Modelli Concorrenti e Algoritmi Distribuiti

Concorrenza e Parallelismo

- Programmazione Parallela : contemporaneità **Fisica**
- Programmazione Concorrente : contemporeneità **Logica**

Due processi concorrenti sono programmi *sequenziali* che vengono eseguiti **contemporaneamente;** in un ambiente monoprocessore la concorrenza è ottenuta per "**interleaving**".
Lo scambio (o switching) tra i due processi è eseguito dallo scheduler.
Se i due processi devono eseguire una stessa operazione nello stesso istante cosa succede? E' necessario sapere se l'istruzione è **atomica**!

Un ' istruzione è **atomica** se viene sempre eseguita senza interruzioni.

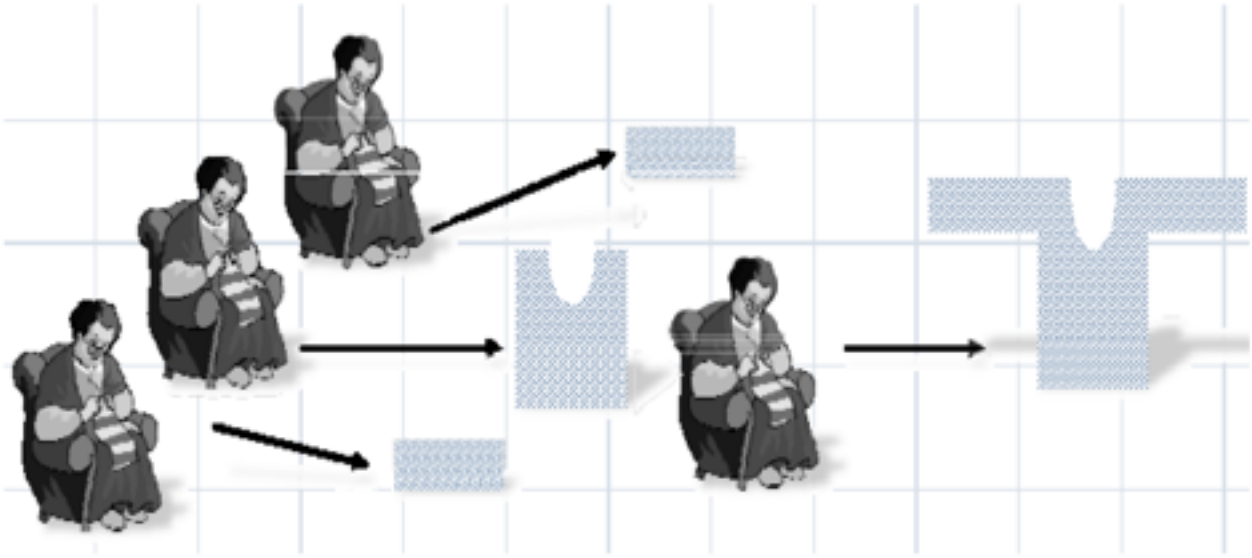
- in un ambiente *monoprocessore* quali sono le azioni atomiche (istruzioni macchina, blocchi di istruzioni)?
- In un ambiente *multiprocessore quali sono le azioni atomiche (istruzioni macchina, microistruzioni)??*

Tipi di Processi Concorrenti

- Processi **indipendenti**
- Processi **cooperanti**
    - Processi che lavorano insieme per un compito comune
- Processi **in competizione**
    - Tipici dei sistemi operativi e dei sistemi di rete

Processi indipendenti
Relativamente rari e poco interessanti

Processi cooperanti - Sincronizzazione



Processi in competizione - condividono risorse

Riassumendo

Per la programmazione concorrente

- occorrono **nuovi costrutti linguistici** per indicare la concorrenza, la comunicazione e la sincronizzazione

- La **correttezza** di un programma concorrente richiede la validità di speciali proprietà (assenza di deadlock, di starvation, ...)

- Per valutare se un programma concorrente è corretto occorrere conoscere alcuni **dettagli architetturali** (atomicità)

Programmi e processi

Algoritmi, Programmi, Processi

Algoritmo : Procedimento logico che deve essere eseguito per risolvere un problema
Programma: descrizione dell'algoritmo mediante un opportuno formalismo (es: C, Java)
Processo: Sequenza di azioni eseguite dall'elaboratore quando esegue un programma

Processo

Entità dinamica che si svolge nel tempo, si può descrivere mediante:

- la **durata** (tempo da inizio a fine del processo)
- Lo **stato del processore** in un particolare istante
- La **traccia dell'esecuzione,** sequenza degli stati del processore
- Il **grafo di precedenza:** i nodi del grafo rappresentano gli eventi e gli archi le precedenze temporali

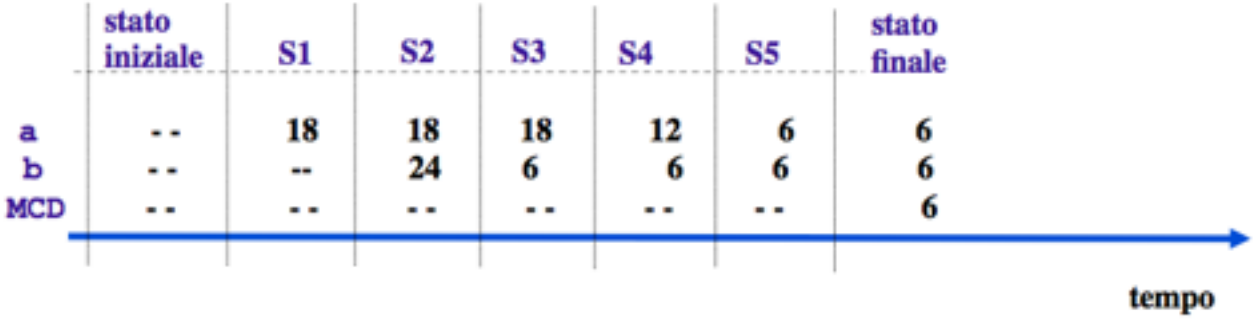
Esempio MDC di due numeri naturali **x** e **y**

Proprietà

a. MCD (x, x) = x
b. MCD (x, y) = MCD (y, x)
c. Se x > y allora MCD (x, y) = MCD (x-y, y)

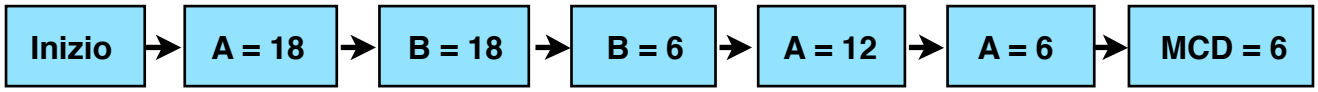
Programma

```
int     MCD (int x, int y)(
        int a, b;
        a = x;
        b = y;
        while(a!=b)
                if (a>b)   a = a-b;
                else       b = b-a;
        return a;
)
```



Grafo di esecuzione del processo (x = 18, y = 24)

Grafo di precedenza ad ordinamento totale del processo che calcola l' MCD (18, 24)



Le proprietà più importanti di un **programma sequenziale** sono :

1. **Terminazione**
2. **Correttezza parziale:** se un processo del programma termina, allora i risultati sono corretti
3. **Correttezza totale:** ogni processo del programma termina e produce risultati corretti

Proprietà di Concorrenza

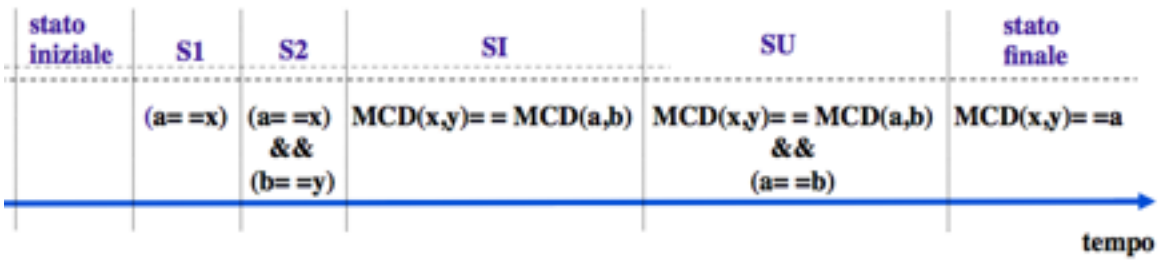
Nella **programmazione concorrente** è necessario verificare anche molte altre proprietà:

- garanzia di **mutua esclusione**
- L'assenza di **deadlock**
- L'assenza di **starvation**

Test e Verifica di un programma

Per controllare la correttezza di un programma si vuole eseguire il programma un certo numero di volte cambiando l'input ed individuare gli eventi anomali (**debugging**).
Per raggiungere una maggiore garanzia si può ricorrere a metodi formali di verifica che fanno uso della logica dei predicati. Si ragiona su una **traccia simbolica** che fa riferimento nn ai valori assunti dalle variabili ma ai domini di valori che le variabili possono assumere.

```
int MCD (int x, int y) {
      int a, b;
      a = x;
      b= y;
      while (a != b)
            if (a > b) a = a - b;
            else b = b - a;
  return a;
}
```



SI rappresenta lo stato iniziale del ciclo ed è caratterizzato dalla relazione (**invariate ciclica**)

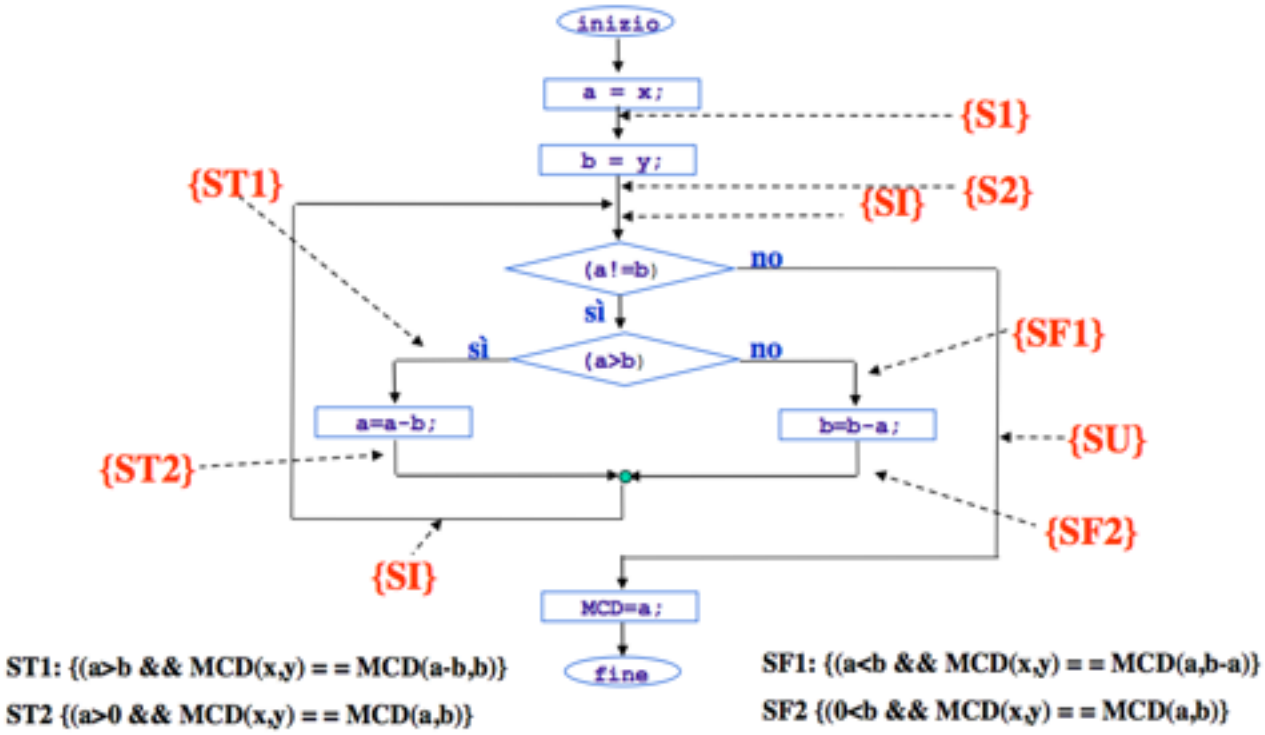
{ MCD (x, y) == MCD (a , b) }

SU rappresenta lo stato finale del ciclo ed è caratterizzato dalla relazione:

{ MCD (x , y) == a }

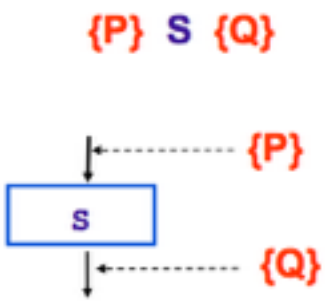
Occorre però dimostrare che **{ MCD (x, y) == MCD (a , b) }** vale in ogni ingresso di ciclo; è sufficiente verificare che:

A. La relazione è vera la prima volta che si esegue il ciclo **while**
B. Se la relazione è vera all'inizio del ciclo **while** allora è vera anche alla fine e quindi vale ancora all'inizio del ciclo successivo (INDUZIONE)

