- The caller of **`join()`** blocks until thread finishes.
- Other versions of join take in a timeout value.
  - If thread doesn't finish before timeout, join returns.
- Alternatively, can check on a thread with **`isAlive:`**
  - returns true if running, false if finished.

Non-determinism

- Threads execution proceeds without a predefined order
- The same code, if run on different computers, could produce different output
  - it depends on how internal scheduling is performed, on processor features, ...
- Such behavior is define as *non-deterministic*
- Non-determinism is a key point in concurrency
  - it is what makes it so hard to handle

Correctness

- A concurrent system is correct if and only if it owns the following properties:
  - *Safety*: bad things do not happen
  - *Liveness*: good things eventually happen
- Safety failures lead to unintended behavior at run time — *things just start going wrong*
  - Read / write conflicts
  - Write / write conflicts
- Liveness failures lead to no behavior — *things just stop running*
  - locking
  - waiting
  - I/O
  - CPU contention
  - failure
- Sadly enough, some of the easiest things you can do to improve liveness properties can destroy safety properties, and vice versa (e.g. locking)

Synchronization

```
public class RGBColor {
    private int r;
    private int g;
    private int b;

    public void setColor(int r, int g, int b) {
        checkRGBVals(r, g, b);
        this.r = r;
        this.g = g;
        this.b = b;
    }
```

- Imagine you have two threads, one thread named "red" and another named "blue". Both threads are trying to set their color of the same **`RGBColor`** object
- If the thread scheduler interleaves these two threads in just the right way, the two threads will inadvertently interfere with each other, yielding a write/write conflict. In the process, the two threads will corrupt the object's state

from *http://www.javaworld.com/jw-08-1998/jw-08-techniques-p2.html*

Synchronization

- Typical issue in concurrent programming

```
class Even {
 private int n = 0;
 public int next(){
   // POST?: next is     always even
   ++n;
   ++n;
   return n;
  }
 }
```

If multiple threads access Even, post-conditions may not be kept

Objects and Locks

- Every instance of class **`Object`** and its subclasses possess a lock
- Scalar fields can be locked only via their enclosing objects
- Locking may be applied only to the use of fields within methods
- Locking an array of **`Object`** does not automatically lock all its elements

Synchronized Blocks and Methods

- Block synchronization takes an argument of which object to lock

```
synchronized (object){
    // Lock is held
    …
}
// Lock is released
```

- Declaring a method as **synchronized** would preclude conflicting traces. Locking serializes the execution of **synchronized** methods

```
synchronized void f() { /* body */ }
```

is equivalent to

```
void f() { synchronized(this) { /* body */ } }
```

Acquiring Locks

- A lock is acquired *automatically* on entry to a **`synchronized`** method or block, and released on exit, even if the exit occurs due to an exception
- Locks operate on a per-thread, not per-invocation basis. A thread hitting synchronized passes if the lock is free or the thread already possess the lock, otherwise it blocks
- A synchronized method or block obeys the acquire-release protocol only with respect to other synchronized methods and blocks on the same target object
  - *methods that are not synchronized may still execute at any time, even if a synchronized method is in progress*
- Synchronized is not equivalent to atomic, but synchronization can be used to achieve atomicity

`synchronized` & OO

- The **`synchronized`** keyword is not considered to be part of a method's signature:
  - the **`synchronized`** modifier is *not* automatically inherited when subclasses override superclass methods
  - methods in interfaces cannot be declared as **`synchronized`**
- Synchronization in an inner class method is independent of its outer class
  - however, a non-static inner class method can lock its containing class, say **`OuterClass`**, via code blocks using:

```
synchronized(OuterClass.this) { /* body */ }
```

- Static synchronization employs the lock possessed by the **`Class`** object associated with the class the static methods are declared in.
  - The static lock for class `C` can also be accessed inside instance methods via:

```
synchronized(C.class) { /* body */ }
```

