
ANNO ACCADEMICO 2024/2025

Modelli Concorrenti e Algoritmi Distribuiti

Teoria

Altair's Notes



DIPARTIMENTO DI INFORMATICA

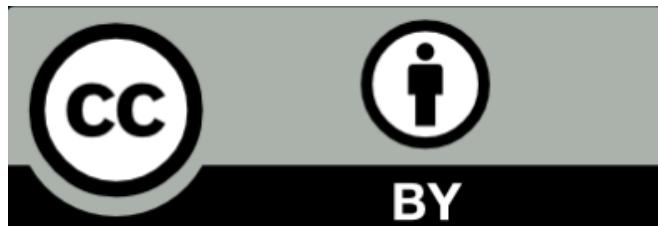
CAPITOLO 1	INTRODUZIONE	PAGINA 5
1.1	Il corso in breve... Cosa si intende per programmazione concorrente? — 5 • Cosa si intende per algoritmo distribuito? — 6	5
CAPITOLO 2	PROGRAMMAZIONE CONCORRENTE	PAGINA 8
2.1	Parallelismo Interleaving d'Istruzioni Atomiche — 8 • Sistemi Monoprocesso e Sistemi Multiprocesso — 11	8
2.2	Correttezza di Programmi Concorrenti Introduzione e Proprietà — 15 • La Correttezza di Programmi — 17 • Il Problema della Mutua Esclusione — 17 • Specifiche di Correttezza — 19 • Statement Atomici Particolari — 24 • Invarianti e Predicati — 25	15
2.3	Costrutti per la Programmazione Concorrente Costrutti Elementari — 27 • La Concorrenza nei Sistemi Operativi — 28 • Interazioni tra Processi — 30	27
CAPITOLO 3	SEMAFORI	PAGINA 32
3.1	Introduzione Invarianti Semaforici — 33 • Semafori per lo Scambio di Segnali Temporali — 34 • Prove di Correttezza — 35	32
3.2	Regione Critica Condizionale Tipi di Regione Critica Condizionale — 37	36
3.3	Semafori Privati Problema dei Lettori/Scrittori — 38	38
CAPITOLO 4	MONITOR	PAGINA 41
4.1	Introduzione Definire un Monitor — 41 • Variabili di Condizione — 42	41
4.2	Invarianti di Monitor	43
CAPITOLO 5	MODELLO A RETE	PAGINA 46
5.1	Introduzione Sistemi Fortemente e Debolmente Connessi — 46 • Canali di Comunicazione — 47 • Dichiarazione di Canali — 48 • Guardie — 49	46
5.2	Comunicazione Sincrona Transputer — 49 • Canali Sincroni - Pipe — 51	49
5.3	Rendez-vous e RPC Rendez-vous — 51 • RPC — 52	51

CAPITOLO 6	LINDA	PAGINA 55
6.1	Il Modello Linda	55
	Introduzione — 55 • Rendez-vous — 56	
6.2	Il Paradigma Master/Workers	56

Premessa

Licenza

Questi appunti sono rilasciati sotto licenza Creative Commons Attribuzione 4.0 Internazionale (per maggiori informazioni consultare il link: <https://creativecommons.org/licenses/by/4.0/>).



Formato utilizzato

Box di "Concetto sbagliato":

Concetto sbagliato 0.1: Testo del concetto sbagliato

Testo contenente il concetto giusto.

Box di "Corollario":

Corollario 0.0.1 Nome del corollario

Testo del corollario. Per corollario si intende una definizione minore, legata a un'altra definizione.

Box di "Definizione":

Definizione 0.0.1: Nome delle definizioni

Testo della definizione.

Box di "Domanda":

Domanda 0.1

Testo della domanda. Le domande sono spesso utilizzate per far riflettere sulle definizioni o sui concetti.

Box di "Esempio":

Esempio 0.0.1 (Nome dell'esempio)

Testo dell'esempio. Gli esempi sono tratti dalle slides del corso.

Box di "Note":

Note:-

Testo della nota. Le note sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive.

Box di "Osservazioni":

Osservazioni 0.0.1

Testo delle osservazioni. Le osservazioni sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive. A differenza delle note le osservazioni sono più specifiche.

1

Introduzione

1.1 Il corso in breve...

1.1.1 Cosa si intende per programmazione concorrente?

La programmazione concorrente nasce con i *sistemi concorrenti* nell'ambito dei *sistemi operativi* con il concetto di *processo* (o *thread*).

Definizione 1.1.1: Sistema concorrente

Un sistema concorrente è un sistema *software* implementato su una *piattaforma hardware* in grado di eseguire *contemporaneamente* più attività diverse che condividono *risorse comuni*^a.

^aPorzioni di memoria centrale, CPU, etc.

Note:-

Un esempio di sistemi concorrenti sono i *sistemi operativi multiprogrammati*.

Corollario 1.1.1 Programma concorrente

Un programma concorrente è un insieme di *moduli sequenziali* che possono essere eseguiti in parallelo.

Tipi di parallelismo:

- ⇒ *Parallelismo reale*: l'esecuzione dei moduli è realmente sovrapposta nel tempo;
- ⇒ *Parallelismo apparente*: si applica la tecnica d'*interleaving delle istruzioni*.

Note:-

In entrambi i casi il termine *concorrenza* si utilizza come un'astrazione per studiare il parallelismo.

In questo corso si tratterà di:

- Introduzione ai *principi* della programmazione concorrente;
- Analisi dei principali *costrutti linguistici* per la programmazione concorrente;
- Applicazione di questi costrutti a vari problemi di *sincronizzazione* e *comunicazione* in programmi concorrenti;
- Studio delle *proprietà di correttezza* dei programmi concorrenti: *no deadlock, no starvation*, etc.

1.1.2 Cosa si intende per algoritmo distribuito?

Definizione 1.1.2: Sistema distribuito

Un sistema distribuito è un sistema composto da *più computer* che non condividono la memoria o altre risorse, ma sono connessi da canali di comunicazione^a.

^aDebolmente connessi.

Corollario 1.1.2 Algoritmo distribuito

Un algoritmo distribuito è un algoritmo progettato per essere eseguito da un sistema distribuito.

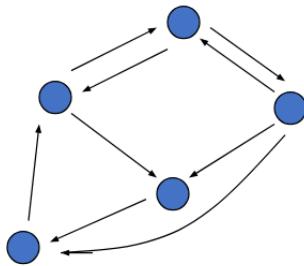


Figure 1.1: Schema di un sistema distribuito

Obiettivi per la parte di corso di algoritmi distribuiti:

- Introduzione di modelli formali che permettano l'analisi di *algoritmi distribuiti di base*;
- Studio della *correttezza* e delle *prestazioni* degli algoritmi distribuiti presentati;
- Analisi di *algoritmi distribuiti* in presenza di *malfunzionamenti* (algoritmi fault tolerant).

2

Programmazione concorrente

2.1 Parallelismo

2.1.1 Interleaving d'Istruzioni Atomiche

Definizione 2.1.1: Processo

Modulo *sequenziale* di un programma concorrente (a volte si usa il termine thread).

Note:-

Per gli scopi di questo corso processo e thread vengono assunti come sinonimi.

```
begin
:
cobegin
    process P():
        ...
    process Q():
        ...
coend
:
end
```

Figure 2.1: Pseudocodice

Domanda 2.1

Che cos'è l'interleaving?

Definizione 2.1.2: Interleaving

Si suppone che ogni esecuzione di un programma concorrente sia ottenuta *interfogliando in maniera arbitraria* le istruzioni dei vari processi.

Nome del programma	
begin	
:	
P	Q
...	...
:	
end	

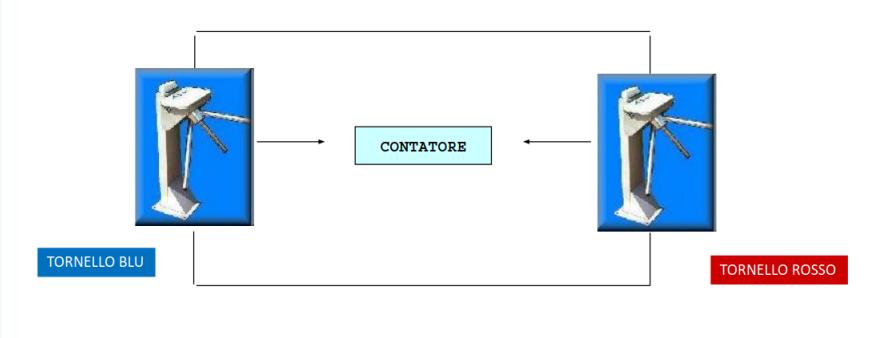
Figure 2.2: Tabella

Note:-

Il risultato dell'interleaving è detto *computazione* o *scenario*.

Esempio 2.1.1 (Quante persone sono entrate in laboratorio?)

Si ha un laboratorio con due tornelli che hanno un contatore condiviso che viene incrementato di 1 quando entra una persona.



Si inizia dichiarando la variabile condivisa *counter* (inizializzata a 0). Dopo di ché si suppone che il tornello blu faccia entrare 100 persone e il tornello rosso altre 100.

```

begin
    int counter ← 0 // variabile condivisa tra i due processi concorrenti
    cobegin
        process tornello_blu():
            for (int j ← 0; j<100; j++):
                sleep(500 + (int)(math.random()*1000) // prossimo arrivo
                counter++

        process tornello_rosso():
            for (int j ← 0; j<100; j++):
                sleep(500 + (int)(math.random()*1000) // prossimo arrivo
                counter++

    coend
    System.println("Numero ingressi:" + counter)
end

```

program Tornelli	
int counter ← 0	
process tornello_blu	process tornello_rosso
for (int j ← 0; j<100; j++): sleep(500 + int (math.random)...) counter++	
System.println ("Numero ingressi:" + counter)	

Alla fine della simulazione verrà stampato il valore 200? Dipende.

- Se l'istruzione **counter++** è atomica e indivisibile verrà stampato il valore 200;
- Ma se fosse realizzata mediante più load, add e store? In tal caso non è garantito che il risultato sarà 200.

Definizione 2.1.3: Istruzione atomica

Un'istruzione viene detta atomica se viene sempre eseguita interamente senza possibilità d'interruzioni^a.

^aNo interleaving.

Note:-

Il risultato dell'esecuzione "simultanea" di due istruzioni atomiche è lo stesso che si otterrebbe dalla loro esecuzione sequenziale indipendentemente dall'ordine. In generale si assumerà che gli assegnamenti e le valutazioni di espressioni logiche siano operazioni atomiche.

Esempio 2.1.2 (Possibile computazione)

Main	
int a ← 0 int b ← 0	
P	Q
p1: a ← a+1 p2: b ← b+1	q1: a ← a+2 q2: b ← b+2
end	

Diamo delle etichette alle istruzioni (indivisibili) dei programmi P e Q

Le possibili computazioni sono:

p1, q1, p2, q2	q1, p1, q2, p2
p1, q1, q2, p2	q1, p1, p2, q2
p1, p2, q1, q2	q1, q2, p1, p2

È importante notare che p2 non può precedere p1 e q2 non può precedere q1.

Algoritmo: Istruzioni Atomiche di Assegnamento	
integer n ← 0	
P	Q
p1: n ← n + 1	q1: n ← n + 1

Per l'algoritmo riportato sopra sono possibili due diversi **scenari**.

Process p	Process q	n
p1: n←n+1	q1: n←n+1	0
(end)	q1: n←n+1	1
(end)	(end)	2

Process p	Process q	n
p1: n←n+1	q1: n←n+1	0
p1: n←n+1	(end)	1
(end)	(end)	2

In entrambi i casi sia avrà sempre $n = 2$ come valore finale.

Algoritmo: Assegnamento con riferimento globale	
integer $n \leftarrow 0$	
p	q
integer temp p1: $\text{temp} \leftarrow n$ p2: $n \leftarrow \text{temp} + 1$	integer temp q1: $\text{temp} \leftarrow n$ q2: $n \leftarrow \text{temp} + 1$

Alcuni scenari di questo algoritmo restituiranno per la variabile n il valore atteso (2):

Process p	Process q	n	p.temp	q.temp
p1: $\text{temp} \leftarrow n$	q1: $\text{temp} \leftarrow n$	0	?	?
p2: $n \leftarrow \text{temp} + 1$	q1: $\text{temp} \leftarrow n$	0	0	?
(end)	q1: $\text{temp} \leftarrow n$	1	0	?
(end)	q2: $n \leftarrow \text{temp} + 1$	1	0	1
(end)	(end)	2	0	1

Altri scenari di questo algoritmo **non** restituiranno per la variabile n il valore atteso (2):

Process p	Process q	n	p.temp	q.temp
p1: $\text{temp} \leftarrow n$	q1: $\text{temp} \leftarrow n$	0	?	?
p2: $n \leftarrow \text{temp} + 1$	q1: $\text{temp} \leftarrow n$	0	0	?
p2: $n \leftarrow \text{temp} + 1$	q2: $n \leftarrow \text{temp} + 1$	0	0	0
(end)	q2: $n \leftarrow \text{temp} + 1$	1	0	0
(end)	(end)	1	0	0

In questo secondo algoritmo l'ordine di esecuzione influenza il risultato. Per cui avere operazioni atomiche non è sufficiente a garantire consistenza a un programma concorrente.