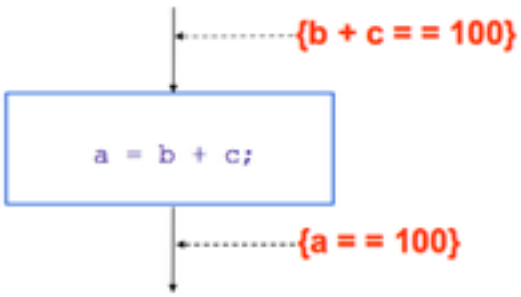


Asserzioni

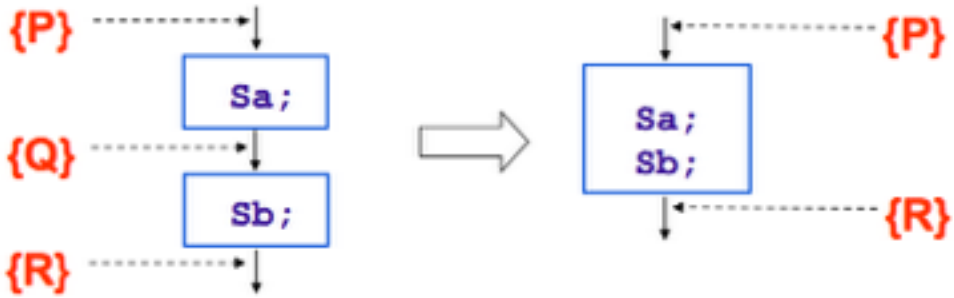
Precondizioni e postcondizioni



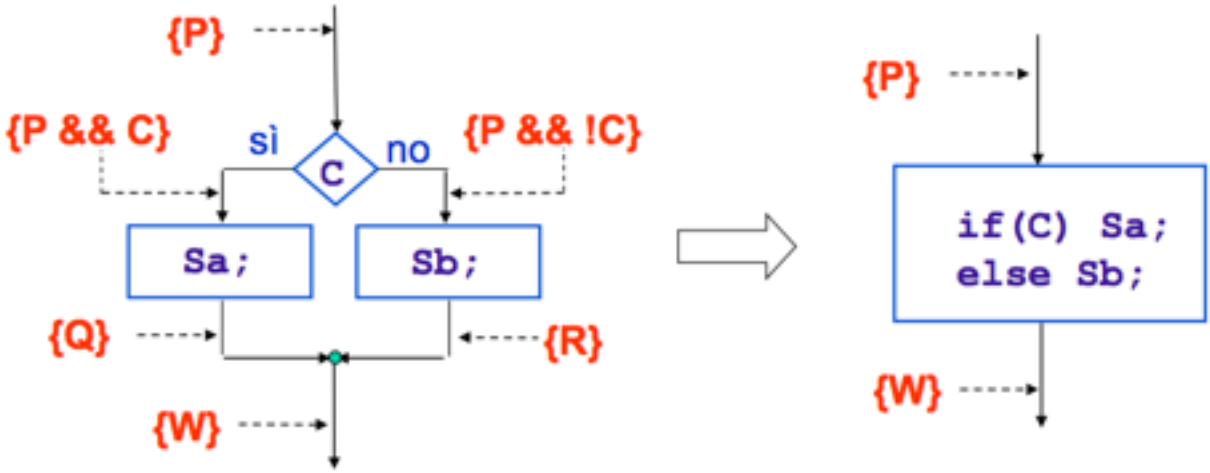
Assegnamento



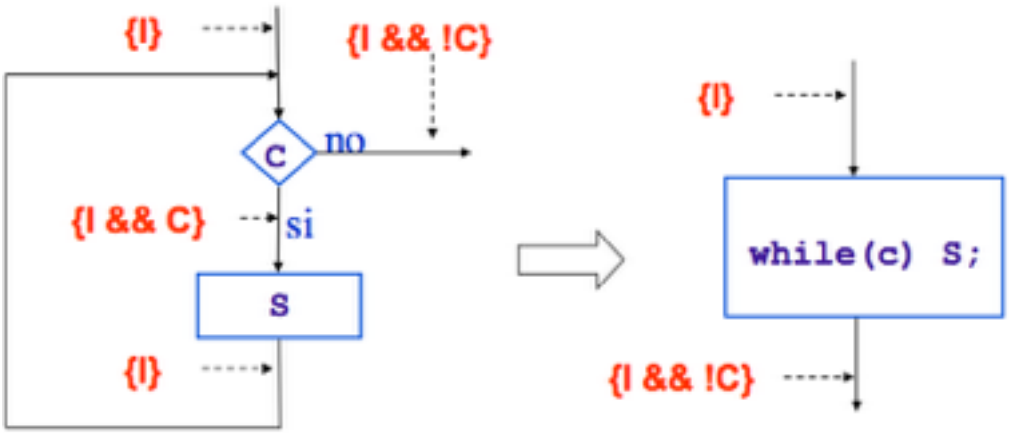
Regola di composizione



Regola condizionale



Regola ripetitiva



Nel caso della programmazione concorrente la prova di correttezza tramite l'uso di invarianti è più complicato, supponiamo di voler provare che nel programma sotto vale l'invariante **i: y <= x.**
Possiamo sfruttare alcune caratteristiche della struttura del programma per stabilire degli invarianti topologici: ad esempio due azioni sequenziali occorrono sempre nello stesso ordine.
Se **A** è un' azione nel programma, **nA** indica il numero di esecuzioni complete di **A**

A -> < A ; nA := nA +1 >

```
int x = 0; y = 0;
public Main {
  cobegin
    process Plus () {
    while (true) {
          x=x+1;
          y=y+1;
     }
    }
    process Minus () {
    while (true) {
          y=y-1;
          x=x-1;
     }
    }
  coend
}
```

Siano:
X^+ : incremento di x
Y^+ : incremento di y
X^- : decremento di x
Y^- : decremento di y

Sfruttando la *topologia* del programma abbiamo i seguenti *invarianti:*

I0: $x = nX^+ - nX^-$ I2: $0 \leq nX^+ - nY^+ \leq 1$
I1: $y = nY^+ - nY^-$ I3: $0 \leq nY^- - nX^- \leq 1$

I2 => $nY^+ \leq nX^+$
I3 => $nY^- \leq nX^-$
$nY^+ - nY^- \leq nX^+ - nX^-$ =>

=> (I0 e I1) $y \leq x$

Sistemi Concorrenti

La programmazione concorrente nasce per gestire i **sistemi concorrenti** cioè sistemi in grado di supportare più utenti (o programmi) **contemporaneamente.**

- **Sistemi intrinsecamente concorrenti**
  - Sistemi Real Time
  - Sistemi operativi
  - Gestione di base di dati
- **Applicazione potenzialmente concorrenti**
  - Uso di algoritmi paralleli per computazioni

Per sistema concorrente intendiamo:
- un sistema software
- Implementato su vari tipi di hardware
- Che porta avanti **contemporaneamente** una molteplicità di attività diverse
- tra di loro correlati (cooperano o competono)

Grafo di precedenza di un processo

- I **nodi** del grafo rappresentano i singoli **eventi**
- Gli **archi** del grafo rappresentano le **precedenze temporali**
- Se il processo è sequenziale, il grafo sarà a **ordinamento totale** (ogni nodo ha un predecessore ed un successore)
- L' **ordinamento totale** del grado è solo in parte dovuto alla natura del problema
- La natura del problema impone di fatto solo un ordinamento parziale tra gli eventi

Grado di precedenza ad ordinamento parziale

Correttezza di un programma concorrente

Nei programmi concorrenti alcune computazioni possono essere corrette altre no, ma le normali tecniche di debugging non funzionano poiché **diverse esecuzioni dello stesso programma possono dare risultati diversi.**

Le tecniche più importanti per la verifica di proprietà di correttezza sono:
- dimostrazioni induttive di **invarianti**.
- Uso di **model checker**. Un model checker è un programma che costruisce il diagramma degli stati di un programma concorrente e simultaneamente verifica una proprietà di correttezza.

Fairness

Esistono due categorie di proprietà di **correttezza**:

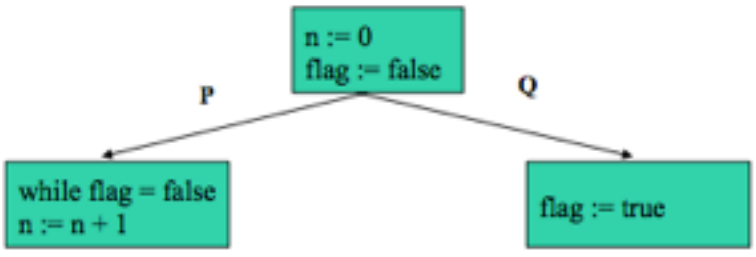
- proprietà di **Safety**:
  - La proprietà P deve sempre essere vera
    (P è vera in ogni stato della computazione es: GUI SO cursore mouse)

- Proprietà di **Liveness:**
  - La proprietà P prima o poi sarà vera
    (in ogni computazione esiste uno stato in cui P è vera)
    Esempio GUI del SO: (se clicco sul mouse il cursore prima o poi cambia)

Ogni possibile interleaving di istruzioni è considerato essere una computazione di un programma concorrente, però questo comporta che in alcune computazioni ci siano statement che non sono mai eseguiti. La proprietà di **fairness** esclude queste computazioni.

Proprietà di Fairness

Una computazione è **fair** se per ogni suo stato è vero che uno **statement sempre abilitato, prima o poi** sarà eseguito!



Interazioni tra processi

Le interazioni tra i processi possono essere classificate:

- **Cooperazione**: interazioni prevedibile e desiderata
  - Scambio di segnali temporali
  - Scambio di informazioni
- **Competizione :** Interazione prevedibile e non desiderata, ma necessaria (sync indiretta o implicita)
  - Mutua esclusione

Linguaggi concorrenti

Un linguaggio concorrente è un linguaggio sequenziale con costrutti per la specifica della **concorrenza** e costrutti per la specifica delle **interazioni.**

Architetture per sistemi concorrenti

SISD: Single **I**nstruction **S**tream, single **D**ata stream
Modello uniprocessore convenzionale (Macchina di Von Neumann)

SIMD: Single **I**nstruction stream, **M**ultiple data stream
Più processori che eseguono la stessa istruzione su diversi “array”

MIMD: Multiple **I**nstruction stream, **M**ultiple **D**ata stream
Più processori che eseguono istruzioni diverse; possiamo ulteriormente distinguere tra:
- Sistemi multiprocessori con memorie comune
- Sistemi a rete senza memoria comune

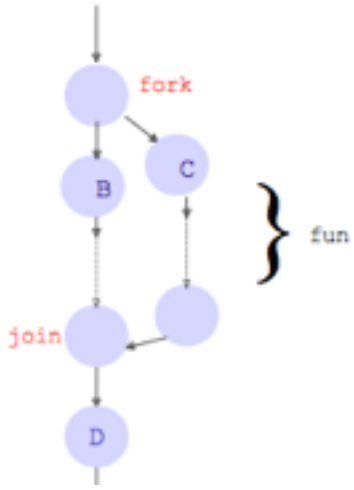
Specifica della concorrenza

Esprimere la concorrenza

L'esecuzione di una **fork** coincide con la creazione e l'attivazione di un processo che inizia la propria esecuzione in parallelo con quella del processo chiamante.

La **join** consente di determinare quando un processo, creato tramite la fork, ha terminato il suo compito, sincronizzandosi con tale evento.

```
  process  p;
  =======
A: ........;
  p = fork fun;
B: ........;
  =======
  join p;
D: ........;
  =======
void  fun () {
  C: ........;
  =======
}
```



La concorrenza viene espressa così:

```
cobegin
     S1;       // istruzione eseguita in parallelo
     S2;       // istruzione eseguita in parallelo
     ...
     Sn;       // istruzione eseguita in parallelo
coend
```

Ogni istruzione Si all'interno può contenere altre istruzioni "cobegin .. coend".

Processo

Costrutto linguistico per individuare, in modo sintatticamente preciso, quali moduli di un programma possono essere eseguiti come processi autonomi.

```
process <identificatore> (<parametri formali>){

     <dichiarazione di variabili locali>
     <corpo del processo>

}
```

Libreria pthread (Linguaggio C)

Il thread è l'unità di scheduling, ed è univocamente individuato da un identificatore interno

```
Pthread_t  tid;
```

Il tipo phthread_t è dichiarato nell'header file:

```
#include <pthread.h>
```

La creazione di un nuovo thread avviene con la chiamata della primitiva pthread_create che prende 3 parametri in ingresso:

1. `thread` : è il puntatore alla variabile che raccoglierà il thread_id
2. `start_routine` : è il puntatore alla funzione che contiene il codice del nuovo thread
3. `arg`: è il puntatore all'eventuale vettore contenente i parametri da esec.
4. `attr`: può essere usato per specificare eventuali attributi

il thread può terminare chiamando il metodo pthread_exit.