Key Rules

I. Always lock during updates to object fields

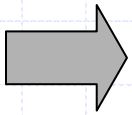
```
synchronized (point) {
   point.x = 5;
   point.y = 7; }
```

II. Always lock during access of possibly updated object fields

```
synchronized(point) {
   if(point.x > 0) {
            ...
}}
```

III. You do not need to synchronize stateless parts of methods

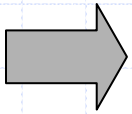
```
public synchronized void service(){
  state = ...; // update state
  operation();
 }
```



```
public void service(){
  synchronized(this) {
    state = ...; // update state
  }
  operation();
}
```

IV. Never lock when invoking methods on other objects

```
public synchronized void service(){
  ...
  h.foo();
 }
```



```
public void service(){
  synchronized(this) {
    state = ...; // update state
  }
  operation();}
```

Deadlock

- Deadlock is possible when two or more objects are mutually accessible from two or more threads, and each thread holds one lock while trying to obtain another lock already held by another thread
- Although fully synchronized atomic objects are always safe, they may lead to deadlock

Tema d'esame 6 maggio 2005

- Consider the following java classes:

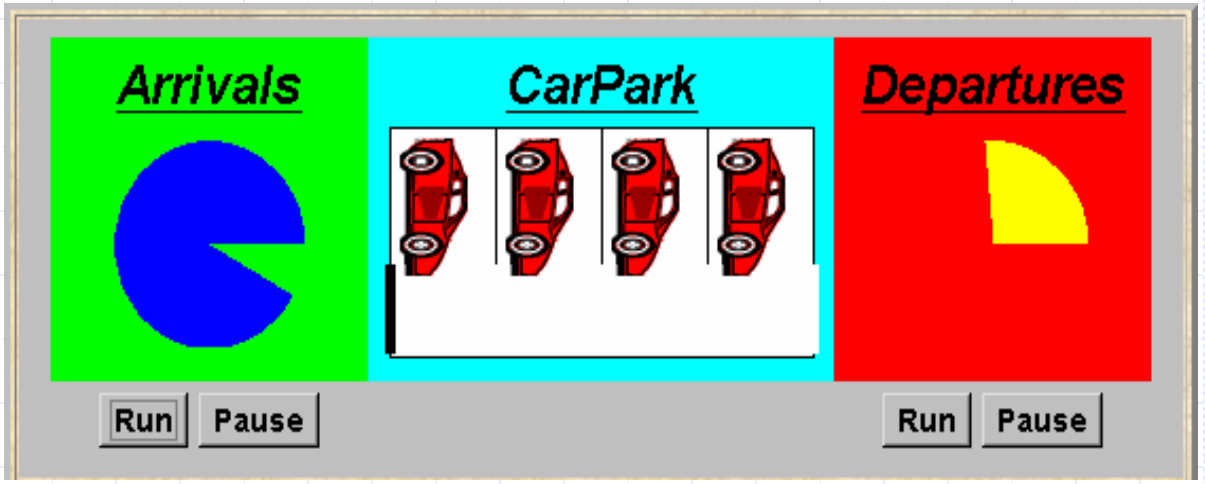
```
class A {

   ...
   synchronized public void ma1(B b)
{
      ...
      b.mb2();
   }

   synchronized public void ma2() {
      ...
   }
 }
```

```
class B {

   ...
   synchronized public void mb1(A a)
{
      ...
      a.ma2();
   }

   synchronized public void mb2() {
      ...
   }
 }
```

- How deadlock can arise if two thread, say X and Y, concurrently access to A and B respectively ?
- How can you effectively solve the issue ?