Note:-

Il comportamento osservato nel secondo algoritmo prende il nome di **race condition** (condizione di corsa).

2.1.2 Sistemi Monoprocesso e Sistemi Multiprocesso

Sistemi monoprocesso (parallelismo apparente):

- I processi sono *alternati nel tempo* per *simulare* un multiprocesso.
- In ogni istante un solo processo è in esecuzione.
- C'è interleaving nell'esecuzione delle singole istruzioni.
- Il meccanismo degli *interrupt* su cui è basato l'avvicendamento dei processi garantisce che l'interrupt venga servito primo o dopo l'esecuzione di un'istruzione, mai durante.
- Ogni istruzione macchina è *atomica*.

Sistemi multiprocessore con memoria comune (parallelismo reale):

- più processi vengono eseguiti *simultaneamente* su processori diversi;
- c'è *sovraposizione* (overlapping) nell'esecuzione delle istruzioni;
- i processi sono *sequenzializzati nell'accesso alla memoria*;
- le operazioni elementari (*microistruzioni*) che implementano istruzioni macchina sono realizzate da CPU diverse, ma le operazioni elementari che consistono in accessi alla memoria devono essere eseguite una alla volta;
- in questo caso è l'*arbitro del bus* che si preoccupa della sequenzializzazione e che garantisce l'atomicità delle operazioni di lettura/scrittura;
- l'accesso fisico al bus può rendere atomiche le istruzioni macchina.

Note:-

In entrambi i sistemi non è possibile prevedere l'alternanza nell'esecuzione di due processi.

Esempio 2.1.3 (Programma concorrente)

Main	
int n ← 0	
P	Q
p1: load R1,n p2: add R1,1 p3: store R1,n	q1: load R1,n q2: add R1,1 q3: store R1,n
print n	

Monoprocesso

Il sistema operativo effettua i context-switch per alternare l'esecuzione dei due processi sulla stessa CPU.

p1	load R1,n	
SO	interrupt	R1=0, n=0
SO	salvataggio P	n=0
SO	ripristino Q	n=0
q1	load R1,n	R1=? n=0
q2	add R1,1	R1=0, n=0
q3	store R1,n	R1=1, n=0
SO	interrupt	R1=1, n=1
SO	salvataggio Q	n=1
SO	ripristino P	n=1
p2	add R1,1	R1=0, n=1
p3	store R1,n	R1=1, n=1
		R1=1, n=1

Multiprocesso

I due processi sono eseguiti da due diversi processori (che hanno ovviamente insiemi di registri distinti)

P	Q	n
p1:load R_p,n		n=0
	q1:load R_q,n	n=0
p2:add R_p,1	q2:add R_q,1	n=0
	q3:store R_q,n	n=0
p3:store R_p,n		n=1
		n=1

Domanda 2.2

Le istruzioni macchina sono atomiche?

Esempio 2.1.4 (exc)

Consideriamo una istruzione macchina del tipo: **exc a,b** che scambi i contenuti delle celle di memoria **a** e **b**, e che sia implementata in questo modo (quindi **non atomico**):

```
exc a,b
rd R1, a
rd R2, b
wr R2, a
wr R1, b
```

Supponiamo che in una macchina multiprocessore ci siano due processi **P** e **Q** che eseguono entrambi l'unica istruzione macchina **exc a,b**. Supponiamo che inizialmente le celle di memoria **a** e **b** contengano rispettivamente i valori 0 e 1.

Main	
a = 0; b = 1	
P	Q
p1: exc a,b	q1: exc a,b

Consideriamo una possibile esecuzione delle due istruzioni parallele tenendo presente che gli accessi alla memoria centrale devono essere sequenzializzati

P	Q	R ^{P1}	R ^{P2}	R ^{Q1}	R ^{Q2}	a	b
rd R1,a		?	?	?	?	0	1
rd R2,b		0	?	?	?	0	1
wr R2,a		0	1	?	?	0	1
	rd R1,a	0	1	?	?	1	1
	rd R2,b	0	1	1	?	1	1
wr R1,b		0	1	1	1	1	1
	wr R2,a	0	1	1	1	1	0
	wr R1,b	0	1	1	1	1	0
		0	1	1	1	1	1

Il valore finale assunto dalle variabili **a** e **b** risulta <1 1>. Ma non esiste nessuna esecuzione sequenziale delle due istruzioni **exc a,b** che dia questo risultato!!

Definizione 2.1.4: Stato di un programma concorrente

Lo stato di un programma concorrente è una tupla costituita dai valori dei *control pointer*^a dei vari processi e dai valori delle variabili locali e globali.

^aIndica lo statement del processo che sta per essere eseguito.

Corollario 2.1.1 Transizione

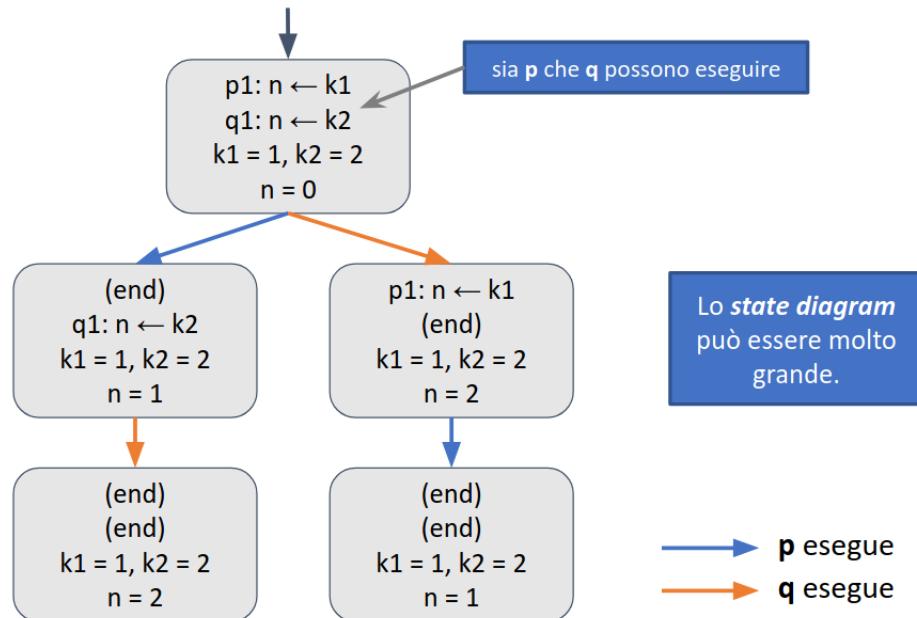
Siano s_1 e s_2 due stati di un programma concorrente. Esiste una *transizione* da s_1 a s_2 , indicata con $s_1 \rightarrow s_2$, se l'esecuzione di una istruzione nello stato s_1 porta nello stato s_2 .

Esempio 2.1.5 (Programma concorrente banale)

Algoritmo: Programma Concorrente Banale	
integer n ← 0	
p	q
integer k1 ← 1 p1: n ← k1	integer k2 ← 2 q1: n ← k2

- Lo stato deve includere i valori dei control pointers (p1 e q1) dei 2 processi, il valore della variabile globale n e i valori delle variabili locali k1 e k2;
- Lo stato iniziale può transire da 2 diversi stati a seconda di quale processo esegua per primo l'assegnamento;
- Si può illustrare il comportamento di questo programma concorrente con un *diagramma degli stati*.

Diagramma degli stati di un programma concorrente:



2.2 Correttezza di Programmi Concorrenti

2.2.1 Introduzione e Proprietà

La definizione di *correttezza totale* di un programma sequenziale richiede che il programma *termini* e che, per ogni input, il *risultato* restituito dal programma sia il valore della funzione che il programma deve calcolare. Purtroppo questa definizione non è adeguata al caso dei programmi concorrenti:

- Può essere desiderabile che un programma concorrente non termini (ad esempio i processi server degli OS).
- Inoltre un programma concorrente non può più essere considerato una funzione perché si aggiungono richieste come *mutua esclusione*, l'assenza di *deadlock*, l'assenza di *starvation*.

Note:-

La correttezza di un programma concorrente viene definita in termini di *validità di proprietà*.

Lampton ha definito due categorie di proprietà di correttezza:

- *Safety*: la proprietà P deve sempre essere vera (P è vera in ogni stato della computazione);
- *Liveness*: la proprietà P prima o poi sarà vera (in ogni computazione esiste uno stato in cui P è vera).

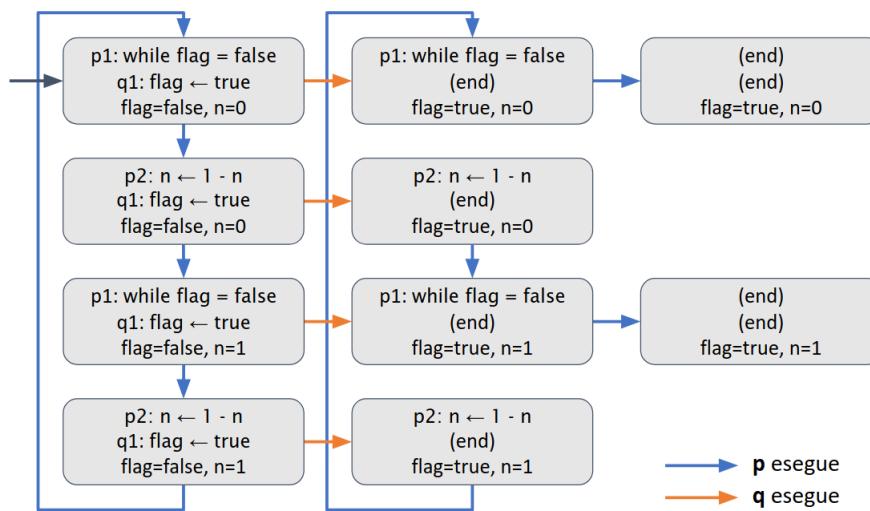
Domanda 2.3

Questo programma termina per tutte le computazioni?

Algoritmo StopTheLoopA	
boolean flag \leftarrow false integer n \leftarrow 0	
p	q
p1: while flag = false: p2: n \leftarrow 1 - n	q1: flag \leftarrow true q2:

Note:-

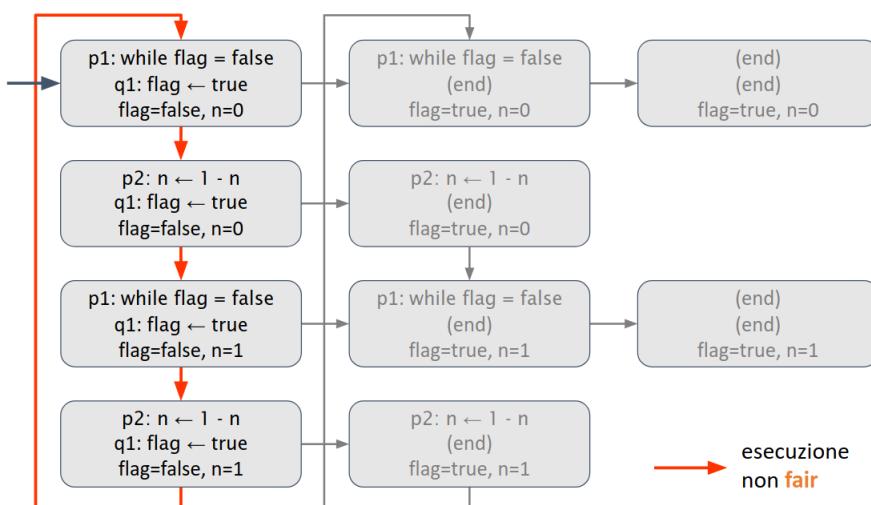
Per rispondere dobbiamo vedere il diagramma degli stati corrispondente.



Risposta: questo programma termina solo in due stati, quindi la risposta è no.

Note:-

Questo comportamento non è *fair*.



Definizione 2.2.1: Fairness

Una computazione è *fair* se per ogni istruzione "costantemente" abilitata esiste uno stato in cui essa viene eseguita, prima o poi.

Note:-

Imporre le *ipotesi di fairness* a un programma concorrente significa escludere dalle sue computazioni quelle che non soddisfano la proprietà di fairness.

2.2.2 La Correttezza di Programmi**La correttezza di un *programma sequenziale*:**

1. Terminazione.
2. Correttezza del risultato.

Definizione 2.2.2: Program testing

Per controllare la correttezza di un programma sequenziale si può:

- Eseguire il programma n volte con dei *punti di break* in modo da individuare eventuali malfunzionamenti (debugging) o testare con input predefiniti (unit test).
- Ricorrere a metodi di verifica (*program verification*) basati su sistemi formali, per esempio la logica dei predicati.

Note:-

La correttezza di un programma concorrente richiede la verifica di opportune proprietà per ogni possibile sua computazione, per cui le tecniche di debugging sono inefficienti.

Per provare la correttezza di un programma concorrente si possono usare:

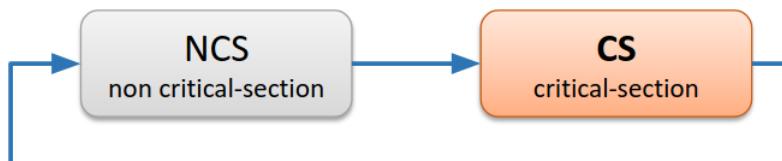
- *Diagramma degli stati*: però spesso si hanno troppi stati ed è difficile verificare le proprietà di Liveness;
- *Metodi formali di verifica*: prove di proprietà scritte in un adeguato linguaggio logico, basate su invarianti;
- *Model checker*: un programma che costituisce il diagramma degli stati di un programma concorrente e simultaneamente verifica le proprietà.