Concorrenza e Sincronizzazione

Esercizio 1

Si consideri il seguente programma concorrente, in cui la sincronizzazione tra processi è espressa usando il formalismo `region s << S1; when ( C ) S2; >>`

```
int x = ?
cobegin
      region x << when (x > 0) x = x - 1; >>
      region x << when (x < 0) x = x + 2; >>
      region x << when (x == 0) x = x - 1; >>
coend
```

a) Quali sono i valori iniziali della variabile x per cui il programma termina , nella ipotesi che lo scheduler garantisca l'attivazione (in tempi finiti) ad ogni processo che diventa e rimane costantemente pronto (scheduler weakly fair) ?

Soluzione.

a) **0, 1, -1**

Esercizio 2

Si consideri il seguente programma concorrente, in cui la sincronizzazione tra processi è espressa utilizzando il formalismo: `region s << S1; when ( C ) S2; >>`

```
int x = 10
bool c = True
cobegin
S1:   region x  << when (x == 0) no-op >> c = False;
S2:   while ( c ) region x << x = x - 1; >>

coend
```

Il programma termina?

Soluzione.

Se viene eseguito 10 volte l'"istruzione `while ( c ) region x << x = x - 1; >>` x arriva a valere 0 e quindi c potrà assumere il valore False e S2 potrà terminare correttamente.

ESERCIZI SU INVARIANTI SEMAFORICI DA FARE DOPO CAPITOLO! (4-ESERCIZI.DOC)

Sincronizzazione tramite Invarianti semaforiche

Semafori

Si supponga che ci siano due processi concorrenti **Px** e **Py :**

```
Process Px {
while (true){x++}
 }
```

```
Process Py {
while (true){y++}
 }
```

Si vuole sincronizzare l'attività di **Px** e **Py** in modo che sia sempre vera la relazione (invariante **I0**)

I0: $x \leq y$

Topologia del codice

Indicando con **nX+(t)** il numero di esecuzione dell'istruzione **x++** sino all'istante **t** e con **nY +(t)** il numero di esecuzione dell'istruzione **y++** sino all'istante **t,** l'invariante **I0** può essere scritto come :

I0 : $\Box t.nX+(t) \leq nY+(t)$

Per la sincronizzazione induciamo un semaforo *sem1* e inseriamo un'operazione *P(sem1)* prima di **x++** e un'operazione *V(sem1)* dopo **y++**

```
Process Px {
while (true){
   P(sem1)
   x++}
 }
```

```
Process Py {
while (true){
    V(sem1)
    y++}
 }
```

Per la topologia del codice abbiamo:

I1: $nX+(t) \leq nP(sem1, t)$ e **I2 :** $nV(sem1, t) \leq nY+(t)$

combinando questi due invarianti con **l'invariante semaforica**

Isem1: $nP(sem1, t) \leq Isem1 + nV(sem1, t)$

otteniamo così:

$$nX+(t) \leq nP(sem1, t) \leq Isem1 + (nV(sem1, t) \leq nY+(t))$$

quindi:

$$nX+(t) \leq nP(sem1, t) \leq Isem1 + nV(sem1, t) \leq Isem1 + nY+(t)$$

Quindi, scegliendo **lsem = 0** otteniamo la validità della relazione richiesta!

Altri vincoli

Supponiamo di richiedere anche che sia sempre vero:

$I3 : \quad y \leq x+10$

i.e.

$\Box t.\ nY+(t) \leq nX+(t) + 10$

Introduciamo un nuovo semaforo *sem2* e inseriamo un'operazione *P(sem2)* prima di *y++* e un operazione *V(sem2)* dopo *x++.* Allora:

$$nY+(t) \leq nP(sem2, t) \leq lsem2 + nV(sem2, t) \leq lsem2 + nX+(t)$$

Scegliendo **lsem2 = 10** otteniamo la validità del vincolo richiesto!

```
Process Px {
while (true){
   P(sem1)
   x++
   V(sem2)}
 }
```

```
Process Py {
while (true){
    P(sem2)
    y++
    V(sem1)}
 }
```

ancora vincoli…

Supponiamo di modificare il vincolo **I0** in modo che la richiesta ora sia:

$I4 : \quad 2x \leq y$ i.e.

$\Box t.\ 2*nX+(t) \leq nY+(t)$

Occorre far precedere **x++** da due occorrenze di *P(sem1)* (denotate con *P2 (sem1)*) allora:

$$\Box t.2*nX+(t) \leq nP+(sem1, t)$$

quindi

$$2*nX+(t) \leq nP(sem1,t) \leq lsem1 + nV(sem1,t) \leq lsem1 + nY+(t)$$

```
Process Px {
while (true){
   P(sem1)
   P(sem1)
   x++
   V(sem2)}
 }
```

```
Process Py {
while (true){
    P(sem2)
    y++
    V(sem1)}
 }
```

Generalizzando

Problema. Dato un insieme di processi concorrenti che eseguono le azioni **A, B, C, D** si vuole imporre una condizione di sincronizzazione, invariante, del tipo:

SYNC: $a*nA(t) + c*nC(t) \leq b*nB(t)+d*nD(t) + e$

dove **a, b, c, d, e** sono numeri interi non negativi.

Soluzione.

Si introduce un semaforo *sem,* con valore iniziale **Isem = e** e si sostituiscono le azioni **A, B, C, D** con:

```
A □ Pa(sem); A
B □ B; Vb(sem)
C □ Pc(sem); C
D □ D; Vd(sem)
```

Deadlock

Le condizioni di sincronizzazione possono essere in conflitto e dare luogo a **deadlock**

Esempio: siano Px e Py come negli esempi precedenti

I4 : 2x≤ y => □t. 2*nX+(t) ≤ nY+(t)

I3 : y≤ x+10 => □t. nY+(t) ≤ nX+(t) + 10

Dopo alcuni passi il sistema va in deadlock

Esercizio 1.

Si consideri l'esecuzione concorrente di tre processi sotto riportati, nel caso in cui i valori iniziali delle variabili **x** e **y** siano entrambi a 0.

```
while (true) {x = x+2 }
```

```
while (true) {y = y-1 }
```

```
while (true) {x = x-1;
              y = y+ 2}
```

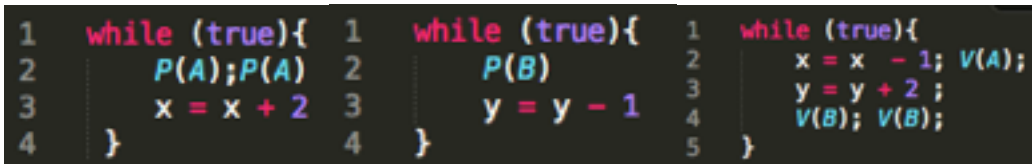
Si inseriscano le opportune primitive semaforiche in modo che siano sempre vere le condizioni :

I0: 0 ≤ y
I1: x ≤ 10

Si commenti l'eventuale presenza di deadlock

Soluzione.

A = 10; B = 0;



```
while (true){
    P(A);P(A)
    x = x + 2
}
```

```
while (true){
    P(B)
    y = y - 1
}
```

```
while (true){
    x = x  - 1; V(A);
    y = y + 2 ;
    V(B); V(B);
}
```

Esercizio 2.

Si consideri l'esecuzione concorrente di tre processi sottoriportati, nel caso in cui i valori iniziali delle variabili **x,y** e **z** siano tutte a 0.

```
while (true) {x = x+2;
              y = y-1;
              z = z-1 }
```

```
while (true) {y = y+2 }
```

```
while (true) {z = z+1;
              x = x-2 }
```

Si inseriscano le opportune primitive semaforiche in modo che siano sempre vere le seguenti condizioni:

I0: $x + y + z \leq 10$
I1: $y \leq 5$

La soluzione secondo le schema fornito può dar luogo a deadlock. Si mostri un possibile scenario.

Soluzione.

Processo 1.

```
int semY = 5, semSum = 10;

while(true){

    P(semSum);
    P(semSum);
    x = x + 2;
    y = y - 1;
    V(semY);
    V(semSum);
    z = z - 1;
    V(semSum);

}
```

Processo 2.

```
while(true){
    P(semY);
    P(semY);
    P(semSum);
    P(semSum);
    y = y + 2;
}
```

Processo 3.

```
while(true){
    P(semSum);
    z = z + 1;
    x = x - 2;
    V(semSum);
    V(semSum);
}
```

Possibile scenario di **deadlock:**
stato dei semafori → **semY = 1 e semSum = 0** indifferente tocca a **Processo2**
viene eseguita la prima P quindi semY = 0, con la seconda p su Y non verrà mai fatta perchè non vi è la possibilità di fare una V su semY dato che tutti gli altri processi sono bloccati sul semSum.

Semafori

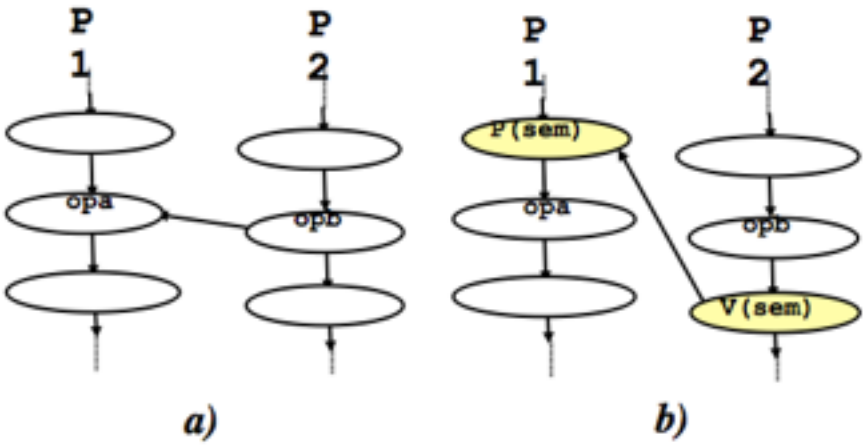
Un semaforo è un tipo di dati astratto, che assume valori interi ≥ 0, su cui si eseguino solo due primitive atomiche: **P** e la **V**.

Semaforo binario

Può assumere solo i valori 0 e 1, viene chiamato **mutex**, perchè viene usato per la mutua esclusione.

Semafori evento: scambio di segnali temporali

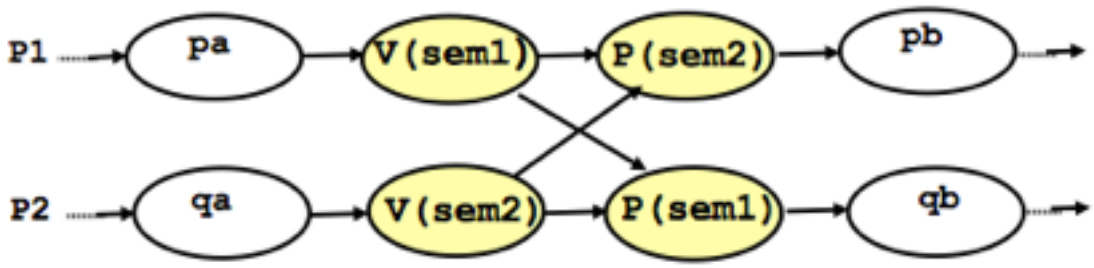
Con **semafori evento** si intendono dei semafori binari utilizzati per imporre un vincolo di precedenza tra le operazioni dei processi.



I due processi **P1** e **P2** eseguono ciascuno due operazioni **pa** e **pb** il primo e **qa** e **qb** il secondo.

Vincolo di rendez-vous: l'esecuzione di **pb** da parte di **P1** è **qb** da parte di **P2** possono iniziare **solo dopo** che entrambi i processi hanno completato la loro prima operazione.

Scambio di segnali temporali in **modo simmetrico**:
ogni processo quando arriva all'appuntamento segnala di esserci arrivato e attende l'altro.



Produttori - Consumatori

P1 e **P2** si scambiano dati di tipo **T** utilizzando una memoria (**buffer**) condivisa.

Vincoli di sincronizzazione:

- Accessi al buffer **mutuamente esclusivi;**
- **P2** può prelevare un dato solo **dopo** che **P1** lo ha inserito;
- **P1** prima di inserire un dato, deve attendere che **P2** abbia prelevato il precedente.

```
void invio(T dato) {
   P(empty);
   <inserisci(dato)>;
 V(empty);
}
```

```
T ricezione( ) {
   T  dato
    P(full)
   <dato = estrai()>;
   return dato;
}
```

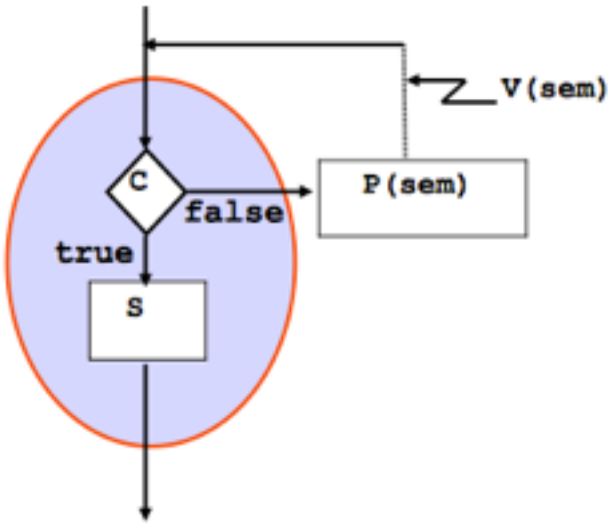
```
V(full);
```

il **buffer** inizialmente è vuoto e il valore di **empty** è 1 mentre **full** è 0.