Condition Synchronization

from *http://www-dse.doc.ic.ac.uk/concurrency/book_applets/CarPark.html*



- A controller is required for a carpark, which only permits cars to enter when the carpark is not full and does not permit cars to leave when there are no cars in the carpark
- Car arrival and departure are simulated by separate threads

CarParkControl Monitor

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() {
    …  --spaces; …
    }

  synchronized void depart() {
    … ++spaces; …
    }
}
```

mutual exclusion by synch methods

condition synchronization ?

block if full? (spaces==0)

block if empty? (spaces==N)

Condition Synchronization in Java

- Java provides a thread wait set per monitor (actually per object) with the following methods:

`public final void notify()`

Wakes up a single thread that is waiting on this object's set

`public final void notifyAll()`

Wakes up all threads that are waiting on this object's set

`public final void wait()`
`throws InterruptedException`

Waits to be notified by another thread. The waiting threa releases the synchronization lock associated with the monitor. When notified, the thread must wait to reacquire the monitor before resuming execution

Condition Synchronization

```
when cond act -> NEWSTAT
```

```
Java:  public synchronized void act()
          throws InterruptedException {
       while (!cond) wait();
        // modify monitor data
        notifyAll()
       }
```

- The **while** loop is necessary to retest the condition **cond** to ensure that **cond** is indeed satisfied when it re-enters the monitor
- **notifyAll** is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed

Condition Synchronization

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait();
    --spaces;
    notify();
  }

  synchronized void depart() throws InterruptedException {
    while (spaces==capacity) wait();
    ++spaces;
    notify();
  }
}
```

Why is it safe to use `notify()` *here rather than* `notifyAll()`*?*

Single Notifications

- You can reduce the context-switch overhead associated with notifications by using a single **`notify`** rather than **`notifyAll`**
- Single notifications can be used to improve performance when you are sure that at most one thread needs to be woken. This applies when:
  - all possible waiting threads are necessarily waiting for conditions relying on the same notifications, usually the exact same condition
  - each notification intrinsically enables at most a single thread to continue. Thus it would be useless to wake up others

Summary

- Each guarded action in the model of a monitor is implemented as a synchronized method which uses a while loop and wait() to implement the guard
- Changes in the state of the monitor are signaled to waiting threads using **`notify()`** or **`notifyAll()`**
- The monitor is referred to the object instance which is used to communicate among Threads
- The lock must always be acquired before wait is invoked:

```
syncronized(lock) {
        ...
        lock.wait();
}
```

- Always re-check condition, after a wait:

```
syncronized(lock) {
  while(!cond) {
    lock.wait();
  }
}
```

YES

```
syncronized(lock) {
  if(!cond) {
    lock.wait();
  }
}
```

NO

Busy Waits

- Implementing guards via waiting and notification methods is nearly always superior to using an optimistic-retry-style busy-wait "spinloop" of the form:

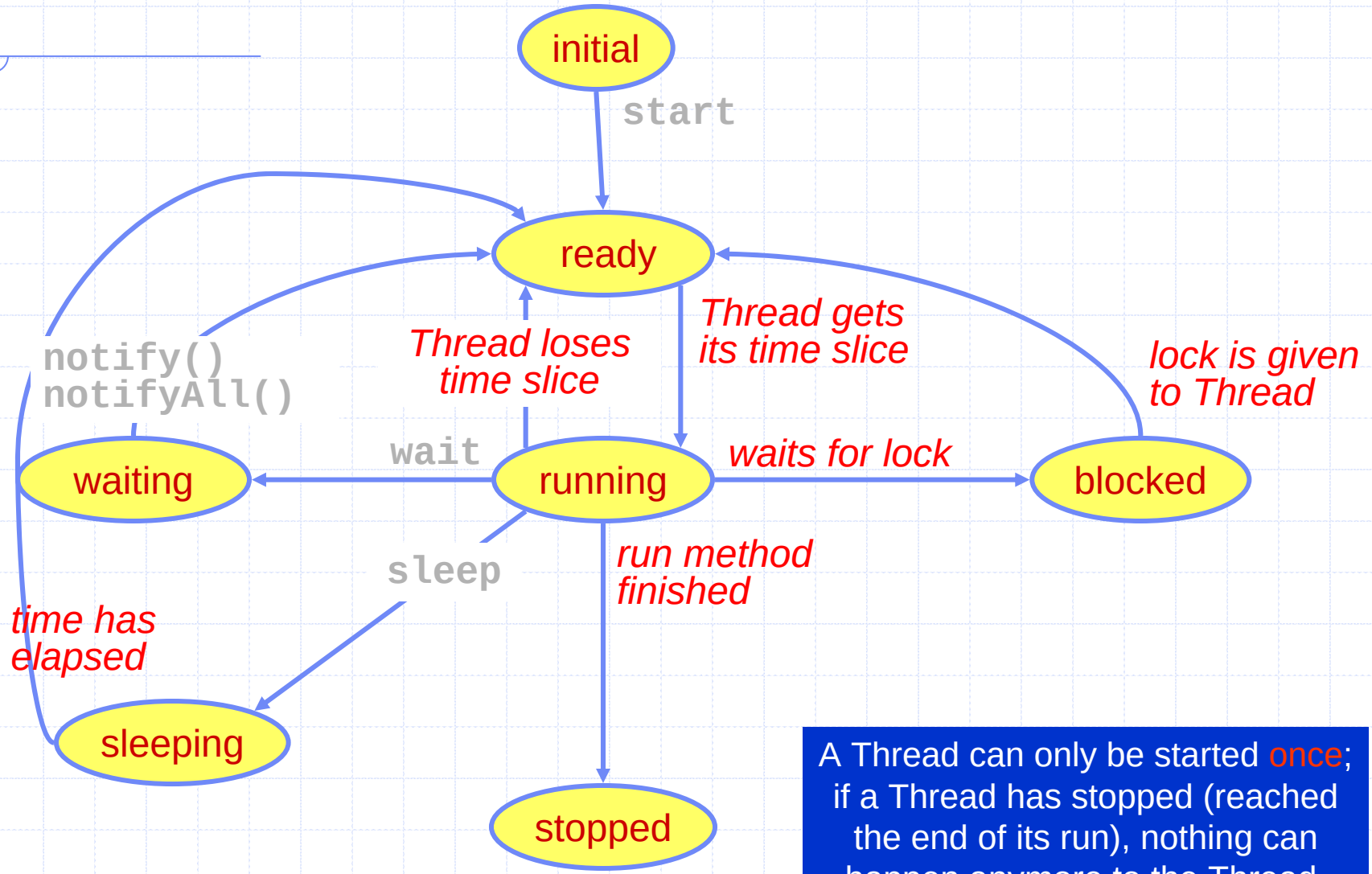
```
protected void busyWaitUntilCond() {
  while (!cond)
      Thread.yield(); }
```

- Reasons are:
  - Efficiency
  - Scheduling
  - Triggering
  - Synchronizing actions
  - Implementations

sleep()

- Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds
- The thread *does not* lose ownership of any monitors
- Useful for:
  - periodic actions
    - need to wait for some period of time before doing the action again.
  - allow other threads to run