2.2.3 Il Problema della Mutua Esclusione**Definizione 2.2.3: Problema della mutua esclusione**

Se un processo esegue la propria sezione critica, nessun altro processo esegue la propria.

Note:-

N processi concorrenti eseguono in un loop infinito una sequenza di statement che può essere divisa in due sotto-sequenze: non-sezione critica (NCS) e sezione critica (CS).



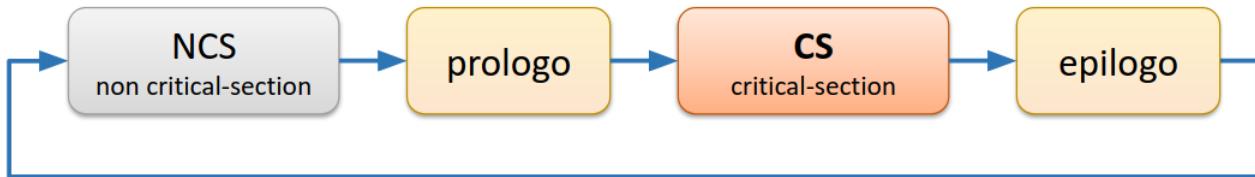
Definizione 2.2.4: Sincronizzazione

Per ottenere la mutua esclusione si deve introdurre un *meccanismo di sincronizzazione*, costituito da statement aggiuntivi, alcuni posti prima dell'accesso alla sezione critica: *prologo* (preprotocol) e alcuni all'uscita della sezione critica: *epilogo* (postprotocol).

Algoritmo Critical Section Problem	
global variables	
p	q
local variables loop forever: non-critical section preprotocol critical section postprotocol	local variables loop forever: non-critical section preprotocol critical section postprotocol

Per risolvere il problema della mutua esclusione si parte da alcune assunzioni:

1. Quando un processo è in CS progredisce nell'esecuzione del suo codice e lo porta a termine (non ci sono loop infiniti, statement di terminazione o malfunzionamenti) (*proprietà di progresso all'interno della sezione critica*).
2. Le computazioni sono fair.
3. Le variabili usate nel prologo e nell'epilogo non sono utilizzate in CS o in NCS.
4. I processi utilizzano sempre correttamente il protocollo.

**Note:-**

- Se un utente si blocca in CS l'intero sistema si blocca o va in controllo a malfunzionamenti;
- Le assunzioni 1 e 2 garantiscono che nelle CS non si presentino malfunzionamenti, cicli infiniti, statement di terminazione, etc.
- Non si fanno restrizioni sul codice in NCS.
- La quarta assunzione garantisce che i protocolli proposti siano sempre eseguiti.
- La terza assunzione garantisce che le variabili del protocollo siano disgiunte rispetto a quelle dei singoli processi.

2.2.4 Specifiche di Correttezza

Approccio di Dijkstra (storico):

- *Mutua esclusione*: solo un processo alla volta deve essere all'interno della CS.
- *Assenza di deadlock*: non può accadere che tutti i processi siano bloccati definitivamente durante l'esecuzione del prologo.
- *Assenza di delay non necessari*: un processo fuori dalla CS non può ritardare l'accesso alla CS da parte di un altro processo.
- *Assenza di starvation*: ogni processo che richiede l'accesso alla CS prima o poi l'ottiene.

Note:-

Le prime tre proprietà sono fondamentali, la quarta in alcuni casi può essere omessa.

Approccio di Silberschatz-Galvin (concetto dei sistemi operativi):

- *Mutua esclusione*.
- *Progresso*: se nessun processo sta eseguendo in CS e alcuni processi richiedono l'accesso alla propria CS, allora i processi che sono in NCS non possono partecipare alla decisione riguardante la scelta di chi può entrare per primo in CS (condensa i punti due e tre di Dijkstra).
- *Attesa limitata*: c'è un limite sul numero di accessi alla CS accordati a un processo, quando un altro processo ha fatto richiesta di accesso alla propria CS, ed è ancora in attesa di entrare (modo alternativo di definire l'assenza di starvation).

Approccio di Ben-Ari (principi di programmazione concorrente e distribuita):

- *Mutua esclusione*: non ci può essere interleaving tra gli statement della CS dei processi.
- *Assenza di deadlock*.
- *Assenza di starvation*.

Approccio di Lynch (algoritmi distribuiti):

- *Mutua esclusione*: non esiste uno stato del programma concorrente in cui più di un processo è in CS.
- *Progresso*: in ogni computazione fair deve essere verificato che:
 - Se uno o più processi stanno eseguendo il prologo e nessuno è in CS, prima o poi uno di questi processi accede alla CS (progresso per l'accesso).
 - Se un processo sta eseguendo l'epilogo prima o poi accede alla NCS (progresso per l'uscita).
- *Assenza di starvation* (lockout freedom): in ogni computazione fair ogni processo che sta eseguendo il prologo (epilogo) prima o poi accede alla CS (esce dalla CS).

Esempio 2.2.1 (Mutua esclusione di due processi)

Si vuole dimostrare la correttezza della seguente soluzione mediante il diagramma degli stati. Ogni stato è definito da terne del tipo $(pi, qj, turn)$.

Algoritmo 1	
integer turn ← 1	
p	q
loop forever: p1: non-critical section p2: await turn = 1 p3: critical section p4: turn ← 2	loop forever: q1: non-critical section q2: await turn = 2 q3: critical section q4: turn ← 1

★ Il costrutto: **await** condizione: istruzioni
è definito come: while not condizione: <esegui istruzioni>

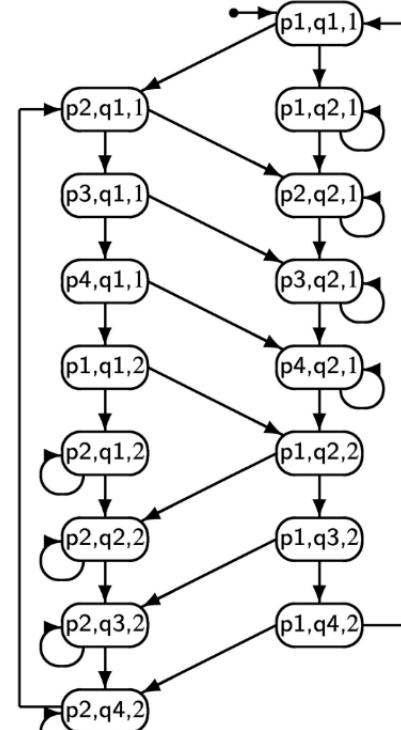
Per determinarne la correttezza si costruisce il diagramma degli stati e si sceglie di seguire l'approccio di Ben-Ari:

- Mutua esclusione;
- Assenza di deadlock;
- Assenza di starvation.

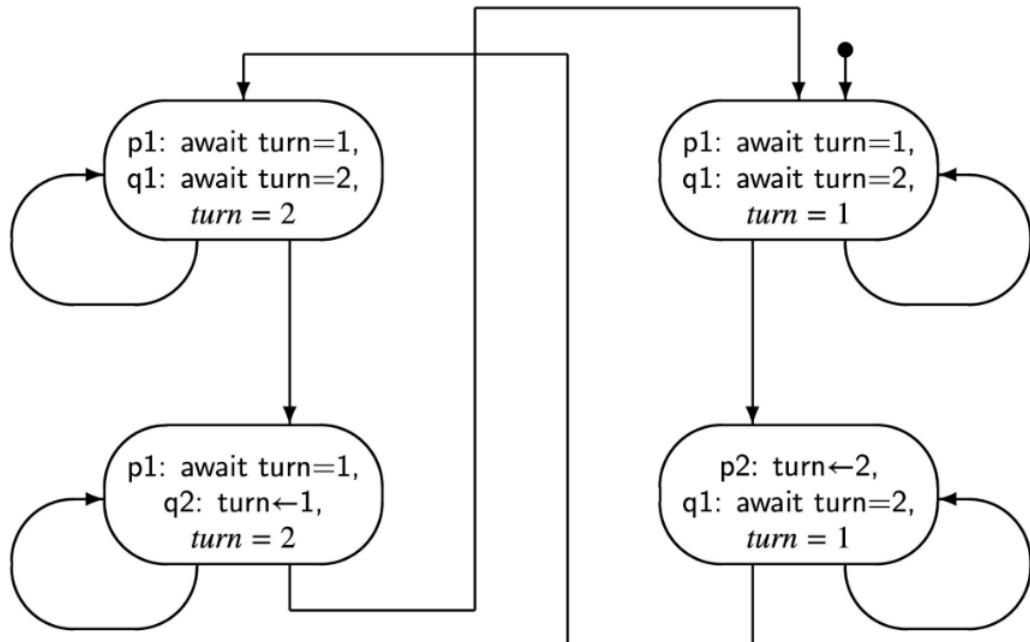
Costruiamo quindi il Diagramma degli stati.

p
loop forever:
p1: non-critical section
p2: await turn = 1
p3: critical section
p4: turn ← 2

q
loop forever:
q1: non-critical section
q2: await turn = 2
q3: critical section
q4: turn ← 1



Mutua esclusione: è sufficiente verificare che nel diagramma non esistono stati in cui p e q siano entrambi in sezione critica, ossia stati del tipo (p3, q3, *).



No deadlock: si utilizza una versione semplificata del diagramma degli stati non considerando la CS. Si vede che ogni stato ha sempre un'uscita.

No starvation: questo non è vero perché non si ha un progresso garantito al di fuori della CS.

Esempio 2.2.2 (Mutua esclusione)

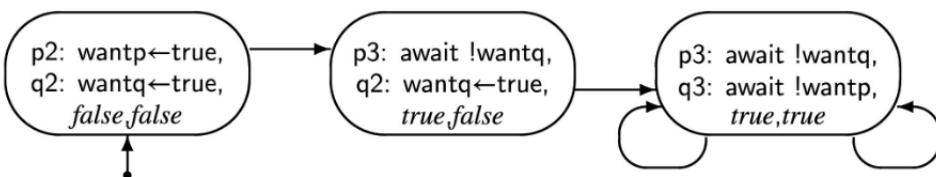
Consideriamo un altro algoritmo che non utilizza turn, ma wantp e wantq per indicare se un processo si trova in sezione critica.

Algoritmo 2	
boolean wantp ← false, wantq ← false	
p	q
loop forever: p1: non-critical section p2: wantp ← true p3: await wantq = false p4: critical section p5: wantp ← false	loop forever: q1: non-critical section q2: wantq ← true q3: await wantp = false q4: critical section q5: wantq ← false

Questo algoritmo garantisce la mutua esclusione però può dare origine a deadlock.

Process p	Process q	wantp	wantq
p1: non-critical section	q1: non-critical section	false	false
p2: wantp \leftarrow true	q1: non-critical section	false	false
p2: wantp\leftarrowtrue	q2: wantq \leftarrow true	false	false
p3: await wantq=false	q2: wantq\leftarrowtrue	true	false
p3: await wantq=false	q3: await wantp=false	true	true

Se entrambi i processi impostano wantp e wantq a true, poi si aspettano a vicenda, e nessuno dei due entra in CS.



Note:-

Entrambi questi algoritmi sono scorretti. Si deve usare l'algoritmo di Dekker (proposto da Dijkstra nel 1965) che usa sia wantp/wantq che turn per stabilire quale dei due processi abbia il diritto d'insistere nel tentativo di accesso.

Esempio 2.2.3 (Algoritmo di Dekker)

Algoritmo di Dekker	
<pre> boolean wantp \leftarrow false, wantq \leftarrow false integer turn \leftarrow 1 </pre>	
p	q
loop forever: p1: non-critical section p2: wantp \leftarrow true p3: while wantq p4: if turn = 2 p5: wantp \leftarrow false p6: await turn = 1 p7: wantp \leftarrow true p8: critical section p9: turn \leftarrow 2 p10: wantp \leftarrow false	loop forever: q1: non-critical section q2: wantq \leftarrow true q3: while wantp q4: if turn = 1 q5: wantq \leftarrow false q6: await turn = 2 q7: wantq \leftarrow true q8: critical section q9: turn \leftarrow 1 q10: wantq \leftarrow false

La sua correttezza può essere provata dal diagramma degli stati.

Per provarla in maniera informale si può ragionare per assurdo sulla mutua esclusione: supponiamo che entrambi i processi siano in CS. Uno dei due sarà entrato per primo, supponiamo q (quindi wantq = true). Ora se p entra in CS prima che q ne esca dovrebbe esistere un istante in cui wantq = false mentre q è in CS, ma ciò è assurdo.

Per l'assenza di deadlock si suppone che entrambi i processi vogliano entrare nel ciclo while. Le variabili wantp e wantq sono entrambe true e si suppone turn = 2. Il valore di turn sarà modificato solo quando q uscirà dalla CS. Quindi il processo p dovrà eseguire p5 ed entrare nel ciclo await. Non c'è nulla che ostacoli

q a uscire dal while ed entrare in CS.