Semafori Condizione

```
void op1 (  ) :  region R << when ( C ) S1; >>
```

- `op1 ( )` → è una regione critica, dovendo operare su una risorsa condivisa
- `S1` → ha come precondizione la validità della condizione logica **C**



In questo schema viene realizzato la **region** sospendendo il processo sul semaforo **sem** da associare alla condizione (semaforo condizione).
E' evidentemente necessaria un'altra operazione **op2** che chiamata da un'altro processo, modifichi lo stato interno di **R** in modo che **C** diventi vera.
Nell'ambito di **op2** viene eseguita la **V(sem)** per risvegliare il processo.

Schema con attesa circolare:

```
Struttura dati della risorsa R
    semaphore mutex = 1;
    semaphore sem = 0;
    int csem = 0;

public void op1( ) {
   P(mutex);
   while (!C) {
       csem ++;
       V(mutex);
       P(sem);
       P(mutex);
   }
   S1;
   V(mutex);
}
```

```
public void op2( ) {

   P(mutex);
S2 ;
   if (csem>0){
       csem --;
       V(sem)
   }
   V(mutex);
}
```

Schema con passaggio di testimone

```
Struttura dati della risorsa R
    semaphore mutex = 1;
    semaphore sem = 0;
    int csem = 0;

public void op1( ) {
   P(mutex);
   if (!C) {
      csem ++;
      V(mutex);
      P(sem);
      csem --;
   }
   S1;
   V(mutex);
}
```

```
public void op2( ) {

   P(mutex);
   S2;
   if(csem>0) V(sem);
     else V(mutex);
}
```

Il secondo schema è **più efficiente** del primo, ma consente di risvegliare **un solo processo alla volta** poiché ad uno solo si può passare il diritto di operare in mutex.

La condizione **C** verificata all'interno di **op1** deve essere verificabile anche all'interno di **op2**. Ciò significa che **non deve contenere variabili locali** o parametri della funzione op1

Gestione di un pool di risorse equivalenti (Semafori condizione):

- Ciascun processo può operare su una qualsiasi risorsa del pool purché libera.
- Necessità di un **gestore** che mantenga aggiornato lo stato delle risorse
- Ogni processo quando deve lavorare su una risorsa **chiede al gestore** l'allocazione di una di esse
- il gestore **assegna** al processo una risorsa libera (se esiste), in modo dedicato, passandogli l'indice relativo.
- il processo opera sulla risorsa senza preoccuparsi della mutua esclusione
- al termine il **processo rilascia** la risorsa al gestore.

```
class tipo_gestore {
   semaphore mutex=1;
   semaphore sem=0;
   int csem=0;
   boolean libero[N];
   int avail=N
   {for(int i=0;i<N;i++)libero[i]=true;}

 public int richiesta(){
   int i=0;
   P(mutex);
   if (avail==0){
       csem++;
       V(mutex);
       P(sem);
       csem--;
   }
   while(!libero[i]) i++;
   libero[i]=false;
   avail--;
   V(mutex);
   return i;
}
}
```

```
      public void rilascio(int r){
        P(mutex);
      libero[r]=true;
          avail++;
       if (csem>0)    V(sem);
      else V(mutex);
        }
```

Semafori contatori

Si definiscono come semafori binari rilasciando il vincolo che il valore appartenga all'insieme {0,1} per i semafori contatore il valore può assumere **qualunque intero:**

$$0 \leq vals \leq max$$

Vale ancora l'invariante semaforico : **nps ≤ ls + nvs**

```
class tipo_gestore {
   semaphore mutex=1;
   semaphore ris=N;
   boolean libero[N];
   {for(int i=0;i<N;i++)libero[i]=true;}

 public int richiesta(){
   int i=0;
   P(ris);
   P(mutex);
   while(!libero[i]) i++;
   V(mutex);
}

        public void rilascio(int r){
          P(mutex);
         libero[r]=true;
       V(mutex);
              V(ris;)
       }              }
```

Produttore / consumatore con buffer circolare

```
class coda_di_n_T {
   T vettore[n];
   int primo=0;ultimo=0;
   public void inserisci(T dato){
       vettore[ultimo]=dato;
       ultimo =(ultimo + 1)%n;
   }
   public T estrai(){
       T dato=vettore[primo];
       primo=(primo+1)%n;
       return dato;
   }
}
```

```
coda_di_n_T    buffer;
      semaphore    full=0
      semaphore    empty=0
      semaphore    mutex=0


void invio(T dato) {
   P(empty);
   P(mutex)
   buffer.inserisci(dato);
   V(mutex)
   V(full);
 V(empty);
}
```

```
          T ricezione( ) {
           T  dato
          P(full)
                P(mutex)
          dato = buffer.estrai
          V(mutex)

       return dato;
}
```

Semafori Privati

Definiamo **semaforo privato per un processo Q** un semaforo contatore per cui l'esecuzione dell'operazione di **P(sem)** è riservata esclusivamente al processo proprietario **Q**

L'esecuzione dell'operazione **V(sem)** non è soggetto ad alcuna restrizione.

Specifica di strategie di allocazione

Condizione di sincronizzazione:

- Qualora si voglia realizzare una determinata politica di gestione delle risorse, la decisione se ad un dato processo sia consentito proseguire l'esecuzione dipende dal verificarsi di una condizione detta **condizione di sincronizzazione,** espressa in termini

di variabili che rappresentano **lo stato della risorsa e di variabili locali ai singoli processi**
- Più processi possono essere bloccati durante l'accesso ad una risorsa condivisa, ciascuno in attesa che la propria condizione di sincronizzazione sia verificata. In seguito alla modifica dello stato della risorsa da parte di un processo, le condizioni di sincronizzazione di alcuni processi bloccati possono essere **contemporaneamente** verificate
- **Problema:** quale processo mettere in esecuzione??? Se i processi sono bloccati su un semaforo, la scelta del processo da svegliare viene effettuata tramite l'algoritmo implementato nella **V(sem)** che normalmente coincide con quello **FIFO**
- Talvolta è però necessario seguire una ben determinata **politica di risveglio dei processi**

Esempio 1

Su un buffer di N elementi più produttori possono depositare messaggi di dimensione diversa. La politica di gestione prevede:

- tra i produttori ha una **priorità di accesso** quello che fornisce il messaggio di **dimensione maggiore**
- finché un produttore, il cui messaggio ha dimensioni **maggiori** dello spazio disponibile nel buffer, rimane sospeso **nessun altro produttore** può depositare un messaggio anche se la sua dimensione potrebbe essere contenuta nello spazio libero del buffer.

Condizione di sincronizzazione:

Il deposito può avvenire se c'è sufficiente spazio per memorizzare il messaggio e non ci sono produttori in attesa. Il prelievo di un messaggio da parte di un consumatore il messaggio e non ci sono produttori in attesa.

Se lo spazio disponibile non è sufficiente **nessun produttore viene riattivato.**

```
class tipo_gestore_risorsa{
     <struttura dati del gestore>;
     semaphore mutex =1;
     semaphore priv[n] = {0,0,..0};

     public void  acquisizione (int i) {
    P(mutex);
    if(<condizione di sincronizzazione>){
            <allocazione della risorsa>;
            V(priv[i]);
   }
   else <registrare la sospensione del processo>;
   V(mutex);
              P(priv[i]);
     }
```

```
public void  rilascio( ) {
   int i;
   P(mutex);
   <rilascio della risorsa>;
   if(<esiste almeno un processo sospeso per il quale la
 condizione di sincronizzazione è soddisfatta>)  {
         <scelta fra i processi sospesi del processo Pi da
   riattivare>;
         <allocazione della risorsa a Pi>;
         <registrare che Pi non è più sospeso>;
         V(priv[i]);
   }
   V(mutex);
}
```

Pregi della soluzione

La sospensione del processo, **non può avvenire entro la sezione critica** in quanto ciò impedirebbe ad un processo che rilascia la risorsa di accedere a sua volta alla sezione critica e di riattivare il processo sospeso. La sospensione avviene al di fuori della sezione critica.

La specifica del particolare algoritmo di assegnazione della risorsa è lasciato la programmatore che può scegliere esplicitamente il processo da attivare **eseguendo l'operazione di V sul suo semaforo privato**.

Difetti della soluzione
L'operazione **P** sul semaforo privato viene **sempre eseguita** anche quando il processo richiedente non deve essere bloccato.

Il codice relativo all'assegnazione della risorsa viene duplicato nelle procedure acquisizione e rilascio.

[Secondo schema e soluzione slide **Modello a Memoria comune pagina 47**]

Semafori privati per gruppi di processi: lettori / scrittori

```
class tipo_gestore_file {
   semaphore mutex = 1;
   semaphore lett = 0;
   semaphore scritt = 0;
   int lb = 0;
   int sb = 0;
   int la = 0;
   boolean sa = false;

public void richiesta_scrittura () {        public void fine_scrittura()    {
   P(mutex);                                   P(mutex);
   if (la>0 OR sa)    {                          sa = false;
       sb++;                                if (lb>0)    V(lett);
       V(mutex);                            else if (sb>0)    V(scritt)
       P(scritt);                             else V(mutex)
       sb--;                                }
   }
   sa=true;
   V(mutex);
}


public void richiesta_lettura () {        public void fine_lettura()    {
   P(mutex);                                   P(mutex);
   if (sb>0 OR sa)    {                          la --;
       lb++;                                if (la==0&&sb>0)    V(scritt);
       V(mutex);                            else        V(mutex);
       P(lett);                           }
       lb--;
   }
   la++;
   if (lb>0)    V(lett);
   else    V(mutex)
}
```

Topologia del codice

Indicando con **nX+(t)** il numero di esecuzione dell'istruzione **x++** sino all'istante **t** e te con **nY+(t)** il numero di esecuzione dell'istruzione **y++** sino all'istante **t,** l'invariante **I0** può essere scritto come: $\forall t.\ nX+(t) \leq nY+(t)$

Per la sincronizzazione introduciamo un semaforo **sem** e inseriamo un'operazione **P(sem)** prima di x++ e un'operazione **V(sem)** dopo y++;

```
Process Px {
while (true){
   P(sem)
   x++}
 }
```

```
Process Py {
while (true){
    V(sem)
    y++}
 }
```

Monitors

I monitors sono una combinazione di **tipo di dato astratto** e **mutua esclusione**.

Definizione

- **Costrutto sintattico** che associa un insieme di operazioni ad una **struttura dati comune** a più processi
- Il compilatore può verificare che esse siano le sole operazioni permesse su quella struttura.
- Le operazioni sono **mutuamente esclusive**: un solo monitor alla volta.

```
monitor alpha {
   <dichiarazioni variabili locali>;
   {<inizializzazione delle variabili
 locali>};

   public void op1 ( ) {
    <corpo della operazione op1 >;
   }
       ----------
   public void opn ( ) {
   <corpo della operazione opn>;
   }
}
```

- Le operazioni **public** sono quelle che possono essere utilizzati dai processi per accedere alle variabili comuni.
- Le variabili comuni **mantengono il loro valore** tra successive esecuzioni delle operazioni del monitor (variabili permanenti) e sono accessibili solo entro il monitor.
- Le variabili locali definiscono lo stato della risorsa.
- Le operazioni non dichiarate public non sono accessibili all'esterno.