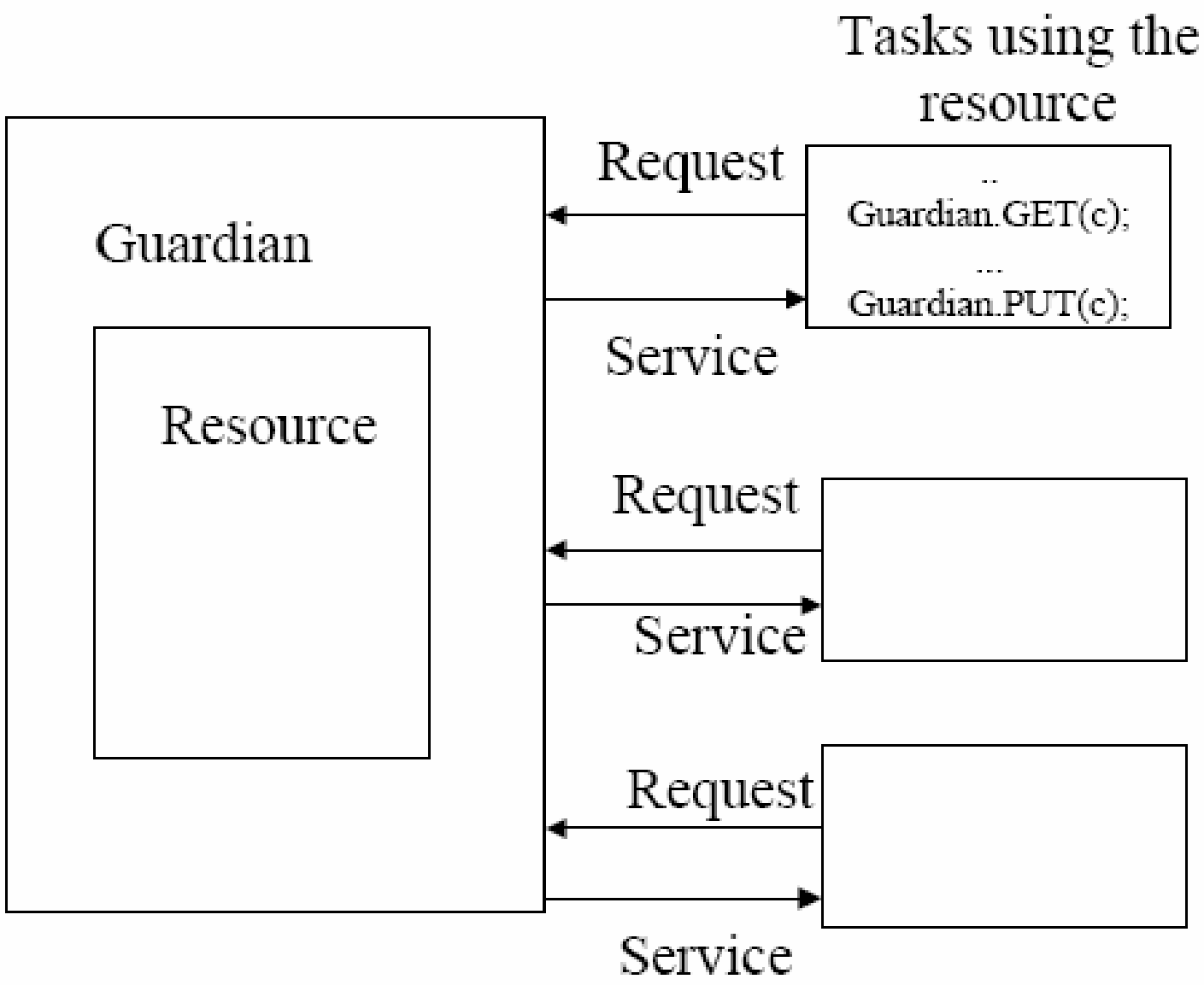
Transition States



A Thread can only be started once; if a Thread has stopped (reached the end of its run), nothing can happen anymore to the Thread.

Task Syncronization in Ada

- Guardians and rendez-vous
- The Ada style of designing concurrent systems
- In Ada a shared object is active (whereas a monitor is passive)
  - it is managed by a *guardian* process which can accept rendez-vous requests from tasks willing to access the object
- Free Ada compiler
  http://www.gnu.org/software/gnat/gnat.html


Tasks using the resource
Request
..
Guardian.GET(c);
...
Guardian.PUT(c);
Guardian
Service
Resource
Request
Service
Request
Service

A Guardian Task

note nondeterministic acceptance of rendez-vous requests (select+when)

```
loop
        select
            when NOT_FULL
                accept PUT (C: in CHAR) do
                This is the body of PUT; the client calls it as if it
                were a normal procedure
                end ;
            or
            when NOT_EMPTY
                accept GET (C: out CHAR) do
                This is the body of GET; the client calls it as if it
                were a normal procedure
            end ;
        end select ;
end loop ;
```

PUT and GET are the *entries* of the guardian task

Semantica dei rendez-vous

1) chiamata ad ENTRY (analoghe a chiamata di procedura)
2) il task chiamato deve fare una ACCEPT -> rendezvous