Per l'assenza di starvation si suppone che il processo p voglia entrare in CS e il processo q sta eseguendo in NCS, p entra. Se invece il processo q esegue in CS o nel prologo/epilogo, per ipotesi prima o poi termina e pone turn = 1 e wantq = false, quindi p può entrare.

Note:-

Nel 1981 Peterson propose una variante dell'algoritmo di Dekker più semplice e più generale. Invece di usare turn utilizza una variabile last per indicare chi è stato l'ultimo a provare ad accedere: questo determina la priorità di accesso.

Esempio 2.2.4 (Algoritmo di Peterson)

Algoritmo di Peterson	
p	q
loop forever:	loop forever:
p1: non-critical section	q1: non-critical section
p2: wantp \leftarrow true	q2: wantq \leftarrow true
p3: last \leftarrow 1	q3: last \leftarrow 2
p4: await wantq=false or last=2	q4: await wantp=false or last=1
p5: critical section	q5: critical section
p6: wantp \leftarrow false	wantq \leftarrow false

**Diagramma stati
algoritmo di Peterson
(abbreviato)**

M.E.:

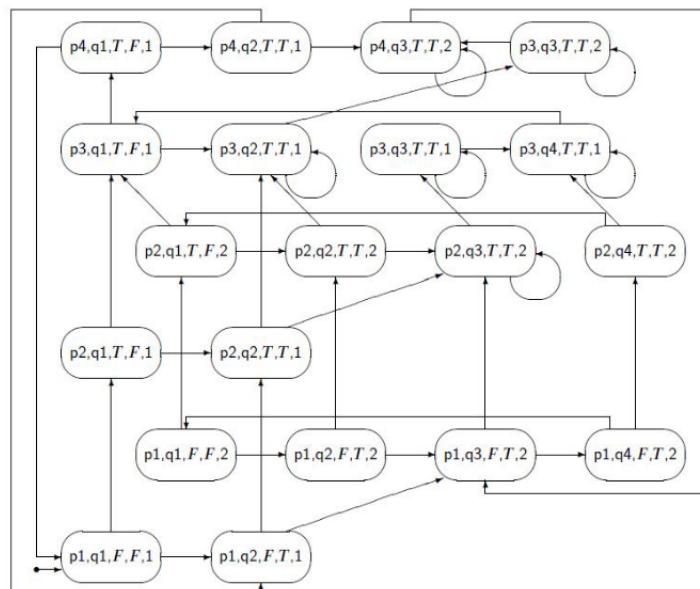
non esistono stati della forma
(p4,q3...)

No deadlock:

è possibile uscire dallo stato
(p3,q3...)

No starvation:

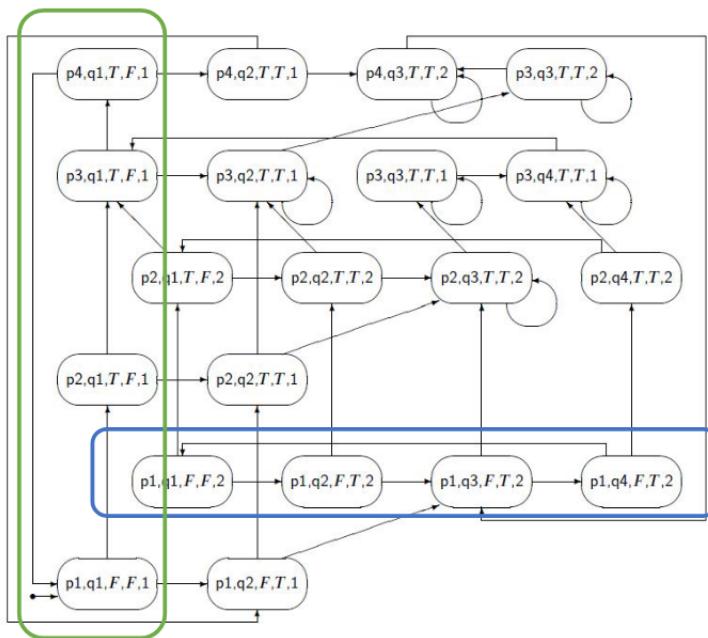
non esistono loop che
coinvolgono stati che non
contengano p4 (o q4)



**Diagramma stati
algoritmo di Peterson
(abbreviato)**

NOTA:

Esistono loop dove **p1** o **q1** non avanzano. Questi però sono esclusi, in quanto sono le computazioni **unfair**.



Note:-

Nelle moderne CPU multi-core gli algoritmi di Dekker e Peterson non funzionano più (almeno non nella forma vista precedentemente). Perché:

- Coerenza della memoria per codice sequenziale per core;
- Utilizzo della cache intermedia;
- Servono le memory fences (per sincronizzare le cache intermedie).

2.2.5 Statement Atomici Particolari

Fino a ora abbiamo supposto l'utilizzo d'istruzioni atomiche di assegnamento (write e read). Esistono istruzioni atomiche particolari che permettono di realizzare facili ed efficienti meccanismi di sincronizzazione:

- test-and-set(common, local);
- exchange(a, b);
- fetch-and-add(common, local, x).

Note:-

Questi statement sono tali che durante la loro esecuzione accedono direttamente in memoria centrale (no cache) e non rilasciano il controllo del bus. Così evitano interleaving di operazioni di write e read da parte di altri processi.

Esempio 2.2.5 (test-and-set)

atomic instruction test-and-set (**common**, **local**):
local \leftarrow **common**
common \leftarrow 1

Algoritmo: Sezione critica con test-and-set	
integer common \leftarrow 0	
p	q
integer local1 loop forever: p1: non-critical section p2: repeat test-and-set(common, local1) p3: until local1 = 0 p4: critical section p5: common \leftarrow 0	integer local2 loop forever: q1: non-critical section q2: repeat test-and-set(common, local2) q3: until local2 = 0 q4: critical section q5: common \leftarrow 0

2.2.6 Invarianti e Predicati

Per provare la correttezza di un programma concorrente si può usare l'induzione su *proprietà invarianti*.

Definizione 2.2.5: Invariante

Un *invariante* è una formula A in un qualche sistema formale che ha la proprietà di *essere sempre vera in ogni punto di qualsiasi computazione*.

Corollario 2.2.1 Predicato

Un *predicato*, chiamato anche *proposizione atomica*, è un'espressione booleana che può essere valutata avendo a disposizione la tupla delle variabili di stato dei processi, e i loro control pointers.

Note:-

Le formule A che verranno utilizzate sono anche proposizioni atomiche.

Definizione 2.2.6: Contatore monotonico

Un *contatore monotonico* (o logical clock) è una meta-variabile il cui valore si incrementa quando un processo esegue una certa operazione o azione.

Data la computazione $\dots a_1 \dots a_2 \dots a_3 \dots a_4 \dots$ ogni occorrenza i-esima dell'azione a_i riceve un valore c_i dal contatore.

In una computazione, due istanze a_i e a_j di un'azione a sono temporalmente ordinate: $c_i < c_j \Leftrightarrow a_i$ è avvenuta prima di a_j .

Osservazioni 2.2.1 Predicati e invarianti

Un predicato (proprietà) A è invariante se è vera in ogni punto di qualsiasi computazione.

Domanda 2.4

Come si dimostra che una proprietà A è un'invariante?

Risposta: la prova che A è un'invariante si fa *per induzione* sugli stati di tutte le computazioni. Si prova che A vale nello stato iniziale (passo base) e si suppone che A valga in uno stato s e in tutti gli stati precedenti a s (ipotesi induttiva) e si dimostra che vale anche in ciascuno dei possibili stati successivi (passo induttivo).

Note:-

In uno programma concorrente gli stati successori possono essere più di uno (per via dell'interleaving) quindi il passo induttivo deve essere verificato per tutti i possibili successori.

Esempio 2.2.6 (Invarianti)

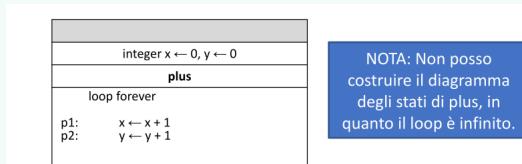
Consideriamo il programma costituito da un unico processo plus; siano:

- **val(x)** il valore della variabile x in uno stato s della computazione;
- **val(y)** il valore della variabile y in uno stato s della computazione;

Si vuole provare che la formula:

$$0 \leq val(x) - val(y) \leq 1$$

è un *invariante* del programma.



Si introducono le notazioni:

- **Npi** (i:1...2): variabili intere che in ogni stato indicano il numero N di esecuzioni di pi fino a quel punto completate;
- **pi** (i:1...2): proposizione atomica vera se il control pointer del processo plus vale pi.

Lemma 1. La seguente formula **A** è un invariante:

$$A: (p1 \rightarrow (Np1 = Np2)) \wedge (p2 \rightarrow (Np1 = Np2 + 1))$$

La dimostrazione è per induzione sugli stati. L'assunto è banalmente vero nello stato iniziale (passo base). Supposto che **A** sia vera in tutti gli stati precedenti lo stato **s** dimostriamo che è vera anche per lo stato **s**. Se nello stato che prendiamo in esame il control pointer vale p2, nello stato precedente il control pointer valeva p1 e l'ipotesi induttiva garantisce che **Np1 = Np2**, ora l'esecuzione di p1 ha comportato l'incremento di **Np1** e quindi possiamo affermare che **p2 → (Np1 = Np2 + 1)**

Se il control pointer vale p1 la prova è analoga.

Dal Lemma 1 si può dedurre che in ogni stato vale l'Invariante

$$0 \leq Np1 - Np2 \leq 1$$

Un'invariante che si basa sull'ordine delle istruzioni di un processo viene indicato come *invariante topologico*.

Lemma 2. La seguente formula B è un invarianto:

$$B: (\text{val}(x) = Np1) \wedge (\text{val}(y) = Np2)$$

La dimostrazione è per induzione sugli stati. L'assunto è banalmente vero nello stato iniziale (passo base).

Supponendo B vera in tutti gli stati precedenti lo stato s , dimostriamo che è vera anche in s . Supponiamo che in s il control pointer valga $p2$: quindi nello stato precedente il control pointer valeva $p1$ e l'ipotesi induttiva garantiva che $\text{val}(x) = Np1$, per transire ad s è stata eseguita $p1$ che ha comportato l'incremento di $Np1$ ma anche l'incremento di x quindi la prima parte della formula è nuovamente vera, la seconda parte della formula non è stata modificata.

Se il control pointer vale $p1$ la prova è analoga.

Teorema. La formula è un invarianto:

$$0 \leq \text{val}(x) - \text{val}(y) \leq 1$$

La prova segue dai lemmi 1 e 2.

Note:-

Con più processi bisogna ragionare sull'interleaving.

2.3 Costrutti per la Programmazione Concorrente

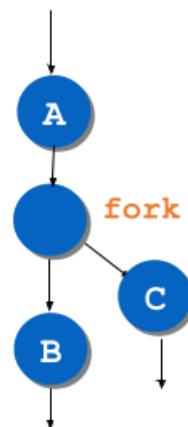
Un programma concorrente è costituito da un insieme di *processi sequenziali asincroni interagenti*. Un linguaggio concorrente dovrà contenere:

- Costrutti che permettano di dichiarare moduli di programma destinati a essere eseguiti concorrentemente come *processi sequenziali*.
- Strumenti linguistici per specificare le *interazioni* tra i processi.

2.3.1 Costrutti Elementari

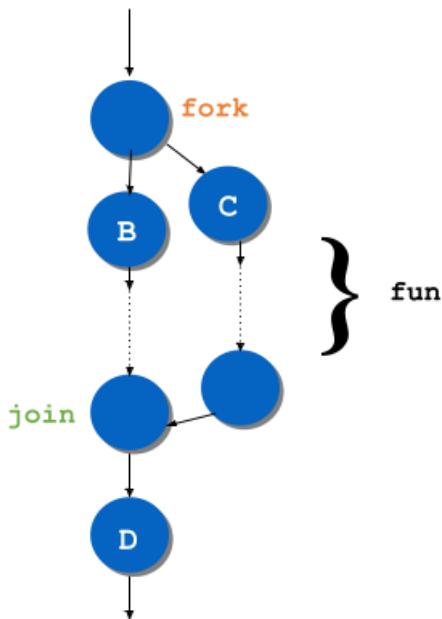
Definizione 2.3.1: Fork

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.



Note:-

La fork corrisponde al modello di avvio dei processi in UNIX.



Definizione 2.3.2: Join

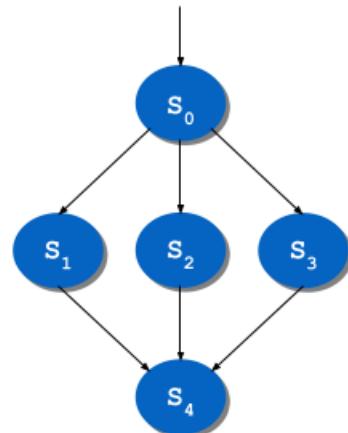
Una *join* serve per sincronizzare i processi. Il processo che fa la join aspetta la terminazione dell'altro processo.

Note:-

In UNIX l'operazione join si chiama *waitpid()*.