Il monitor definisce un **tipo (alpha)** e per creare un **istanza** del monitor (tutta la struttura dati) occorre definire una variabile **x** con quel tipo → **alpha x;**

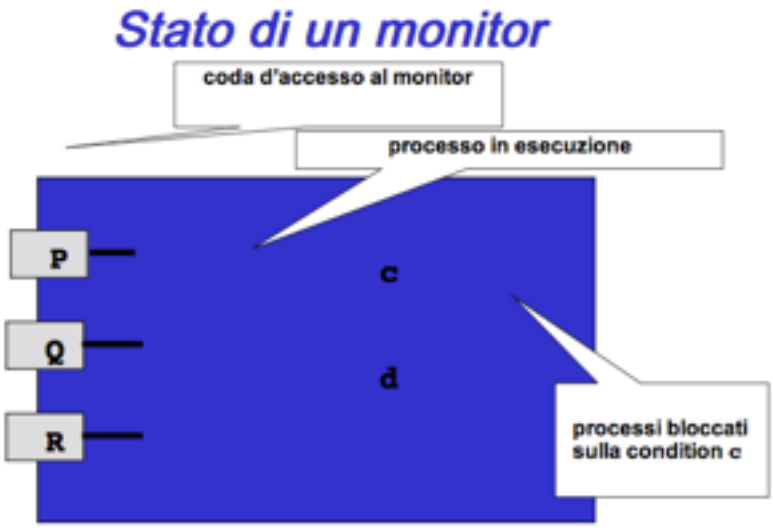
Per chiamare una generica **opi** definita nel monitor → **ris.opi();**

Variabili di tipo condizione

La sospensione di un processo interno di un monitor avviene utilizzando variabili di un nuovo tipo, detto **condition.**

- La dichiarazione di una var **cond** di tipo condizione ha la forma
  → `condition cond`
- Ogni variabile di tipo condizione rappresenta una **coda** nella quale i processi si sospendono
- La procedura nel monitor garantisce su tali variabili mediante le operazioni

→ `wait(cond)` : sospende il processo e lo mette nella coda indicata da cond
→ `signal(cond)` : rende attivo un processo in attesa nella coda indicata da cond

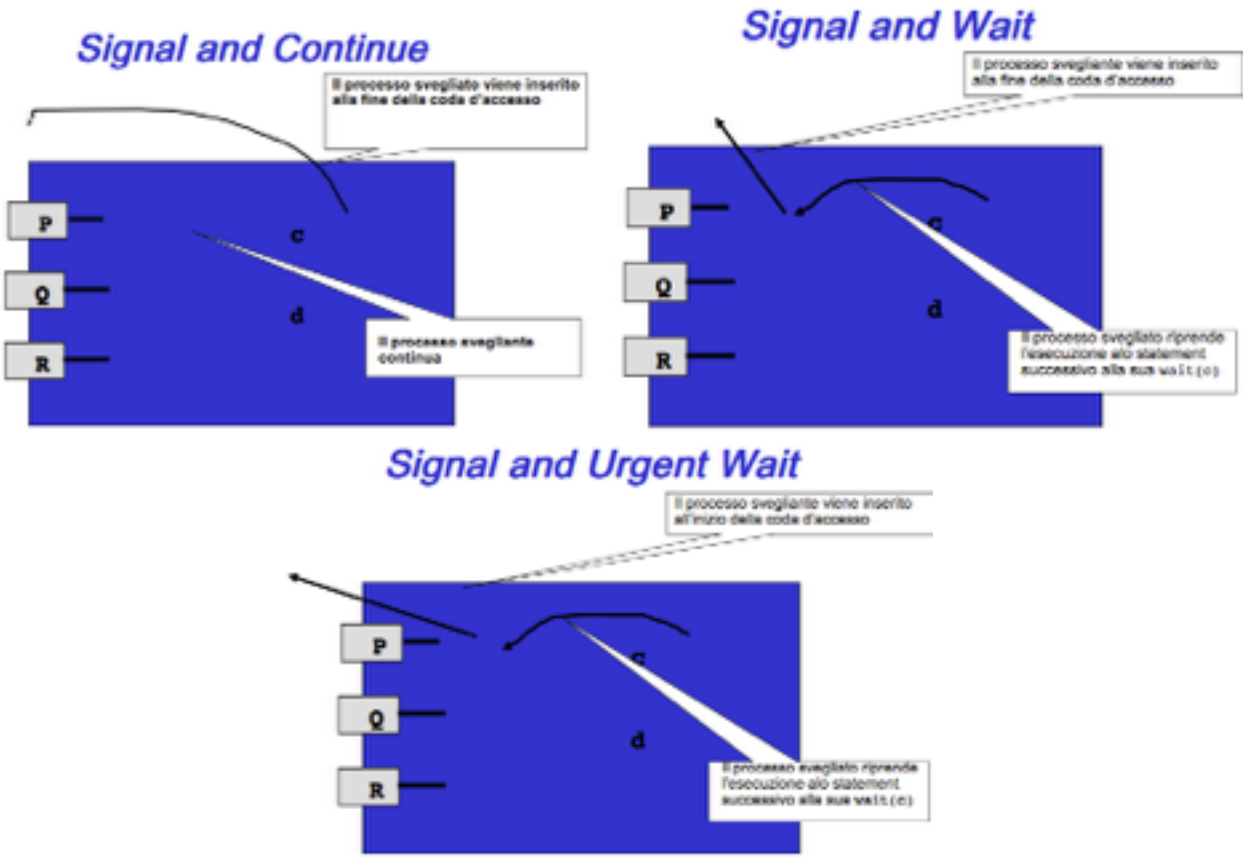


Semantiche dell'operazione signal

signal_and_continue : il processo svegliante continua l'esecuzione, mentre il processo svegliato si pone in attesa di entrare nel monitor

signal_and_wait_: il processo svegliato riprende immediatamente l'esecuzione, mentre il processo svegliante si pone in attesa di rientrare nel monitor

signal_and_urgent_wait : il processo svegliato riprende l'esecuzione, mentre il processo svegliante si pone in attesa di rientrare nel monitor in una coda ad alta priorità (**urgent queue**)



E' possibile usare **array** di variabili condizione. Ogni processo può sospendersi sulla propria variabile condizione (semafori privati).

```
monitor nome_monitor {
      <dichiarazione delle variabili locali>;
      {<inizializzazione delle variabili locali>};
         condition cond [num_max];
             ….
            public void op1 ( ) {
               if(!B) wait(cond[k]);
            ….
            }
            public void op2 ( ) {
               ….
               signal(cond[i]);
            }
 }
```

Esempio produttori - consumatori con buffer limitato

```
monitor buffer_circolare {

        T buffer[N];

        int contatore = 0;
        int testa=0;
        int coda=0;
        condition full;
        condition empty;
        public void inserisci (T m) {

                if (contatore == N) then wait(empty);

                buffer[coda] = m;
                coda = (coda+1)% N;
                contatore ++;

                signal(full);

        }

         public T ricevi ( ) {

                T m;
                if (contatore == 0) then wait(full);  m=buffer [testa];
                testa = (testa+1)% N;
                contatore --;
                signal (empty); return m;

}}
```

Esempio di uso: Allocatore di una risorsa

```
monitor allocatore {
         boolean occupato = false; condition libero;
         public void richiesta ( ) {

         if (occupato) wait (libero);

                occupato=true;

         }

         public void rilascio ( ) { occupato=false; signal (libero);}

}
```

Esempio di uso: realizzazione di una semaforo

```
monitor semaforo {

        int valore = N; condition attesa;

        public void P( ) { if (valore==0) valore--; }

        public void V( ) { valore++; signal (attesa);}

 }
```

Realizzazione del costrutto monitor tramite semafori

Il compilatore assegna ad ogni istanza di un monitor:

- un semaforo **mutex** inizializzato a 1 per la mutua esclusione delle operazioni sul monitor; la richiesta di un processo di utilizzare un'operazione equivale all'esecuzione di una **P(mutex).** Un semaforo **urgent** inizializzato a 0 ed un contatore **urgentcount** per la gestione dei processi sveglianti

Il compilatore assegna per ogni variabile **cond** di tipo **condition** :

- un semaforo **condsem** inizializzato a 0 sul quale il processo si può sospendere tramite un **wait(condsem)**
- un contatore **condcount** inizializzato a 0 per tenere conto dei processi sospesi su **condsem**.

Prologo di ogni funzione ⇒ `P(mutex);`
Epilogo di ogni funzione ⇒ `if(urgentcount>0) V(urgent); else V(mutex);`

`wait(condsem)` ⇒
```
{condcount++;
if(urgentcount>0) V(urgent;
else V(mutex);
P(condsem);
condcount--:
}
```

`signal(condsem)` ⇒
```
if(concount>0){
urgentcount++;
V(condsem);
P(urgent);
urgentcount--;
}
```

Ulteriori operazioni sulle variabili condizione

- **Sospensione** con indicazione della priorità → **wait(cond , p)**

→ i processi sono accodati rispettando il valore (crescente o decrescente) di **p** e vengono risvegliati nello stesso ordine.

- informazioni sullo **stato** della coda → **empty (cond)**
  → operazione che fornisce il valore **false** se esistono processi sospesi nella coda associata e **cond , true** altrimenti.

- **Risveglio** di tutti i processi sospesi sulla variabile condizione → **signal_all(cond)**
  → si può usare solo con la semantica *signal_and_continue.*

Allocazione di una risorsa mediante la strategia Shortest-job-next

Più processi competono per l'uso di una risorsa. Quando questa viene rilasciata, essa viene assegnata, tra tutti i processi sospesi, a quello che la **userà per il periodo di tempo inferiore.**

```
monitor allocatore{
   boolean occupata = false;
   condition non_occupata;
   public void richiesta(int tempo){
     if (occupata) wait(non_occupata,tempo);
         occupata = true;
   }
   public void rilascio(){
     occupata = false;
     signal(non_occupata);
   }
}
```

Gestione di un disco a teste mobili SCAN

```
typedef int traccia;
typedef enum (SU, GIU) dir;

monitor movimento_braccio {

       traccia pos = 0;
       boolean occupato = false;
       dir direzione = SU;
       condition dir_SU, dir_GIU;

       public void richiesta(traccia dest){
       // se il braccio è libero servo subito la richiesta se il braccio è occupato
             if (occupato) {
             // servo la richiesta quando torno in direzione SU
                   if (pos < dest || (pos == dest && direzione == GIU))
                         wait (dir_SU, dest);
                   else wait(dir_GIU, (N – dest));
             }
             occupato = true;
             pos = dest;
}
```

```
public void rilascio{
// viene eseguita quando ho finito di servire la richiesta e rilascio la testina libera
	occupato = false;
	if (direzione == SU) {
	if (!empty(dir_SU)
		signal(dir_SU);
	else {
		direzione = GIU;  // se non ci richieste cambio direzione
		signal (dir_GIU);
	}else if (!empty(dir_GIU))
				signal(dir_GIU)
			else
				direzione = SU; // se non ci richieste cambio direzione
}
```

Monitor: Lettori - Scrittori

```
monitor lettori_scrittori {
	int num_lett = 0;
	boolean occupato = false;
	condition ok_scrivi,ok_leggi;

	public void inizio_lettura {
		if(occupato OR !empty(ok_scrivi))
			wait(ok_leggi);
		num_lett++;
		signal(ok_leggi);
	}

	public void fine_lettura {
		num_lett--;
		if(num_lett == 0)
			signal(ok_scrivi);
	}

	public void inizio_scrittura {
		if(num_lett ≠ 0 OR occupato)
			wait(ok_scrivi);
		occupato = true;
	}

	public void fine_scrittura {
		occupato = false;
		if(!empty(ok_leggi))
			signal(ok_leggi);
		else signal(ok_scrivi)
	}
}
```

Prova di correttezza

La prova di correttezza di un monitor si ottiene mediante **invarianti di monitor,** cioè relazioni che rimangono valide prima e dopo ogni chiamata di procedura del monitor. Per dimostrare che una relazione è un invariante di monitor è sufficiente verificare che:

a) la relazione è vera al momento dell'inizializzazione del monitor
b) supposta vera al momento della chiamata di una procedura di monitor, è vera anche alla fine dell'esecuzione della stessa:
- occorre porre attenzione alla semantica della **signal(cond)** per stabilire quando è terminata l'esecuzione della procedura monitor.

$$(R > 0 \land W = 0) \lor (W \neq 1) \lor (W = 1 \land R = 0)$$

Lemma.
E' facile provare che le seguenti formule sono invarianti:

$$R \geq 0$$
$$W \geq 0$$
$$R = num\_lett$$
$$W = 0 \lor occupato = false$$

Prova:

a) all'inizio è chiaramente vera
b) supposta vera all'inizio di ciascuna delle quattro procedure di monitor, è vera anche alla fine

Inizio lettura:

```
public void
inizio_lettura     {
if(occupato.OR.
     !empty(ok_scrivi))
 wait(ok_leggi);
   num_lett++;
   signal(ok_leggi);
}
```

caso 1) la procedura termina senza svegliare un altro lettore

La procedura incrementa **R** e potrebbe rendere falsa

la relazione: $(R > 0 \land W = 0)$

Ma lo statement **if** ci assicura che la procedura viene completata solo se **occupato= false** e quindi **W = 0.**
Inoltre la procedura potrebbe rendere falsa la relazione $(W = 1 \land R = 0)$, ma nuovamente

lo statement **if** ci assicura che la procedura viene completata solo se **W = 0**.

Caso 2) La procedura sveglia un'altro processo lettore.

La chiamata della primitiva **signal (ok_leggi)** può svegliare un lettore **r** incrementando **R** , ma lo statement **if** assicura che ciò accade solo se **occupato = false** e quindi **W = 0.**
La semantica *signal-and-urgent-wait* ci assicura che **r** continua immediatamente quindi rimane vero che **W = 0.**

fine_lettura:

```
public void fine_lettura {
   num_lett--;
   if(num_lett==0)
 signal(ok_scrivi);
}
```

caso 1) Supponiamo che la procedura termini senza svegliare un processo scrittore.

La procedura decrementa **R** e potrebbe rendere falsa la relazione (R > 0) e rendere vera la relazione (R = 0) ma questo non cambia l'invariante.

caso 2) La procedura sveglia un processo scrittore.

La **signal** è eseguita se **(R = 0)** quindi la prima e la terza delle sotto-formule rimangono valide. Inoltre al momento della chiamata della procedura **fine_lettura R** doveva valere 1, per cui **(W = 0).**
La semantica *signal-and-urgent-wait* ci assicura che lo scrittore risvegliato continua immediatamente l'esecuzione, mantenendo quindi **(W = 1)**

inizio_scrittura:

```
public void
inizio_scrittura    {
if(num_lett!=0.OR.occupato
)    wait(ok_scrivi);
occupato=true;
}
```

La procedura incrementa **W** e potrebbe rendere falsa la relazione $(W \square 1)$, ma lo statement **if** ci assicura che la procedura viene completata solo se **occupato = false** e quindi **W = 0.**

Inoltre la procedura potrebbe rendere falsa la relazione $(R > 0 \square W = 0)$, ma nuovamente lo statement **if** ci assicura che la procedura viene completata solo se **R = 0.**

fine_scrittura:

```
public void
fine_scrittura{
occupato=false;
if(!empty(ok_leggi))
 signal(ok_leggi);
else
 signal(ok_scrivi)
   }
}
```

caso 1) La procedura termina senza svegliare altri processi.