Di solito task body usa costrutto SELECT per decidere se e quando accettare.

```
SELECT
 WHEN C1 => PRG1
OR
 WHEN C2 => PRG2
OR
 PRG3 -- sempre aperta
ELSE
 PRG4
END SELECT;
```

- C1 e C2 sono alternative APERTE <=> C1, C2 = true
- PRGk puo` contenere ACCEPT o DELAY

1) esegui se esiste PRGk con Ck aperta e contenente una ACCEPT

2) se non esiste, allora esegui PRGk' con Ck' aperta e DELAY (il piu` piccolo DELAY aperto)

3) se tutto chiuso, esegui ramo ELSE. Se non c'e` ELSE, sospendi in attesa di apertura.

Tema d'esame

- Si consideri un conto corrente bancario possono accedere in modo concorrente per effettuare operazioni di versamento e di prelievo.
  - È definito un importo massimo prelevabile, e l'accesso al conto corrente da parte di un utente che vuole effettuare un prelievo è possibile solo se, nel conto corrente è presente un importo tale da rendere possibile il prelievo della cifra massima senza "andare in rosso"
  - L'accesso all'utente che vuole effettuare un prelievo è permesso se l'importo che si intende prelevare è inferiore all'attuale disponibilità
- Si dica quali delle soluzioni applicative (a) o (b) sono realizzabili con semplici programmi in Ada o in Java, motivando sinteticamente la risposta nel caso negativo.

Soluzione Java (1)

```
public class ContoCor {
        private int disponibile;
        private int soglia;
        ContoCor () {... disponibile=0; soglia=2000; ...}

        public synchronized void deposita(int importo){
                disponibile += importo;
                notifyAll();
        }

        public synchronized void ritira(int importo){
                while (!(disponibile >= soglia) )
                try { wait(); }
                catch(InterruptedException e) {}
                disponibile -= importo;
        }
}
```

Soluzione Ada (1)

```
task ContoCor is
    entry deposita(importo: in INTEGER);
    entry ritira(importo: in INTEGER);
end ContoCor;
task body ContoCor is
 disponibile: INTEGER := 0; soglia := 2000;
    loop
        select
            when disponibile >= soglia
            accept ritira(importo: in INTEGER) do
                disponibile := disponibile – importo;
            end ritira;
        or
            when true
            accept deposita(importo: in INTEGER) do
                disponibile := disponibile + importo;
            end deposita;
        end select;
    end loop;
end ContoCor;
```

Soluzione Java (2)