Definizione 2.3.3: Cobegin/Coend

La concorrenza può essere espressa in modo strutturato con n processi che eseguono in parallelo.



2.3.2 La Concorrenza nei Sistemi Operativi

Definizione 2.3.4: Processo

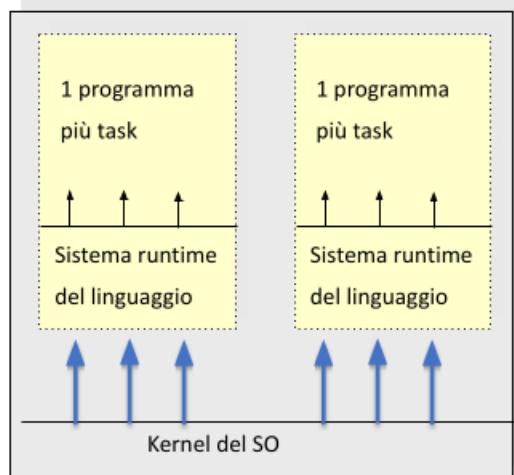
Costrutto linguistico per individuare, in modo sintatticamente preciso, quali moduli di un programma possono essere eseguiti come processi autonomi. Specializza quindi il concetto di funzione, indicando esplicitamente che andranno in esecuzione concorrente.

```

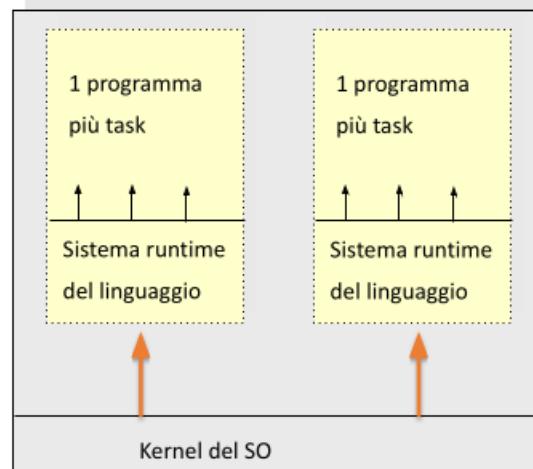
process <identificatore>(<parametri formali>):
    <dichiarazione di variabili locali>
    ...
    <corpo del processo>
    ...
  
```

Note:-

Quando i programmi sono concorrenti, i task possono essere *kernel thread* (con supporto del SO) o *user thread* (senza supporto del SO).



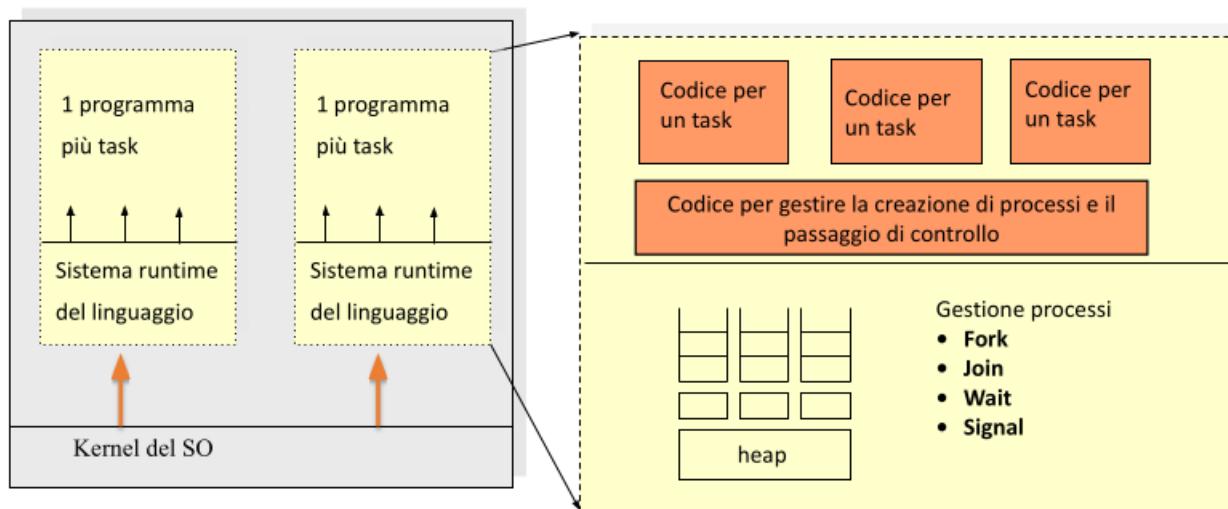
Linguaggio di programmazione concorrente **con supporto del SO**



Linguaggio di programmazione concorrente **senza supporto del SO**

Note:-

Se il linguaggio utilizza gli *user thread* è il runtime stesso che fornisce tutte le procedure per la gestione dei processi concorrenti di quel programma.



Linguaggio di programmazione concorrente **senza supporto SO**

2.3.3 Interazioni tra Processi

Le *interazioni tra processi* possono essere classificate in due modi.

Definizione 2.3.5: Cooperazione

La *cooperazione* è un'interazione prevedibile e desiderata (sincronizzazione diretta ed esplicita):

- Scambio di segnali temporali.
- Scambio di informazione.

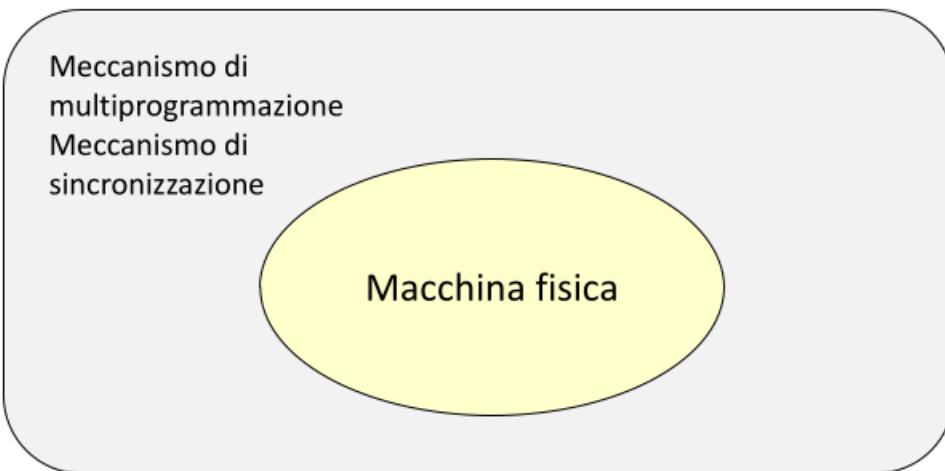
Definizione 2.3.6: Competizione

La *competizione* è un'interazione imprevedibile e non desiderata, ma necessaria (sincronizzazione indiretta o implicita):

- Mutua esclusione.

La macchina su cui un programma concorrente va in esecuzione deve essere in grado di:

- Eseguire in concorrenza un certo numero n di processi sequenziali ($n > 1$).
- Permettere agli processi di sincronizzarsi offrendo primitive adeguate.



Macchina virtuale e macchina fisica

Si hanno due diverse organizzazioni logiche:

- *Modello a Memoria Comune*: i processi comunicano tramite variabili condivise.
- *Modelli a Rete*: i processi comunicano attraverso una sottorete di comunicazione.

Osservazioni 2.3.1

- L'esecuzione di un programma concorrente non è il modo migliore per studiarne la concorrenza.
- Si possono utilizzare dei simulatori di concorrenza: strumenti software che permettono di controllare l'interleaving di azioni atomiche.
- I simulatori mantengono le strutture dati per emulare l'esecuzione concorrente e si aggiornano dopo aver interpretato un singolo statement.

3

Semafori

3.1 Introduzione

Definizione 3.1.1: Semaforo

Un *semaforo* è un tipo di dato astratto^a, che assume valori interi $i, \geq 0$, su cui si eseguono solo due primitive atomiche: P e V (wait, signal).

^aOpaco, non si sa com'è implementato.

Note:-

Una variabile di tipo *semaphore* è condivisibile da due o più processi.

Osservazioni 3.1.1

- Strumento linguistico di "basso livello" per *problemi di sincronizzazione*.
- Normalmente realizzato a *livello macchina* per realizzare strumenti di sincronizzazione di "più alto livello".
- Disponibile in librerie standard per la realizzazione di programmi concorrenti con linguaggi sequenziali.

Corollario 3.1.1 P

La P prova ad accedere al semaforo. Quando accede decrementa il contatore di 1.

```
void P(semaphore s):
    await(val_s > 0)
    val_s--
```

Corollario 3.1.2 V

Incrementa di 1 il contatore.

```
void V(semaphore s):
    val_s++
```

Un semaforo è:

- *Generale*(o contatore): se il suo valore può assumere qualsiasi valore intero non negativo.
- *Binario*: se può assumere solo i valori 0 e 1 (la V aumenta solo se il valore corrente è 0).

Implementazione dei Semafori

La definizione di Dijkstra, essendo astratta, non tiene conto degli aspetti implementativi. Per quanto riguarda l'operazione P sono disponibili varie soluzioni:

- *Busy Waiting*: attesa attiva, *spinlock*.
- *Insieme di processi bloccati*.
- *Coda FIFO di processi bloccati*.

Note:-

In questo corso non si faranno assunzioni sul tipo di implementazione, ma solamente sulla gestione fair dei processi bloccati.

3.1.1 Invarianti Semaforici

Definizione 3.1.2: Invariante Semaforico

- I_{sem} : valore intero ≥ 0 con cui il semaforo *sem* viene inizializzato.
- NV_{sem} : numero di volte che l'operazione V(sem) è stata *eseguita* fino allo stato s.
- NP_{sem} : numero di volte che l'operazione P(sem) è stata *completata* fino allo stato s.

$$val_{sem}(s) = I_{sem} + NV_{sem}(s) - NP_{sem}(s), \text{ con } val_{sem} \geq 0$$

In ogni stato vale:

$$NP_{sem} \leq I_{sem} + NV_{sem}$$

Il Problema della Mutua Esclusione

Il problema della mutua esclusione tra un insieme di processi si può risolvere con l'utilizzo di un semaforo binario chiamato *mutex*.

Definizione 3.1.3: Mutex

Il mutex è un semaforo binario inizializzato a 1 con semantica:

- $val(mutex) = 1$, la sezione critica è libera.
- $val(mutex) = 0$, la sezione critica è stata acquisita da un processo.

Il *prologo* per l'accesso alla CS è costituito dall'operazione P(mutex) e l'*epilogo* dall'operazione V(mutex).

Mutua esclusione con semafori (n processi)				
binary semaphore mutex $\leftarrow 1$				
p₁	...	p_i	...	p_n
loop forever NCS P(mutex) CS V(mutex)		loop forever NCS P(mutex) CS V(mutex)		loop forever NCS P(mutex) CS V(mutex)

Una buona soluzione ha:

- *Mutua Esclusione.*
- *Assenza di Deadlock.*
- *Assenza di Starvation.*

Note:-

L'assenza di starvation dipende strettamente dall'implementazione dei semafori, quindi si può assumere (ipotesi strong).

```
binary semaphore mutex ← 1
loop forever
    NCS
    P(mutex)
    CS
    V(mutex)
```

Mutua esclusione

Sia $nc(s)$ una proposizione atomica che indica il numero di processi nello stato s che stanno operando in sezione critica. La M.E. è dimostrata se si prova che in ogni stato s: $0 \leq nc \leq 1$.

Dal codice si ha che : $nc = NP_{mutex} - NV_{mutex}$ (*invariante topologico*)

Per l'*invariante semaforico* del semaforo **mutex** sappiamo che:

$$NP_{mutex} \leq 1 + NV_{mutex} \Rightarrow NP_{mutex} - NV_{mutex} \leq 1 \Rightarrow nc \leq 1$$

Dal codice abbiamo inoltre.

$$NV_{mutex} \leq NP_{mutex} \Rightarrow NP_{mutex} - NV_{mutex} \geq 0 \Rightarrow nc \geq 0$$

```
binary semaphore mutex ← 1
loop forever
    NCS
    P(mutex)
    CS
    V(mutex)
```

Assenza di deadlock (per accedere alla sezione critica)

Il deadlock si verifica se, in un certo stato s, ci sono m ($1 < m \leq n$) processi sono bloccati sul semaforo **mutex** mentre nessuno sta eseguendo in sezione critica. Il fatto che m processi siano bloccati può accadere solo se i processi stanno eseguendo l'operazione **P(mutex)** e il valore di **mutex** è uguale a 0. La prova è per assurdo. Si suppone che esista un tale stato s.