La procedura decrementa **W** ma le modifiche di **W** non falsificano l'invariante.

Caso 2) La procedura sveglia un processo lettore

Al momento della chiamata della procedura, per ipotesi induttiva, $(W \square 1)$, quindi l'esecuzione di fine_scrittura porta **W = 0,** il risveglio del lettore fatta dalla **signal** conserva la verità dell'invariante.

caso 3) La procedura sveglia un processo scrittore

Per ipotesi induttive $(W \square 1)$ quindi lo scrittore che esegue è l'unico scrittore, pertanto l'esecuzione della **signal(ok_scrivi)** comporta **W = 1.**

Realizzare monitor in Java

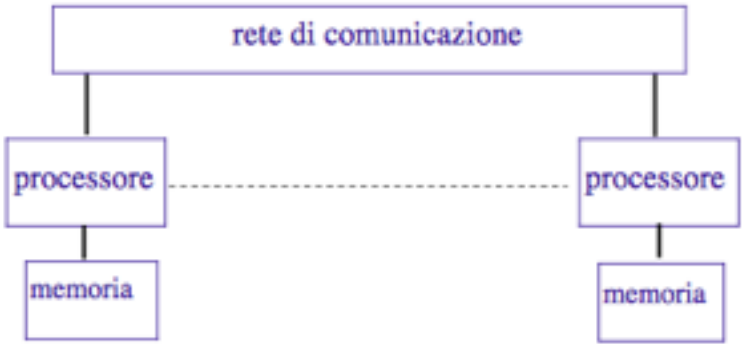
I thread di una applicazione condividono lo spazio di indirizzamento, ogni tipo di interazione tra thread avviene tramite oggetti comuni.

- Interazione di tipo competitivo (mutex) : meccaniscmo **objects lock**
- Interazione di tipo cooperativo (scambio informazioni): **wait - notify**

[codice su slide ma si può evitare… notifyAll]

Modello a scambio di messaggi

Modello architetturale di macchina (virtuale) concorrente.



Aspetti caratterizzanti

- Ogni processo può accedere esclusivamente alle risorse allocate nella propria memoria (virtuale) locale.
- ogni risorsa del sistema è accessibile ad un solo processo
- se una risorsa è necessaria a piu processi applicativi, ciascuno di questi, **processi clienti,** dovrà richiedere all'unico processo che può operare sulla risorsa (**processo server**) di eseguire sulla stessa l'operazione richiesta e restituirgli i risultati dell'operaz.
- in questo modello architetturale di macchina concorrente il concetto di **gestore** di una risorsa coincide con quello di processo **server.**
- cambia quindi il paradigma di programmazione utilizzato per consentire ad un processo di usufruire dei servizi offerti da una risorsa
- il meccanismo di base utilizzato dai processi per qualche tipo di interazione è costituito dal **meccanismo di scambio di messaggi**

Canali di comunicazione

Il concetto che entra in gioco nel modello a scambio di messaggi è quello di **canale** inteso genericamente come collegamento logico mediante il quale due processi comunicano.

E' compito del nucleo del sistema operativo fornire l'astrazione **canale** come meccanismo primitivo per lo scambio di informazioni.

E' compito di un **linguaggio di programmazione** che utilizza il modello a scambio di messaggi, offrire gli strumenti linguistici di alto livello per consentire al programmatore di specificare canali di comunicazione e di utilizzarli per programmare le varie interazioni tra i processi dell'applicazione.

Parametri caratterizzanti il canale:

- la tipologia del canale, intesa come direzione del flusso di dei dati che un canale può trasferire
- la designazione del canale e dei processi sorgente e destinatario di ogni comunicazione
- il tipo di sincronizzazione fra i processi comunicanti

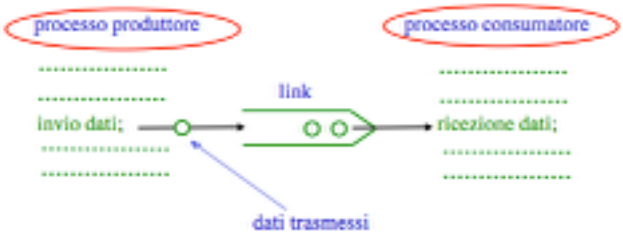
Esistono due tipi di canali con riferimento alla direzione del flusso:

1) **Monodirezionali**
2) **Bidirezionali**

Tipi di canale con riferimento alla designazione dei processi comunicanti:

- **link** canale simmetrico
  - da uno a uno
- **port** canale asimmetrico
  - da molti a uno
- **mailbox** canale asimmetrico
  - da molti a molti

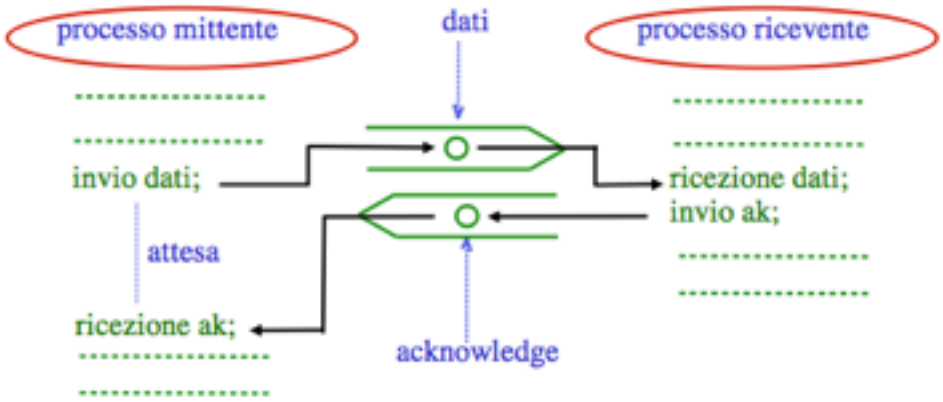
Simmetrica: Produttore - Consumatore **Asimmetrica: Cliente - servitore**



Tipologia di sincronizzazione

- Comunicazione **ASINCRONA**
- Comunicazione **SINCRONA**
- Comunicazione con **SINCRONIZZAZIONE ESTESA**

Comunicazione sincrona mediante comunicazioni asincrone



Schema relativo ad una chiamata di procedura remota

Primitive di comunicazione

Dichiarazione di canale:

```
port < tipo > < identificatore >;
```

Es:
```
port int ch1;
```

L'identificatore **ch1** denota un canale utilizzato per trasferire messaggi di tipo **integer**. Un canale asimmetrico **da-molti-a-uno.** Viene dichiarato locale a un processo (il ricevente) ed è visibile ai processi mittenti mediante la *dot notation.*

Primitive di invio

```
send ( < valore > ) to porta;
```

`< porta >` identifica in modo univo il canale a cui inviare il messaggio, normalmente espresso come

```
< nome processo > . < nome locale della porta >
```

Esempio:

```
send ( 125 ) to P.ch1;
```

Il processo che la esegue invia il valore **125** al processo **P** tramite il canale **ch1** da cui solo **P** può ricevere.

Primitive di ricezione

```
receive ( < variabile > ) from porta;
```

`< porta >` identifica il canale, locale al processo ricevente, dal quale ricevere il msg.
`< variabile >` è l'identificatore di una variabile dello stesso tipo di `< porta >` a cui assegnare il valore del messaggio ricevuto.
La primitiva sospende il processo se non ci sono messaggi sul canale e restituisce un valore del tipo predefinito **process** che identifica il nome del processo mittente.

Esempio:

```
proc = receive (m) from ch1;
```

Il processo che la esegue si sospende se sul proprio canale **ch1** non ci sono messaggi altrimenti entra il primo di loro e ne assegna il valore a **m** , quindi assegna alla variabile **proc,** di tipo **process,** il nome del processo che ha inviato il messaggio.

Comandi con guardia

```
< guardia > → < istruzione >;
```

`< guardia >` è costituita da una coppia:

```
(< espressione booleana >); < primitiva receive >
```

Una guardia viene valutata e può fornire tre diversi valori:

- **guardia fallita** : se l'espressione booleana ha il valore **false**
- **guardia ritardata** : se l'espressione booleana ha valore **true** ma la **receive** è bloccante poiché sul canale su cui viene eseguita non ci sono messaggi pronti
- **guardi valida:** se l'espressione booleana ha valore **true** e la **receive** può essere eseguita senza ritardi

Esempio:

```
if
     < guardia_1 > → < istruzione_1 >;
     .
     .
     < guardia_n > → < istruzione_n >;
fi
```

- Vengono valutate le guardie di tutti i rami;
- se una o più guardie sono **valide** viene scelto, in maniera non deterministica, uno dei rami con guardia **valida** e la relativa guardia viene eseguita; viene quindi eseguita l'istruzione relativa al ramo scelto e con ciò termina l'esecuzione dell'intero comando alternativo.
- se tutte le guardie non valide sono **ritardate,** il processo in esecuzione si sospende in attesa che arrivi un messaggio che abilita la transizione di una guardia da **ritardata a validata** a quel punto procede come nel caso precedente
- se tutte le guardie sono fallite l'esecuzione del comando termina

Esempio di processo server

```
process server {

      port int servizioA;  // escluzione di A() su R
      port real servizioB; // esecuzione di B() su R

      tipo_di_R R;
      int x;
      real y;

      do
            [] (condA); receive (x) from servizioA; →  {

                  R.A(x);
                  < eventuale restituzione dei risultati al cliente >;
            }
            [] (condB); receive (y) from servizioB; → {

                  R.B(y);
                  < eventuale restituzione dei risultati al cliente >;
            }

}
```

Primitive asincrone

- comunicazione **asimmetrica**
- send **non bloccante**
- receive **bloccante**
- definizione **canali mediante costrutto** port

```
port < tipo > < identificatore >;
send (< valore >) to porta;
receive (< variabile >) from < porta >;
```

Client - Server : Esempio 1

Processo servitore che offre un unico servizio senza condizioni di sincronizzazione:

```
process cliente {
port tipo_out risposta;
tipo_in  a;
tipo_out  b;
process p;
  ……
  ……
send(a)to server.input;
p=receive(b)from risposta;
            ……
}
```

```
tipo_out fun(tipo_in x);
process  server {
 port tipo_in input;
 tipo_var  var;
 process p;
 tipo_in  x;
 tipo_out y;
 {< eventuale
 inizializzazione>;}
 while(true){
    p=receive(x)from input;
    y =fun(x);
    send(y)to p.risposta;
 }
}
```

Client - Server : Esempio 2

Processo servitore **con più servizi** (due) senza condizioni di sincronizzazione (**comandi con guardia**):

```
tipo_out1 fun1(tipo_in1 x1);
tipo_out2 fun2(tipo_in2 x2);
process  server {
  port tipo_in1   input1;
  port tipo_in2   input2;
  tipo_var  var;
  process  p;
  tipo_in1  x1;  tipo_in2  x2;
  tipo_out1  y1;  tipo_out2  y2;
  {< eventuale istruzione di
inizializzazione>;}
  do
     p = receive (x1) from input1; ->
           y1=fun1(x1);
           send (y1) to p.rispostal;
   □ p = receive (x2) from input2; ->
          y2=fun1(x2);
          send (y2) to p.risposta2;
   od;
}
```

Client - Server : Esercizio 3

Processo servitore **con più servizi** (due) e specifica condizioni di sincronizzazione (**comandi con guardia**):

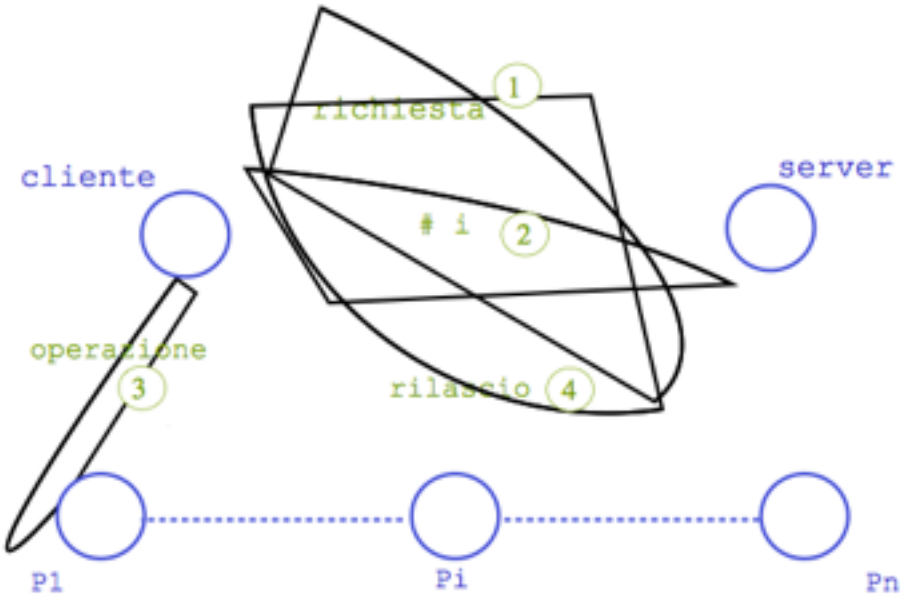
```
tipo_out1 fun1(tipo_in1 x1);
tipo_out2 fun2(tipo_in2 x2);
process  server {
  port tipo_in1   input1;
  port tipo_in2   input2;
  tipo_var  var;
  process  p;
  tipo_in1  x1;  tipo_in2  x2;
  tipo_out1  y1;  tipo_out2  y2;
  {< eventuale istruzione di
inizializzazione>;}
  do
    □ (cond1); p = receive (x1) from input1; ->
            y1=fun1(x1);
            send (y1) to p.risposta1;
    □ (cond2); p = receive (x2) from input2; ->
           y2=fun1(x2);
           send (y2) to p.risposta2;
    od;
}
```

Corrispondenza tra monitor e Client - Server [NON SI CAPISCE DALLE SLIDE]

Monitor	Client - Server
Risorsa condivisa: istanza di un monitor locale a un processo server	Risorsa condivisa: struttura dati locale a un processo server
identificatore di funzione di accesso al monitor	porta del processo server
tipo dei parametri della funzione	tipo della porta
Tipo del valore restituito dalla funzione	tipo della porta da cui il processo cliente riceve il risultato
per ogni funzione del monitor	un ramo (con guardia) dell'istruzione rispettiva che costruisce il corpo del server
condizione di sincronizzazione di una funzione	Espressione logica componente la guardia della corrispondente alla funzione
chiamata : **server** seguito da attesa dei risultati sulla propria porta	invio della richiesta sulla corrispondente porta del ???
esecuzione in mutex fra le chiamate alle funzioni del monitor server	scelta di uno dei rami con guardia valida del comando rispettivo del ???