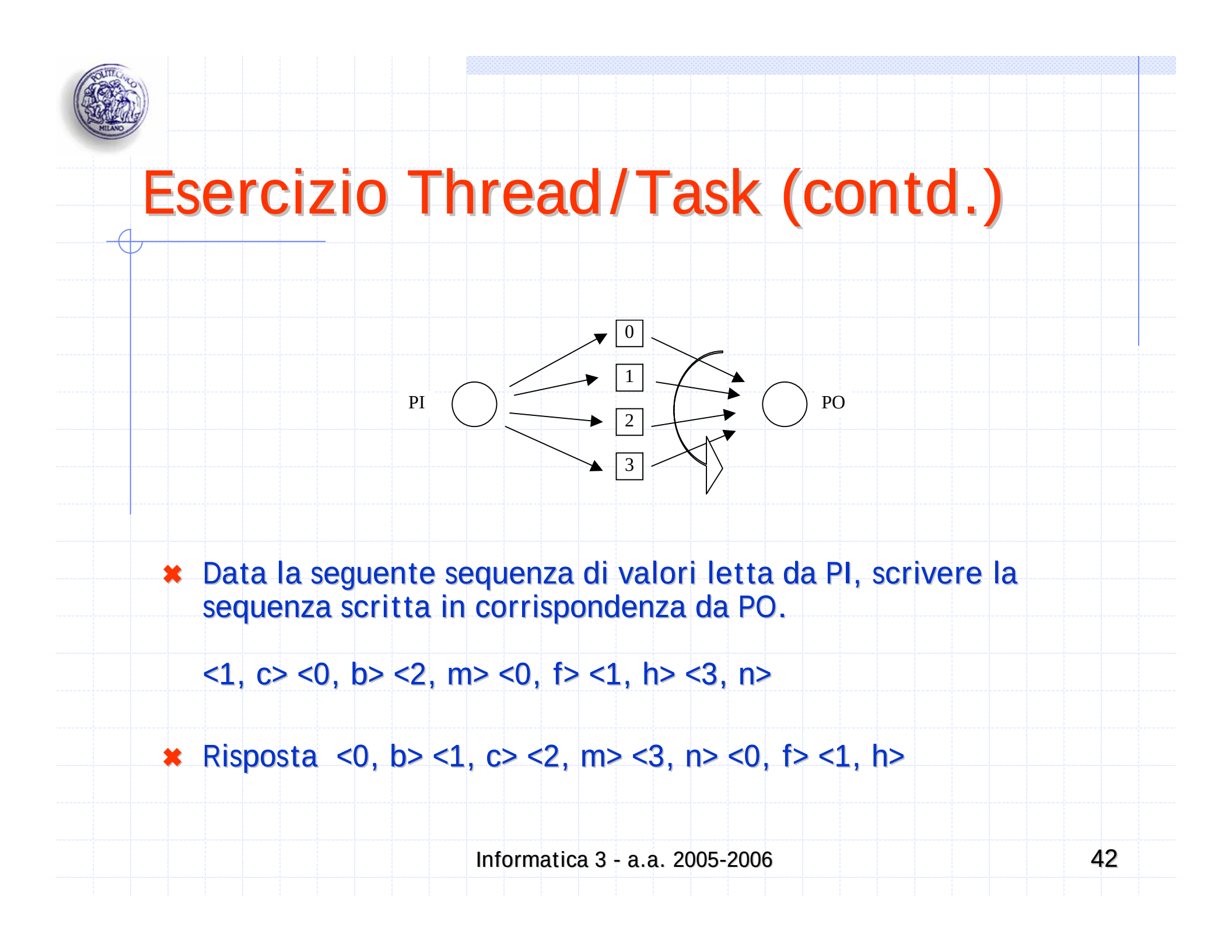
```
public class ContoCor {
        ... TUTTO COME SOPRA ...

        public synchronized void ritira(int importo){
                while (!(disponibile >= importo) )
                try { wait(); }
                catch(InterruptedException e) {}
                disponibile -= importo;
        }
}
```

Soluzione Ada (2)

- In Ada non esiste una semplice variante del caso (b) perché nella clausola when di una select non è possibile valutare il parametro della entry citata nella corrispondente accept;
- il valore del parametro risulta noto solo DOPO l'esecuzione della accept e quindi non può essere utilizzato per decidere l'esecuzione dell'accept stessa.

Esercizio Thread/Task

- Si realizzi in Java e in Ada il seguente sistema.
  - Il modulo PI esegue ripetutamente le seguenti operazioni: legge da tastiera una coppia di valori <i, ch>, dove i è un numero tra 0 e 3, ch un carattere, e inserisce il carattere ch nel buffer i (ognuno dei quattro buffer contiene al più un carattere).
  - Il modulo PO considera a turno in modo circolare i quattro buffer e preleva il carattere in esso contenuto, scrivendo in uscita la coppia di valori <i, ch> se ha appena prelevato il carattere ch dal buffer i.
  - L'accesso a ognuno dei buffer è in mutua esclusione; PI rimane bloccato se il buffer a cui accede è pieno, PO se è vuoto.