Dal codice sappiamo che se nello stato s, m processi sono bloccati sull'operazione **P(mutex)** e nessuno degli N processi è in CS: $NP_{mutex} = NV_{mutex}$ (derivato dall'inv. topologico)

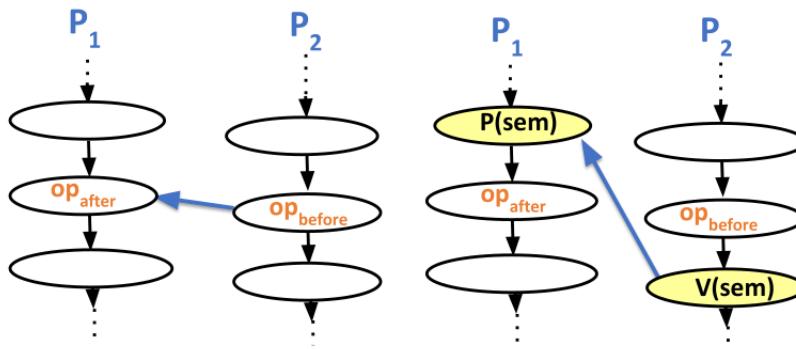
Per la definizione di semaforo inoltre abbiamo che: $val_{mutex} = 1 + NV_{mutex} - NP_{mutex}$

ma nello stato s il valore di **mutex** deve essere uguale a 0 (stiamo assumendo s sia un deadlock). Quindi

$$\begin{aligned} &\Rightarrow 1 + NV_{mutex} - NP_{mutex} = 0 \\ &\Rightarrow NV_{mutex} = NP_{mutex} - 1 \quad (\text{applichiamo } NP_{mutex} = NV_{mutex}) \\ &\Rightarrow NP_{mutex} = NP_{mutex} - 1 \\ &\Rightarrow 0 = -1 \quad \text{ASSURDO!!} \end{aligned}$$

3.1.2 Semafori per lo Scambio di Segnali Temporali

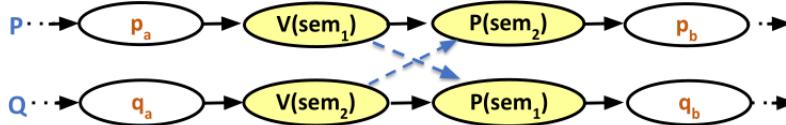
Un uso frequente dei semafori è quello di imporre un ordine di esecuzione tra le operazioni dei processi concorrenti.



Definizione 3.1.4: Barriera Sincrona

La *barriera sincrona* (o Rendezvous sincrono) tra due processi:

- Due processi P e Q eseguono ciascuno due operazioni consecutive. Pa e Pb il primo, Qa e Qb il secondo.
- *Vincolo di barriera*: l'esecuzione di Pb da parte di P e quella di qb da parte di Q possono iniziare solo dopo che entrambi i processi hanno completato la loro prima operazione.



3.1.3 Prove di Correttezza

Produttore-Consumatore con un Buffer Infinito

- Due processi, produttore e consumatore, si scambiano dati di tipo *type* utilizzando un buffer condiviso infinito.
- Il consumatore non deve cercare di prelevare un dato dal buffer quando questo è vuoto. Se il consumatore trova il buffer vuoto deve attendere.
- Si suppone che *take* e *append* siano atomiche.

Algoritmo produttore/consumatore con buffer infinito	
produttore	consumatore
<pre> infinite queue of data Type buffer ← empty semaphore full ← 0 </pre>	<pre> data Type d loop forever d ← produce append(d, buffer) V(full) </pre>

Algoritmo produttore/consumatore con buffer infinito	
produttore	consumatore
<pre> data Type d loop forever d ← produce append(d, buffer) V(full) </pre>	<pre> data Type d loop forever P(full) d ← take(buffer) consume(d) </pre>

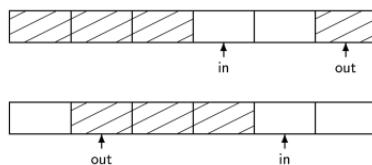
Produttore-Consumatore con un Buffer Limitato

- Due processi, produttore e consumatore, si scambiano dati di tipo *type* utilizzando un buffer condiviso in grado di contenere max dati.
- Il consumatore non deve cercare di prelevare un dato dal buffer quando questo è vuoto. Se il consumatore trova il buffer vuoto deve attendere.

- Il produttore non deve cercare di inserire un dato nel buffer quando questo è pieno.
- Si suppone che *take* e *append* siano atomiche.

Algoritmo produttore/consumatore con buffer finito	
produttore	consumatore
<pre> data Type d loop forever d ← produce P(empty) append(d, buffer) V(full) </pre>	<pre> data Type d loop forever P(full) d ← take(buffer) V(empty) consume(d) </pre>

Produttore-Consumatore con un Buffer Circolare



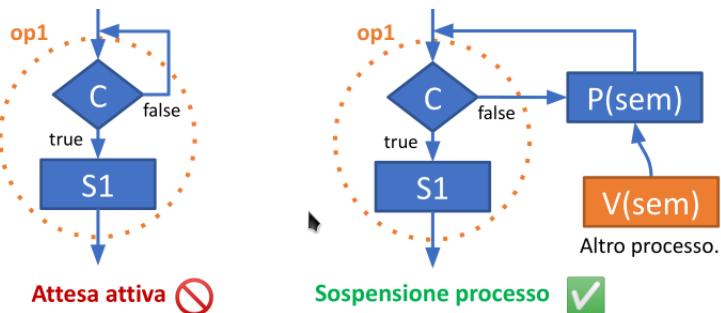
Algoritmo produttore/consumatore con buffer circolare	
produttore	consumatore
<pre> data Type array[0..max-1] buffer integer in, out ← 0 semaphore full ← 0 semaphore empty ← max data Type d loop forever d ← produce P(empty) buffer[in] ← d in ← (in + 1) mod max V(full) </pre>	<pre> data Type d loop forever P(full) d ← buffer[out] out ← (out + 1) mod max V(empty) consume(d) </pre>

Note:-

La prova di correttezza è ancora vera se e solo se si assume che ci siano solo un consumatore e un produttore.

3.2 Regione Critica Condizionale

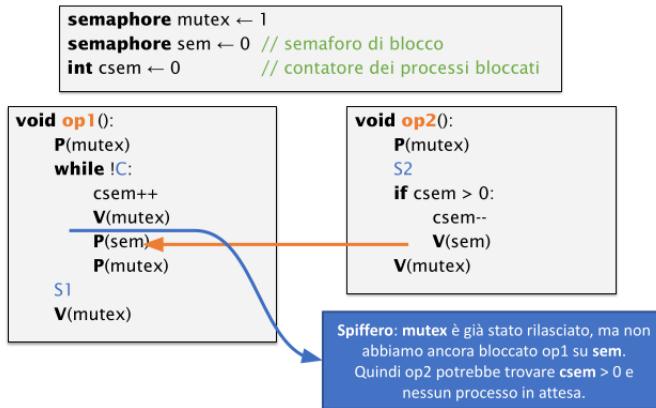
- Può accadere che un'operazione *op1* richieda di eseguire in sezione critica un codice S1 che necessita della validità di una condizione logica *C* (precondizione per op1).
- Se C non è verificata il processo deve sospendersi e uscire dalla sezione critica.



Condizioni necessarie per realizzare l'attesa:

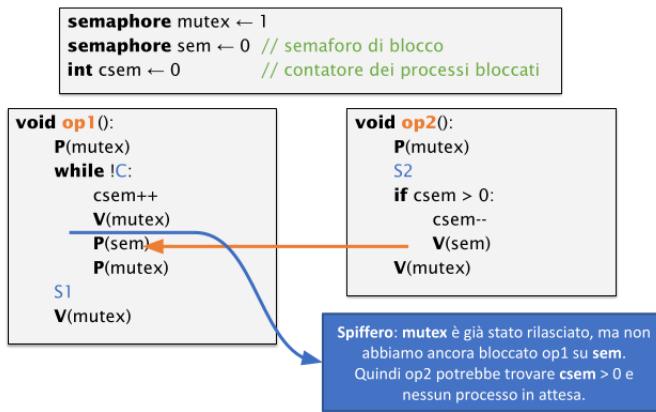
- Disporre di un semaforo sem, inizializzato a zero, il cui compito è *bloccare il processo*.
- Supporre che sia disponibile un'operazione op2 invocata da un altro processo che modifichi le variabili condivise, in modo che C diventi vera.
- Inserire in coda al codice di op2 gli statement opportuni per eseguire il risveglio del processo bloccato sul semaforo sem.
- Per sapere se ci sono processi bloccati su sem introduciamo una variabile csem, inizializzata a 0, che memorizza il numero di processi bloccati su sem.

3.2.1 Tipi di Regione Critica Condizionale



Regione critica con attesa circolare:

- Perché quando un processo bloccato sul semaforo sem viene risvegliato e ritorna in CS, ha bisogno di testare nuovamente la condizione C?
- È necessario perché l'esecuzione dell'istruzione P(mutex) da parte del processo risvegliato non assicura che il processo possa subito rientrare in CS.



Regione critica con passaggio del testimone:

- Nello schema del passaggio del testimone il processo che ha eseguito op2 sveglia il processo bloccato sul semaforo sem senza rilasciare la sezione critica (esecuzione di V(mutex)), così il processo risvegliato trova già la sezione critica bloccata (non deve eseguire P(mutex)) e non serve ritestare la condizione C.
- Quindi il processo che termina l'esecuzione di op2 passa al processo svegliato il diritto di operare in CS (da qui il nome di passaggio di testimone).

Osservazioni 3.2.1

- La regione critica con attesa circolare potrebbe essere sub-ottimale perché richiede al processo di riverificare la validità di C.
- Nel caso in cui già sapessimo che C diventa vera dopo S2, stiamo rieseguendo un controllo non necessario.
- Tuttavia l'attesa circolare è più generale del passaggio del testimone perché non assume che si debba sapere che la proprietà C sia vera dopo l'esecuzione di S2.

3.3 Semafori Privati

Partendo dal problema della regione critica condizionale:

- Più processi possono essere bloccati durante l'esecuzione dell'operazione op1, ciascuno in attesa che la propria condizione di sincronizzazione sia verificata.
- In seguito alla modifica dello stato della risorsa condivisa da parte del processo che ha eseguito op2, può accadere che le condizioni di sincronizzazione dei processi bloccati risultino tutte vere.
- Se i processi sono bloccati tutti su un semaforo sem, la scelta del processo da svegliare dipende da come è implementata V(sem).
- Per cui si deve eseguire una *politica di risveglio dei processi bloccati*.
- Per farlo basta sostituire sem con un array di semafori privati (uno per ciascun processo che può eseguire op1).

3.3.1 Problema dei Lettori/Scrittori

Un insieme di processi condivide un database. I processi sono divisi in due gruppi: i *lettori* e gli *scrittori*.

Vincoli di accesso al database:

- Gli scrittori accedono al database in mutua esclusione. Quando uno scrittore opera sul database, nessun altro scrittore o lettore può operare.
- I lettori possono accedere al database in un numero arbitrario.

La priorità viene data per garantire l'alternanza:

- Un lettore non accede al database se uno scrittore sta operando sul database o se ci sono scrittori in attesa.
- Quando uno scrittore termina e ci sono lettori in coda, questi lettori possono accedere al database (anche se ci sono altri scrittori in attesa).
- Uno scrittore non accede al database se questo è già occupato.

```

semaphore mutex ← 1
    // un semaforo privato per tutta la
    // classe dei processi (lettori o scrittori)
semaphore lett ← 0, scritt ← 0
    // numero lettori attivi
    int la ← 0
    // c'è uno scrittore attivo?
    boolean sa ← false
    // numero lettori bloccati
    int lb ← 0
    // numero scrittori bloccati
    int sb ← 0

```

Le condizioni nelle inizio_* verificano se il processo deve mettersi in attesa. Dopo essersi messi in attesa, verranno svegliati tramite passaggio di testimone della SC.

```

void inizio_scrittura():
    P(mutex)
    if la > 0 or sa:
        sb++
        V(mutex)
        P(scritt)
        sb-
        sa ← true
        V(mutex)

```

```

void fine_scrittura():
    P(mutex)
    sa ← false
    if lb > 0:
        V(lett)
    else if sb > 0:
        V(scritt)
    else:
        V(mutex)

```

```

void inizio_lettura():
    P(mutex)
    if sb > 0 or sa:
        lb++
        V(mutex)
        P(lett)
        lb-
        la++
        if lb > 0:
            V(lett)
        else:
            V(mutex)

```

```

void fine_lettura():
    P(mutex)
    la-
    if la == 0 and sb > 0:
        V(scritt)
    else:
        V(mutex)

```


4

Monitor

4.1 Introduzione

Problemi dei semafori:

- Basilari, poco strutturati e poco sicuri.
- La correttezza è onere del programmatore.
- Il linguaggio di programmazione tipicamente non fornisce un modo per strutturare l'uso dei semafori (eccetto la mutua esclusione), perché l'uso può richiedere un'interazione elaborata tra frammenti di codice di processi diversi.

Definizione 4.1.1: Monitor

Il monitor è un costrutto strutturato per linguaggi di alto livello. Si tratta di un costrutto sintattico che associa un insieme di operazioni a una struttura dati condivisa da più processi.

Note:-

Sulla struttura dati del monitor si agisce solamente attraverso operazioni definite *mutuamente esclusive* (un solo processo per volta è attivo nel monitor).

4.1.1 Definire un Monitor

monitor alpha:

<dichiarazioni variabili>
<inizializzazione delle variabili>

public void op₁():

<codice dell'operazione op₁>

public void op_n():

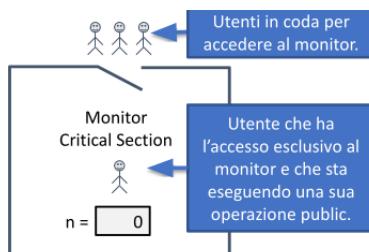
<codice dell'operazione op_n>

Definizione di monitor:

- Le operazioni public sono le sole operazioni che possono essere utilizzate dai processi per accedere alla struttura dati condivisa.
- Le variabili condivise mantengono il loro valore tra successive esecuzioni delle operazioni del monitor (*variabili permanenti*).
- Le variabili condivise sono *accessibili solo dai metodi del monitor* (sono private).
- Le operazioni non dichiarate public *non sono accessibili dall'esterno*. Sono usabili solo dalle funzioni public.