Monitor	Client - Server
corpo della funzione	istruzione del ramo corrispondente alla funzione
relazione invariante del monitor rispettiva del server	relazione invariante dell'istruzione

Gestore di un pool di risorse equivalenti



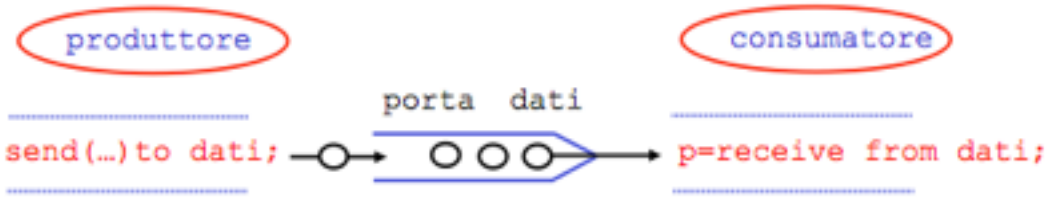

```
process server{
   port signal richiesta;
   port int rilascio;
   int disponibili=N;
   boolean  libera[N];
   process p ;
   signal s;
   int  r;
   for     (int     i=0;     i<N;     i++)libera[i]=true;
   //inizializzazione
   do
    (disponibili>0); p= receive(s)from richiesta; ->
int i=0;
while (!libera[i]) i++;
libera[i] = false;
disponibili--;
                  send (i) to p.risorsa;
    p=receive(r)from rilascio; ->
                  disponibili++;
                  libera[r] = true;
    od;
}
```

Client - Server

Simulazione di un semaforo

```
process semaphore{
    port signal P;
    port signal V;
    int  valore = i;
    process proc ;
    signal s;
    do
 (valore>0); p=receive(s)from P; ->
     valore--;
     send (s)to proc.risposta;}
    p=receive (s) from V; ->
          valore++;
    od;
}
```

Scambio di dati produttore - consumatore



Scambio di dati singolo produttore - più consumatori


```
process smistatore{
     port T dati;
     port signal pronto;
     T messaggio;
     process prod, cons; signal s;
     while (true) {
            cons =
              receive(s) from pronto;

            prod =
               receive(messaggio)
                          from dati;
            send(messaggio)
                   to cons.dati;
     }
}
```

Più produttori - Più consumatori [buffer limitato]


```
process smistatore{
    port T dati;
    port signal pronto_prod, pronto_cons;
    T messaggio;
    process prod, cons;
    signal s;
    int contatore=0;
    do
     (contatore<N); prod=receive(s)from pronto_prod; ->
       contatore ++;
       send(s)to prod.ok_to_send;
     □ (contatore>0); cons:=receive(s)from pronto_cons; ->
          prod=receive(messaggio)from dati;
          contatore--;
          send(messaggio)to cons.dati;
  od;
}
```

```
  process produttorei{
       port signal ok_to_send;
       T messaggio;
       process p;
       signal s;
       ……
       <produci il messaggio>;
          send(s)to smistatore.pronto_prod;
p = receive(s)from ok_to_send;
send(messaggio)to smistatore.dati;
  ……
}
```

```
  process consumatorej{
       port T dati;
       T messaggio;
       process p;
       signal s;
       ……
       send(s)to smistatore.pronto_cons;
     p = receive(messaggio)from dati;
<consuma il messaggio>;
  ……
}
```

Specifica di strategia di priorità

```
process server{
   port signal richiesta;
   port int rilascio;
   int avail = N;
   boolean libera[N];
   process p;
   signal s;
   int r;
   int sospesi=0;
   boolean bloccato [M];
   process client[M];
    for(int i=0;i<N;i++)
          libera[i] = true;
    for(int j=0;j<M;j++)
          bloccato[i] = false;
    client[0]=P0;….
    client[M-1]=PM-1
```

```
do
p=receive(s) from richiesta;
if(avail>0){
  inti=0;
 while(!libera[i])i++;
 libera[i]=false;
 avail--;
 send(i)top.risorsa;}
 else{intj=o;
 sospesi++;
 while(client[j]!=p)j++;
 bloccato[j]=true;}
p=receive(r) from rilascio;
if(sospesi==0){
  avail++;
 libera[r]=true;
 else{inti=o;
 while(!bloccato[i])i++;
 sospesi--;
  bloccato[j]=false;
  send(r)toclient[i].risorsa}
od;
```

Realizzazione primitive asincrone in ambiente mono / multi - processore

Ipotesi:

- Tutti i messaggi sono di un unico tipo **T** predefinito
- Ad ogni processo è associato un insieme di porte (molte - a- uno) mantenute in un vett
- Ad ogni porta è associato un buffer di lunghezza indefinita in modo che non possa mai essere pieno
- I processi vengono identificati dal loro PID
- Le porte vengono identificate da un intero che rappresenta l'indice nel vettore delle porte locali al processo

Funzioni del kernel:

- `void send (T inf, PID proc, int ip)`
- `void receive ( T&inf, PID&proc, int ip)`

Primitive per operare sulle porte

```
boolean  coda_vuota (coda_di_messaggi  c)
{
   if (c.primo == null) return true;
   return false;
}
```

```
typedef struct {
                coda_di_messaggi
     coda;
       p_porta  puntatore;
} des_porta;
```

```
typedef des_porta * p_porta
```

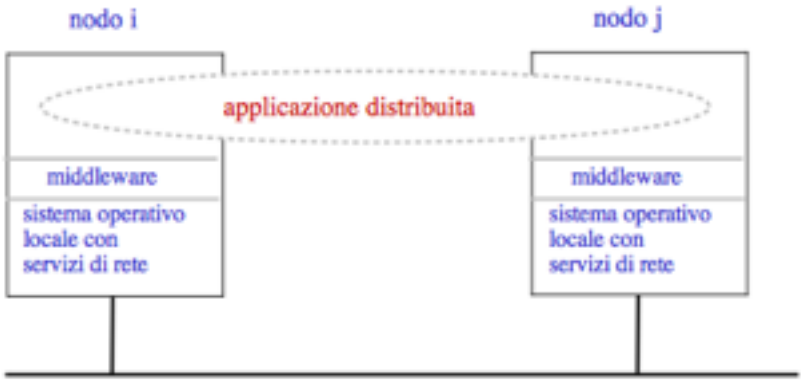
Descrittore di processi

- Oltre ai consueti campi nel descrittore di un processo, è presente un campo **porta_processi** costruita da un vettore di M identificatori di porte
- il campo **stato** di un processo è costituito da un **vettore di M bit** ciascuno dei quali è univocamente associato ad una della **M** porte del processo; il valore 1 di uno di questi bit significa che il processo è in attesa di un messaggio sulla porta corrispondente.

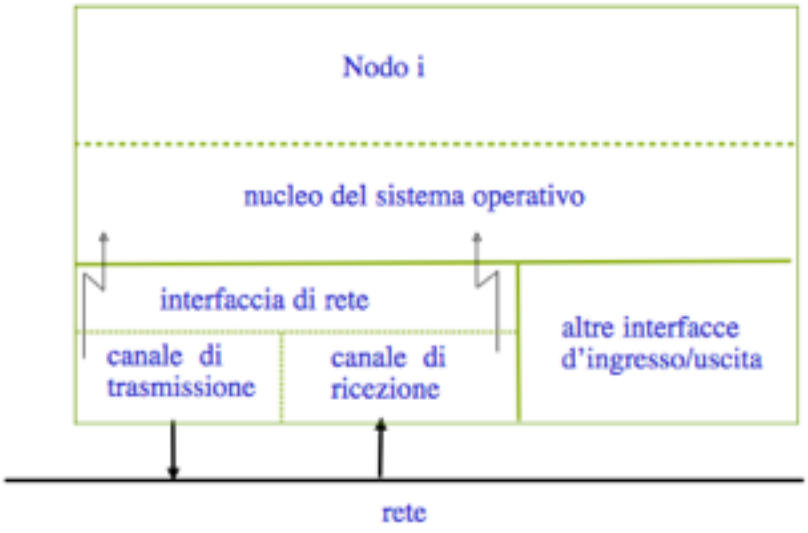
Primitive per operare sullo stato dei processi

- `boolean bloccato_su ( p-des p, int ip )`
- `void blocca_su ( int ip)`

Architetture distribuite Network Operating System



Architetture distribuite Interfaccia di rete



```
    void invia_pacchetto (packet p) {
if  (<canale di trasmissione occupato>)
 packet_queue.inserisci(p);
 else {
      <inserimento di p nel registro buffer del canale>;
      <attivazione trasmissione>;  }
 }
```

```
    void tx_interrupt_handler() {
        packet p;
        salvataggio_stato();
        if (!packet_queue.vuota()){
            p = packet_queue.estrai();
   <inserimento di p nel registro buffer del canale>;
            <attivazione trasmissione>;  }
        ripristino_stato();
 }
```

```
typedef struct {
        int indice-nodo;
        int PID_locale;
} PID;
```

```
void send (T inf, PID proc, int ip){
          if (proc.indice_nodo == nome_nodo)
               local_send(inf, proc, ip);
   else remote_send(inf, proc, ip);
}
```

```
void remote_send (T inf, PID proc, int ip){
          packet p;
          PID mit=PID();
          indice_nodo_destinatario= proc.indice_nodo;
           <vengono riempiti i vari campi del  pacchetto p: in particolare, viene
         inserito nel campo relativo al nodo a cui inviare il pacchetto il valore
         indice_nodo_destinatario. Inoltre,  nel  campo del pacchetto destinato
         a contenere le informazioni da inviare vengono inseriti il nome  mit  del processo
         che invia, e i tre parametri della funzione  inf,  proc,  e  ip>;
             invia_pacchetto(p);
}
```

```
void rx_interrupt_handler() {
  packet p;
  PID mit; T inf; PID proc; int ip;
  salvataggio_stato();
  <assegnamento a p del pacchetto ricevuto presente nel buffer del canale>;
  <attivazione ricezione>;
  <estrazione dal  campo del pacchetto p, contenente le informazioni
   ricevute, del nome  mit  del mittente e dei tre parametri della funzione
   send  inf,  proc,  e  ip>;
  messaggio * m=new messaggio;
  m -> informazione = inf;
  m -> mittente = mit;
  inserisci_porta(m, proc, ip);
  ripristino_stato();
}
```

Esercizi

Introduzione alla concorrenza

Esercizio 1

```
int x = 1
cobegin
      S1:    x := x + 5
      S2:    x := x * 3.
coend
```

Si diano tutti i possibili valori di **x** alla fine dell'esecuzione del codice sopra riportato.
Si discuta il problema dell'atomicità.

Soluzione.

x può assumere i seguenti valori :

- 6 → perché viene eseguita la load di 1 contemporaneamente quindi entrambi han x = 1 e l'ultimo che fa la scrittura sarà S1
- 3 → perché viene eseguita la load di 1 contemporaneamente quindi entrambi han x = 1 e l'ultimo che fa la scrittura sarà S2
- 18 → come se venisse eseguito prima S1 poi S2
- 8 → come se venisse eseguito prima S2 poi S1

Sia la moltiplicazione che l'addizioni non sono operazioni atomiche ma sono (in assembler) rappresentate da 3 operazioni (load, add/mult, store) infatti S2 può fare la load del valore di x prima o dopo che S1 ha fatto la store, e da questo ne dipende il risultato.