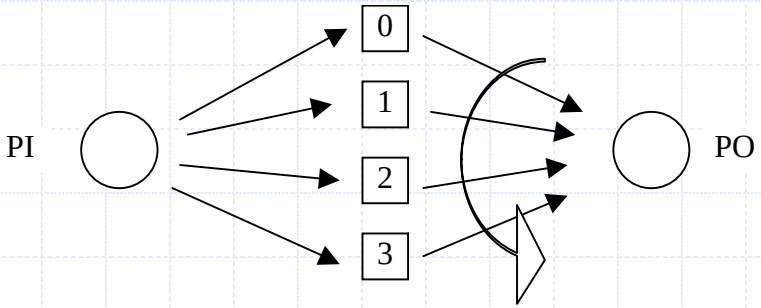
Esercizio Thread/Task (contd.)

- Data la seguente sequenza di valori letta da PI, scrivere la sequenza scritta in corrispondenza da PO.

  <1, c> <0, b> <2, m> <0, f> <1, h> <3, n>

- Risposta <0, b> <1, c> <2, m> <3, n> <0, f> <1, h>

Esercizio Thread/Task (contd.)



- Descrivere brevemente in quali casi si può verificare una situazione di deadlock tra PI e PO. Illustrare con un semplice esempio.
- Risposta: Deadlock: <1,a> <1,b>

Java

```
public class Buf {
    private char q;
    private boolean full;
    Buf() {
        full = false;
    }
    public synchronized void put (char item) {
        while (full)
            try { wait(); }
            catch (InterruptedException e) { }
        q = item;
        full = true;
        notify();
    }
    public synchronized char get () {
        while (!full)
            try { wait(); }
            catch (InterruptedException e) { }
        full = false;
        notify();
        return q;
    }
}
```

Task Pi

```
public class Pi extends Thread {
    private Buf[] buff;
    private String[] commands;
    Pi (Buf[] b, String[] c) {
        buff = b;
        commands = c;
    }
    public void run() {
        for(int i=0; i < commands.length; i++)
            buff[(int)commands[i].charAt(0)-(int)'0'].
                        put(commands[i].charAt(2));
    }
}
```

Task Po

```
public class Po extends Thread {
    private Buf[] buff;
    Po (Buf[] b) {
        buff = b;
    }

 public void run() {
        while(true) {
            for(int i=0; i < buff.length; i++)
                System.out.println("Buff "+i+":"+buff[i].get());
        }
    }
}
```

Main

```
public class pi_po {
    public static void main (String [] args){

        Buf[] bfs = new Buf[4];
        Pi pi0;
        Po po0;

        bfs[0] = new Buf();
        bfs[1] = new Buf();
        bfs[2] = new Buf();
        bfs[3] = new Buf();

        pi0 = new Pi(bfs, args);
        po0 = new Po(bfs);

        pi0.start();
        po0.start();
    }
}
```

ADA

```
task body BUF is
  Q : CHARACTER;
  FULL : BOOLEAN := FALSE;
begin
  loop
    select
      when not FULL =>
      accept PUT(X: in CHARACTER) do
        Q := X;
        FULL := TRUE;
      end PUT;
    or
      when FULL =>
      accept GET(X: out CHARACTER) do
        X := Q;
        FULL := FALSE;
      end GET;
    end select;
  end loop;
end BUF;
```

Task Pi

```
task body PI is
   B : INTEGER;
   V : CHARACTER;
begin
   loop
      TEXT_IO.PUT("Buff? >");
      Ada.Integer_Text_IO.GET(B);
      TEXT_IO.PUT("Char? >");
      TEXT_IO.GET(V);
      TEXT_IO.PUT_LINE(" ");
      BFS(B).PUT(V);
   end loop;
end PI;
```

Task Po

```
task body PO is
   I : INTEGER := 0;
   V : CHARACTER;
begin
   loop
      for I in 0..3 loop
         BFS(I).GET(V);
         Ada.Integer_Text_IO.PUT(I);
         TEXT_IO.PUT(":");
         TEXT_IO.PUT(V);
         TEXT_IO.PUT_LINE(" ");
      end loop;
   end loop;
end PO;
begin -- corpo
   null;
end PI_PO;
```