Dichiarazione di un monitor:

- La definizione di un monitor (monitor alpha:...) definisce un *tipo* (alpha).
- Per creare un'*istanza* del monitor occorre definire una variabile con quel tipo (alpha x).
- Per chiamare un'operazione *op_i* si utilizza la dot notation (*x.op_i()*).

**Osservazioni 4.1.1**

- Il monitor è un'*entità statica*: un insieme di procedure che possono essere richiamate dai singoli processi che condividono dei dati permanenti.
- Lock*: meccanismo che garantisce che un solo processo per volta sia all'interno del monitor in un certo istante.

4.1.2 Variabili di Condizione**Definizione 4.1.2: Variabile di Condizione**

Una variabile di condizione permette la sospensione di un processo all'interno del monitor nel caso in cui si debba verificare una condizione.

Osservazioni 4.1.2

- La dichiarazione di una variabile x di tipo *condition* ha la forma: condition x.
- Ogni variabile di tipo condition rappresenta una *coda FIFO* nella quale i processi si possono sospendere.
- Le procedure del monitor agiscono su tali variabili con due operazioni:
 - wait(x). Sospende il processo e lo mette nella coda x.
 - signal(x). Risveglia il primo processo in attesa nella coda x se esiste.

Wait(x):

1. Inserisce il processo q al fondo della coda.
2. Pone lo stato di q a "blocked".
3. Rilascia il controllo del monitor.

Signal(x) (se la coda non è vuota):

1. Preleva il primo processo p presente in coda.
2. Pone lo stato di p a "ready".
3. Continua nel monitor.

Semafori	Monitor
L'operazione P può essere o non essere bloccante.	L'operazione wait è sempre bloccante.
L'operazione V modifica sempre il valore del semaforo.	L'operazione signal non ha nessun effetto se la coda è vuota.
Un processo sbloccato da un'operazione V riprende subito la sua esecuzione.	Un processo sbloccato da una signal deve aspettare che il processo sbloccante lasci la CS del monitor (come avviene tale rilascio dipende dalla semantica).

Tipi di semantiche per l'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*). Questa politica viene anche detta immediate resumption requirement (IRR) ed è l'implementazione classica.

Ulteriori operazioni:

- **empty(x)**: restituisce true se la coda è vuota, false altrimenti.
- **signal_all(cond)**: risveglia tutti i processi sospesi sulla variabile cond (si può usare solo con semantica **signal_and_continue**).

Domanda 4.1

I monitor e i semafori hanno la stessa capacità di sincronizzazione? Ogni problema di sincronizzazione che si può risolvere con i semafori si può risolvere anche con il monitor e viceversa?

Risposta: sì, la dimostrazione si fa costruendo un semaforo con un monitor e costruendo un monitor con un semaforo.

4.2 Invarianti di Monitor

Definizione 4.2.1: Invariante di Monitor

Un Invariante di monitor IM è una proposizione che risulta soddisfatta immediatamente prima e subito dopo ogni operazione del monitor.

Note:-

L'invariante non è necessariamente soddisfatto durante l'operazione del monitor.

Per dimostrare che una proposizione IM è un invariante di monitor è sufficiente verificare che:

- *Passo base:* la proposizione IM è vera al momento dell'inizializzazione del monitor.
- *Passo induttivo:* supposta IM vera all'inizio della chiamata di un'operazione di monitor, si prova che IM è vera anche all'uscita dal monitor alla fine dell'esecuzione della stessa operazione.

Per esempio, per provare la correttezza nel problema dei lettori/scrittori:

- R: numero dei processi lettori che stanno leggendo, cioè hanno completato l'operazione di inizio _ lettura e non hanno ancora invocato la fine _ lettura.
- W: numero dei processi scrittori che stanno scrivendo, cioè hanno completato l'operazione di inizio _ scrittura e non hanno ancora invocato la fine _ scrittura.
- Invariante: $R > 0 \rightarrow W = 0 \wedge (W \leq 1) \wedge (W = 1 \rightarrow R = 0)$:
 - $R > 0 \rightarrow W = 0$: se ci sono lettori attivi non possono esserci scrittori attivi.
 - $(W \leq 1)$: al massimo uno scrittore può essere attivo.
 - $(W = 1 \rightarrow R = 0)$: se c'è uno scrittore attivo non possono esserci lettori attivi.

5

Modello a Rete



5.1 Introduzione

Aspetti caratterizzanti del modello a rete:

- Ogni processo può accedere esclusivamente alle risorse allocate nella propria *memoria locale* (eventualmente virtuale), non esiste *memoria condivisa*.
- La comunicazione avviene solo per *scambio di messaggi*.
- Rimane il concetto di *interleaving*.

5.1.1 Sistemi Fortemente e Debolmente Connessi

Definizione 5.1.1: Sistema Fortemente Connesso

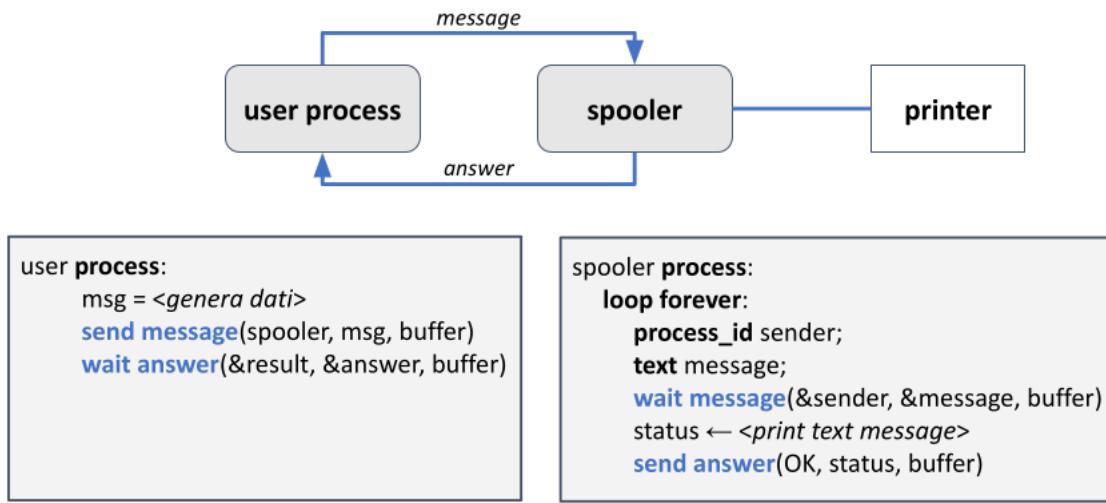
Un sistema è fortemente connesso (tightly coupled) se ogni componente necessita della completa conoscenza delle caratteristiche degli altri componenti del sistema.

Corollario 5.1.1 RC 4000

RC 4000 era un sistema operativo monoprocesso multiprogrammato progettato alla fine degli anni '60. La memoria non occupata dal nucleo del OS era suddivisa tra i processi attivi.

Il kernel associa a ogni processo una coda di messaggi, un pool di buffer per la comunicazione, e 4 primitive di kernel:

- send message(receiver, message, buffer): copia il messaggio in un buffer libero e lo manda al receiver (senza bloccare il mittente).
- wait message(sender, message, buffer): attende l'arrivo di un messaggio nella coda del processo.
- send answer(result, answer, buffer): copia la risposta nel buffer del mittente e glielo rimanda. Il processo rispondente (sender) non attende la consegna.
- wait answer(result, answer, buffer): attende la risposta dal processo che ha ricevuto il messaggio inviato con send message. L'argomento result è un codice di successo/fallimento. Ritorna il buffer usato nel pool dei buffer liberi.

**Definizione 5.1.2: Sistema Debolmente Connesso**

Un sistema è debolmente connesso (loosely coupled) se ogni componente necessita di poche o nessuna conoscenza delle caratteristiche degli altri componenti del sistema.

5.1.2 Canali di Comunicazione**Definizione 5.1.3: Canale**

Un canale è un collegamento logico mediante il quale due processi comunicano.

Note:-

È solitamente compito del nucleo dell'OS fornire l'*astrazione di canale* come meccanismo primitivo per lo scambio di informazioni.

Parametri caratterizzanti il canale:

- Il tipo di *sincronizzazione* tra i processi comunicanti.
- La designazione del canale e dei processi *sorgente* e *destinatario* di ogni comunicazione (indirizzamento).

- La tipologia del canale intesa come *direzione del flusso di dati* che un canale può trasferire (il canale è unidirezionale).

Tipologie di sincronizzazione:

- *Comunicazione asincrona:*

- Il processo sender deposita il messaggio e prosegue.
- Il processo receiver si blocca se non ci sono messaggi nel canale.
- Non c'è sincronizzazione tra processi.
- Necessità di un buffer di messaggi.

- *Comunicazione sincrona:*

- Il processo Receiver si blocca se non ci sono messaggi nel canale.
- Il processo Receiver si blocca se il Sender non è pronto a consegnare il messaggio.
- Lo scambio di messaggi è un mezzo di sincronizzazione tra i processi.
- Non c'è necessità di un buffer di messaggi.

- *Comunicazione con sincronizzazione estesa (Rendez-vous):*

- Il processo Sender aspetta il Receiver, consegna il messaggio e aspetta la risposta.
- Il processo Receiver aspetta il Sender, legge il messaggio e gli consegna la risposta.
- I due processi rimangono sincronizzati fino a che messaggio e risposta non sono consegnati.

Tipi di canale:

- Da uno a uno (link): produttore/consumatore, pipe UNIX, transputer.
- Da molti a uno (port): client/server.
- Da molti a molti (mailbox).

Note:-

Inoltre i canali possono essere *unidirezionali* o *bidirezionali*.

5.1.3 Dichiarazione di Canali

port <tipo> <identificatore>;

- Canale da molti a uno.
- È tipato ed è denotato da un identificatore.
- Ci si accede mediante dot notation.

Primitive:

- Invio (send(<value>) to <port>):
 - <port> identifica in modo univoco il canale a cui inviare il messaggio.
 - <value> è il contenuto del messaggio (dello stesso tipo di <port>).
- Ricezione (<proc> = receive(var) from <port>):
 - <port> identifica in modo univoco il canale a cui inviare il messaggio.
 - <var> è una variabile a cui assegnare il valore del messaggio ricevuto (dello stesso tipo di <port>).
 - <proc> è l'identificatore di una variabile processo a cui viene assegnato l'identificatore del processo mittente.

Note:-

Queste primitive non permettono di stabilire se la comunicazione è asincrona o sincrona; la scelta della realizzazione sincrona o asincrona dipende dal linguaggio di programmazione e l'utilizzatore deve esserne a conoscenza.

5.1.4 Guardie

Definizione 5.1.4: Guardia

La <guardia> è costituita da una coppia (<espressione booleana>; <primitiva receive>).

La guardia viene valutata e ha tre possibili esiti:

- **Fallita:** se l'espressione booleana è false.
- **Ritardata:** se l'espressione booleana è true ma la receive è bloccata poiché sul canale su cui viene eseguita non ci sono messaggi pronti.
- **Valida:** se l'espressione booleana è true e la receive può essere eseguita senza ritardi.

Comandi con guardie multiple:

- 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 viene eseguita la receive contenuta nella guardia scelta; viene quindi eseguita l'istruzione relativa al ramo scelto. Successivamente, l'esecuzione dell'intero comando viene ripetuta.
- 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 valida. A quel punto procede come nel caso precedente.
- Se tutte le guardie sono fallite l'esecuzione del blocco do..od (guardie multiple) termina.

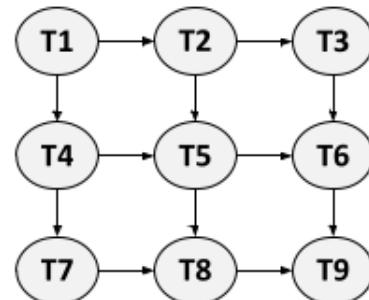
5.2 Comunicazione Sincrona

5.2.1 Transputer

Corollario 5.2.1 Transputer

Il termine Transputer identifica una famiglia di micro-computer costituiti da un unico chip contenente:

- Un processore con memoria propria.
- Quattro canali unidirezionali.