Esercizio 2

Si consideri l'algoritmo seguente:

Concurrent Algorithm	
integer n ←0	
P	q
p1: while (n < 2) p2: write (n)	q1: n ←n + 1 q2: n ←n + 1

a) costruire uno scenario che dia in output la sequenza 002;
b) quante volte il valore 2 può apparire in output;
c) quante volte il valore 1 può apparire in output.

Soluzione.

a) **p1 p2 p1 p2 p1 q1 q2 p2**
b) **1**
c) **infinite** (perchè posso fare q1 p1 p2 p1 p2 p1 p2 p1 p2 p1 …….)

Esercizio 3

Si consideri l'algoritmo seguente

Stop the Loop	
integer n ¬0 boolean flag ¬ false	
P	**q**
p1: while (flag == false) p2: n ¬1 - n	q1: while (flag ==false) q2: if (n == 0) q3: falg ¬ true

a) si costruisca uno scenario per cui il programma termina
b) quali sono i possibili valori di n per cui il programma termina?
c) il programma termina per tutti gli scenari?

Soluzione.

a) **q1 q2 p1 p2 q3**
b) **0, 1**
c) **q1 q2 q3** → sei fottuto!

Esercizio 4

Si supponga che la funzione f assuma valore 0 per un valore intero i: f(i) = 0. I cinque algoritmi concorrenti cercano il valore i. Un algoritmo viene detto corretto se per ogni scenario, entrambi i processi terminano dopo che è stato trovato il valore i. Per ciascuno degli algoritmi si dica se è corretto, motivando la risposta (in caso di risposta negativa fornire un controesempio).

Algoritmo1: ZeroA	
boolean found	
P	**q**
integer i ¬ 0 p1: found ¬ false p2: while (not found) p3: i ¬ i+1 p4: found ¬ f(i) = 0	integer j ¬ 1 q1: found ¬ false q2: while (not found) q3: j ¬ j -1 q4: found ¬ f(j) = 0

Algoritmo1: ZeroB	
boolean found ¬false	
P	**q**
integer i ¬ 0 p1: while (not found) p2: i ¬ i+1 p3: found ¬ f(i) = 0	integer j ¬ 1 q1: while (not found) q2: j ¬ j -1 q3: found ¬ f(j) = 0

Algoritmo1: ZeroC	
boolean found ¬false	
P	**q**
integer i ¬ 0 p1: while (not found) p2: i ¬ i+1 p3: if f(i) = 0 p4: found ¬ true	integer j ¬ 1 q1: while (not found) q2: j ¬ j -1 q3: if f(j) = 0 q4: found ¬ true

Algoritmo1: ZeroD	
boolean found ¬false integer turn ¬ 1	
P	**q**
integer i ¬ 0 p1: while (not found) p2: await turn ==1 turn ¬ 2 p3: i ¬ i+1 p4: if f(i) = 0 p5: found ¬ true	integer j ¬ 1 q1: while (not found) q2: await turn ==2 turn ¬ 1 q3: j ¬ j-1 q4: if f(j) = 0 q5: found ¬ true

Algoritmo1: ZeroE	
boolean found ¬false integer turn ¬ 1	
P	**q**
integer i ¬ 0 p1: while (not found) p2: await turn ==1 turn ¬ 2 p3: i ¬ i+1 p4: if f(i) = 0 p5: found ¬ true p6: turn ¬ 2	integer j ¬ 1 q1: while (not found) q2: await turn ==2 turn ¬ 1 q3: j ¬ j-1 q4: if f(j) = 0 q5: found ¬ true q6: turn ¬ 1

Soluzione.

Zero A) Errato: Può capitare che f(i) ritorni True in uno dei due processi, ma poco dopo l'assegnamento a found viene assegnato False dal secondo processo che esegue f(i) e quindi si va in loop perchè il valore è stato trovato ma i due cicli non se ne accorgono.

Zero B) Errato: identico a prima anche se la variabile viene inizializzata precedentemente.

Zero C) Esatto: Funziona perchè f(i) ritorna un valore ma questo verrà perso perchè viene solamente usato per la if che quindi se sarà true found verra impostata a True altrimenti no! Quindi il problema di prima è evitato perchè found non può andare a true e poi tornare a false

Zero D) Esatto: il semaforo serve semplicemente a far eseguire prima P poi Q ma il programma termina ed è corretto,il codice è **ZeroC**

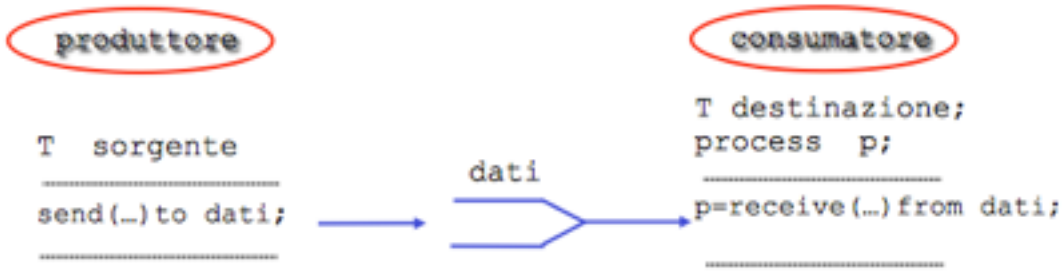
Zero E) Esatto: identico a prima, alla fine viene impostato turno = 2 o =1 ma è "inutile"

Primitive sincrone

- **send** → Bloccante
- **receive** → Bloccante
- Comunicazione **asimmetrica**
- Non esistono **code di messaggi** associati ai processi gestiti dal meccanismo di comunicazione
- **Minore grado di concorrenza**

```
                send (<valore>) to <porta>
<variabile> = receive (<variabile>) from <porta>
```

Scambio di dati: singolo produttore - singolo consumatore



Guardie di input

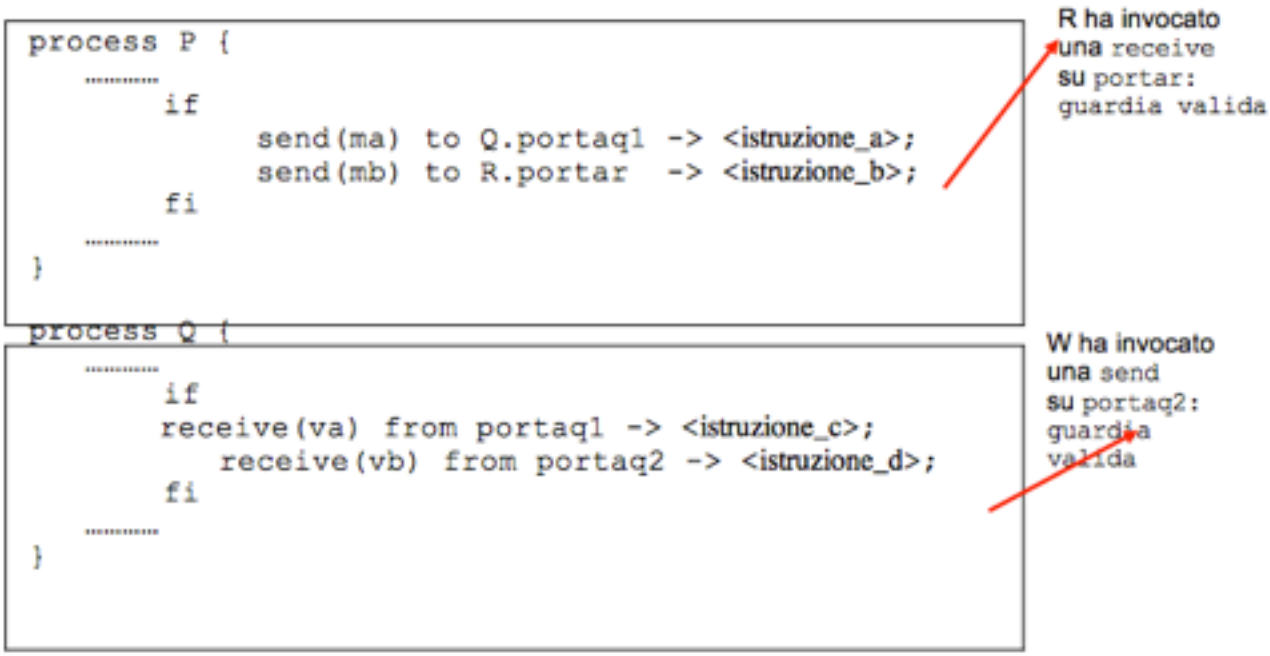
Guardie potrebbero avere anche la seguente forma:

(< espressione booleana >): <primitiva **send** >

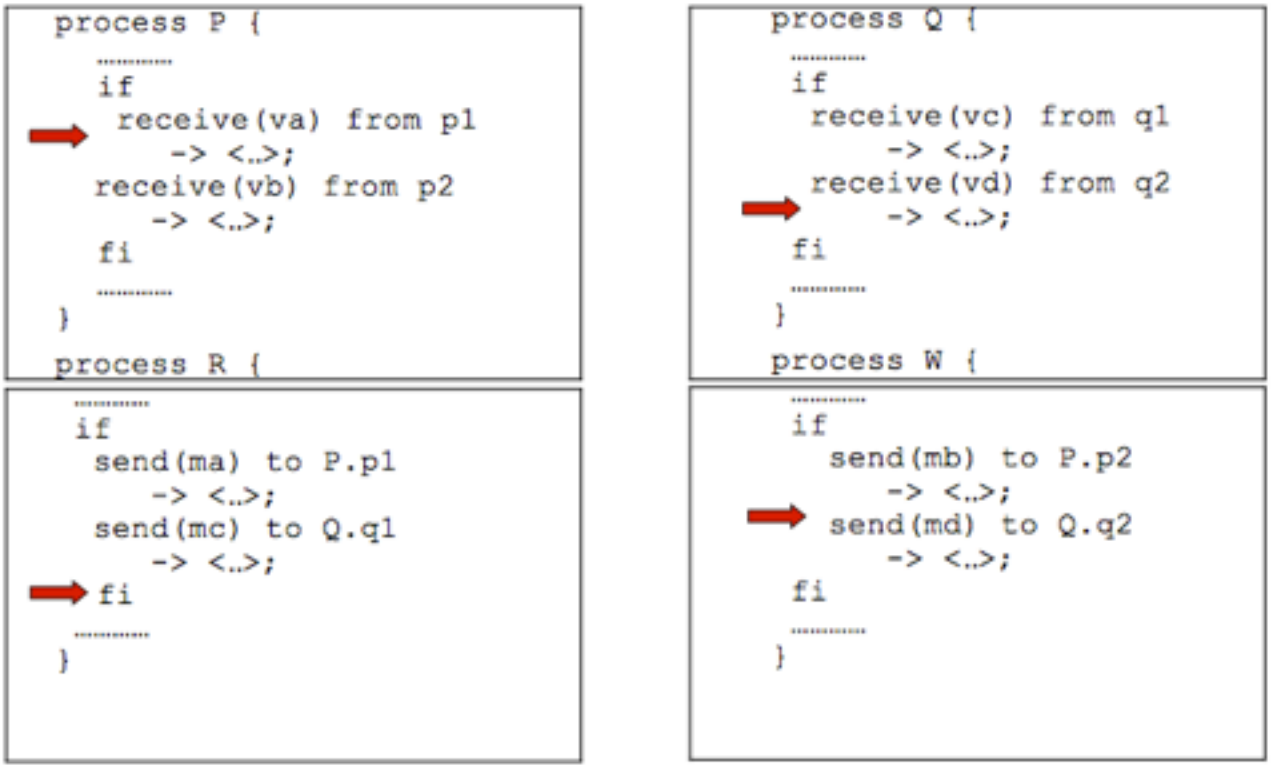
```
.......
      if
            send (mes) to proc1.pr → < istruzione_1 >;
            .
            .
            .
            send (mes) to procN.pr → < istruzione_N >;

      fi
```

Difficoltà di uso di guardie di input e guardie di output



Possibile condizione di stato



Processi servitori: Gestore di un pool di risorse equivalenti

```
process client{
   port int risorsa;
   int  r;
   process p ;
   signal s;
   . . .
   send(s) to server.richiesta;
   p = receive(r) from risorsa;
    <uso risorsa r>
   send(r) to server.rilascio;
   . . .
}
```

```
process server{
   port signal richiesta;
   port int rilascio;
   int disponibili=N;
   boolean  libera[N];
   process p ;
   signal s;
   int  r;
   for     (int     i=0;     i<N;     i++)libera[i]=true;
   //inizializzazione
   do
    (disponibili>0); p= receive(s)from richiesta; ->
int i=0;
while (!libera[i]) i++;
libera[i] = false;
disponibili--;
                  send (i) to p.risorsa;
    p=receive(r)from rilascio; ->
                  disponibili++;
                  libera[r] = true;
    od;
}
```

Processi servitori: Gestore di una risorsa dedicata

```
process client{
   port int risorsa;
   signal s;
   . . .
   send(s)                    to
   server.richiesta;
 <uso la risorsa >
   send(s) to server.rilascio;
   . . .
}
```