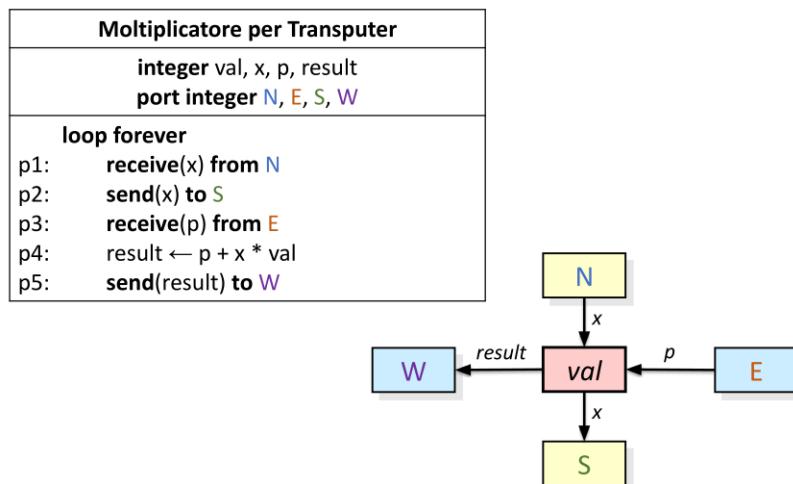


Osservazioni 5.2.1

- La comunicazione è *sincrona*, mediante istruzioni in linguaggio macchina.
- I quattro canali unidirezionali permettono l'interconnessione con altri transputer (link).
- Processi statici.

Moltiplicazione parallela di matrici:

- Passare un elemento x da NORD.
- Passare x a SUD immodificato.
- Moltiplicare x per val .
- Prendere il valore somma parziale p da EST.
- Sommare p a $x * val$.
- Mandare il risultato a OVEST.

**Note:-**

Nei linguaggi dedicati ai transputer (CSP, OCCAM,...) erano stati introdotti comandi selettivi.

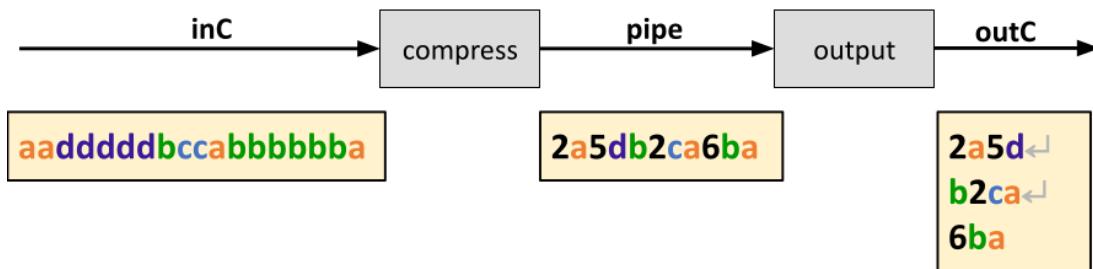
Moltiplicatore per Transputer con input selettivo
<pre style="font-family: monospace; margin: 0;">integer val, x, p, result port integer N, E, S, W</pre>
<pre style="font-family: monospace; margin: 0;">loop forever either p1: receive(x) from N // acquisisci subito da N p2: send(x) to S // invia subito ad S p3: receive(p) from E // attendi dati da E or p4: receive(p) from E // acquisisci subito da E p5: receive(x) from N // attendi dati da N p6: send(x) to S p7: result ← p + x * val p8: send(result) to W</pre>

5.2.2 Canali Sincroni - Pipe

Definizione 5.2.1: Problema di Conway

Nei sistemi operativi i canali sono chiamati pipe e sono usati per collegare insiemi di programmi esistenti. Come esempio di una computazione concorrente con pipe consideriamo una variante del problema di Conway (run-length compression):

- L'input è una sequenza di caratteri inviati da un ambiente esterno a un canale di input inC.
- L'output è la stessa sequenza mandata ad un ambiente esterno tramite un canale outC dopo aver eseguito due trasformazioni:
 - Le n ($2 \leq n \leq 9$) occorrenze consecutive di un carattere char sono sostituite dalla coppia $\langle n, \text{char} \rangle$.
 - Nella sequenza di output viene inserita un ritorno_a_capo dopo K caratteri



Algoritmo di Conway	
constant integer MAX ← 9, K ← 4 channel of integer inC, pipe, outC	
compress	output
char c, previous ← 0 integer n ← 0 inC ⇒ previous loop forever p1: inC ⇒ c p2: if (c=previous) and (n < MAX-1) p3: n ← n + 1 else p4: if n > 0 p5: pipe ← intToChar(n + 1) p6: n ← 0 p7: pipe ← previous p8: previous ← c	char c integer m ← 0 loop forever q1: pipe ⇒ c q2: outC ← c q3: m ← m + 1 q4: if m ≥ K q5: outC ← newline q6: m ← 0

Nel modello a porte con canali sincroni realizziamo quindi contemporaneamente comunicazione e sincronizzazione.

5.3 Rendez-vous e RPC

5.3.1 Rendez-vous

Definizione 5.3.1: Rendez-vous

Il processo server specifica le operazioni che offre un servizio come una sequenza di istruzioni che può comparire in un punto qualunque del suo codice. Utilizza un'istruzione di input (accept) che lo sospende in attesa di una richiesta dell'operazione da parte dei client. All'arrivo della richiesta il processo server esegue il relativo insieme di istruzioni e invia al client chiamante i risultati.

Osservazioni 5.3.1

- Meccanismo di comunicazione e sincronizzazione tra processi in cui un processo che richiede un servizio a un altro processo rimane sospeso fino al completamento del servizio richiesto (meccanismo sincrono).
- I processi rimangono sincronizzati durante l'esecuzione del servizio da parte del ricevente (*processo accettante*) e fino alla ricezione dei risultati da parte del mittente (*processo chiamante*).
- Il processo accettante non deve conoscere l'identità del processo chiamante, così il rendezvous è particolarmente utile per gestire i rapporti client/server (meccanismo asimmetrico)
- Analogia semantica con una normale chiamata di funzione. Il programma chiamante prosegue solo dopo che l'esecuzione della funzione è terminata. La differenza sostanziale sta nel fatto che la funzione (servizio) viene eseguita remotamente da un processo diverso dal chiamante.

Sintassi:

- entry <servizio> (in <par-ingresso>, out <par-uscita>): processo che offre un servizio.
- accept <servizio> (in <par-ingresso>, out <par-uscita>) {<codice del servizio >} → <codice post sincronizzazione >: accettazione di nuove richieste:
 - *Servizio*: il client è in attesa della risposta.
 - *Post sincronizzazione*: la sincronizzazione con il client è terminata ma il server può ancora fare delle operazioni.
 - Si possono usare le guardie per selezionare le richieste da servire.

Corollario 5.3.1 ADA

Il linguaggio ADA, sviluppato per conto del Department of Defense (DOD) degli USA, adotta come metodo di interazione tra i processi il rendezvous.

Rendez-vous in ADA:

- Una entry definita in un task P può essere chiamata da un altro task Q.
- Lo schema base di un task contiene una parte di specifica che definisce le operazioni entry e una parte body che consente la realizzazione di tali operazioni.
- Possono essere utilizzati per implementare semafori.

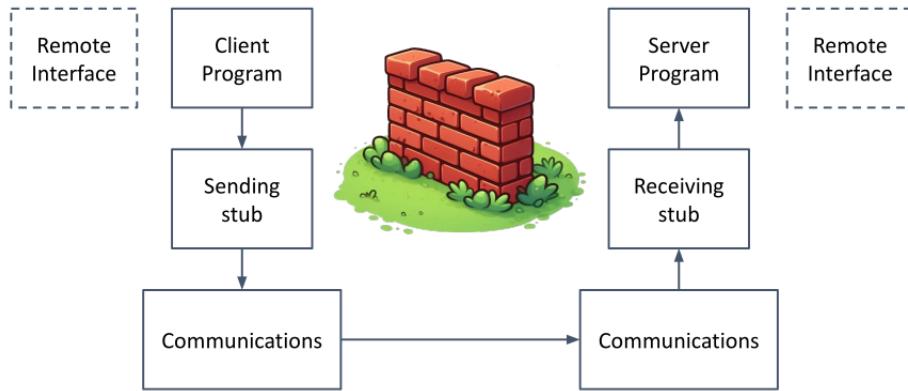
5.3.2 RPC

Definizione 5.3.2: Remote Procedure Call (RPC)

Per ogni operazione che un processo client può richiedere, viene dichiarata, lato server, una procedura. In esecuzione, per ogni nuova richiesta di operazione, viene avviato un nuovo processo (thread) servitore con il compito di eseguire la procedura corrispondente.

Note:-

RPC rappresenta esclusivamente un meccanismo di comunicazione tra processi; la possibilità che più operazioni siano eseguite concorrentemente da thread diversi comporta che si debba provvedere separatamente a una sincronizzazione tra processi servitori.



Osservazioni 5.3.2

- L'insieme delle procedure remote è definito all'interno di un componente sw (modulo), che contiene anche le variabili locali al server ed eventuali procedure e processi locali.
- I singoli moduli operano in spazi di *indirizzamento diversi* e possono quindi essere allocati su nodi distinti di una rete.
- Il server crea un thread che esegue l'operazione richiesta.
- In ogni stato è possibile che più thread concorrenti all'interno del modulo accedano a variabili interne.
- Necessità di sincronizzazione mediante semafori, monitor, ...

6

Linda

6.1 Il Modello Linda

6.1.1 Introduzione

Nei linguaggi tipo Ada o nel linguaggio dei transputer, i processi comunicano mediante un meccanismo sincrono:

- Connessi nel tempo: sincronia.
- Connessi nello spazio: indirizzo del canale.

Linda permette:

- Il *disaccorpamento* nel tempo e nello spazio.
- La *persistenza dell'informazione* per cui i dati (messaggi) possono esistere anche dopo la terminazione del processo che li ha creati.

Definizione 6.1.1: Linda

Il modello Linda definisce una struttura globale detto spazio delle tuple che si può immaginare come una grande bacheca su cui si possono attaccare dei post-it, chiamati note o tuple.

Corollario 6.1.1 Tupla

Una tupla è una sequenza tipata di dati.

Il match di due tuple si verifica se hanno lo stesso numero di valori e se i valori nelle medesime posizioni:

- O sono entrambi uguali, nel caso siano entrambi valori.
- Oppure uno è un valore e l'altro è una variabile dello stesso tipo del valore.

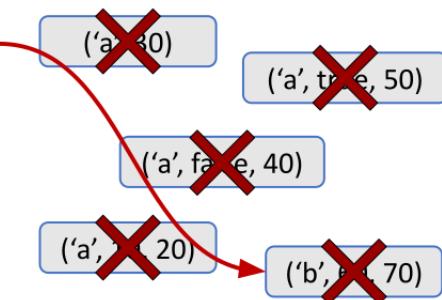
postnote (v1,v2,...)	Crea una tupla con elementi (v1,v2,...) e la aggiunge allo spazio delle tuple
removenote (x1,x2,...):	I parametri x1,x2,... possono essere variabili; se si verifica il match con una tupla dello spazio, questa viene eliminata e i suoi valori sono assegnati alle variabili x1,x2,... Se tale tupla non esiste il processo si blocca . Se ne esistono più di una, ne viene scelta una in modo non deterministico;
readnote (x1,x2,...)	Come la removenote ma senza rimozione della tupla dallo spazio

Note:-

Quindi le tuple di Linda possono contenere delle variabili e dei valori. In maniera analoga alla chiamata di procedura, il match di due tuple corrispondenti provoca il trasferimento dei valori attuali di una tupla nelle variabili dell'altra.

Esempio di statement Linda

```
int n, int m, bool b
L1: removenote('b',m,n)  blocco ok
L2: removenote('a',m,n)  ok
L3: readnote ('a',m)     ok
L4: removenote('a',n)    ok
L5: removenote('a',m,n)  blocco
L6: removenote('a',b,n)  ok
L7: removenote('a',b,m)  ok
L8: postnote('b',60,70)  ok
```



6.1.2 Rendez-vous

In linda è possibile implementare il Rendez-vous, le tuple avranno come parametri:

- Stringa contente il *nome del servizio*.
- Nome del *processo chiamante*.
- Due *parametri di input* (intero e carattere) e uno di *output*.

6.2 Il Paradigma Master/Workers

In Linda è possibile scrivere programmi flessibili in grado di adattarsi al numero di processi esistenti (load balancing) utilizzando il paradigma master/workers:

- **Master:** inserisce nello spazio tanti postnote quanti sono i tasks da eseguire.
- **Worker:** recupera uno dei postnote ed esegue la computazione.

master process

```

postnote ('A', 1, (1,2,3))
postnote ('A', 2, (4,5,6))
postnote ('A', 3, (7,8,9))
postnote ('B', 1, (1,0,1))
postnote ('B', 2, (0,1,0))
postnote ('B', 3, (2,2,0))

postnote ("Next",1)          righe prima matrice
for I in 1..3:
    for J in 1..3:
        removenote ('C', I: integer, J: integer, C: integer)
        print C(I,J)
    
```

NOTA: le tuple contengono l'intera riga/colonna delle matrici A e B, mentre per la matrice C contengono un singolo valore.

worker processes

```

loop forever:
removenote ("Next", Element)
postnote ("Next", Element + 1)
exit when Element > 3*3
I := (Element -1)/3 + 1           riga corrispondente
J := (Element -1) mod 3 + 1       colonna corrispondente
readnote ('A', I, row_I)         legge l'intera riga I di A
readnote ('B', J, col_J)         legge colonna J di B
X := prodotto_interno (row_I, col_J) prodotto riga per colonna
postnote ('C', I, J, X)
    
```

Può esserci qualunque numero di workers concorrentemente in esecuzione, e ognuno si prende uno o più task diversi da svolgere.

