



---

# Teoria della Replicazione

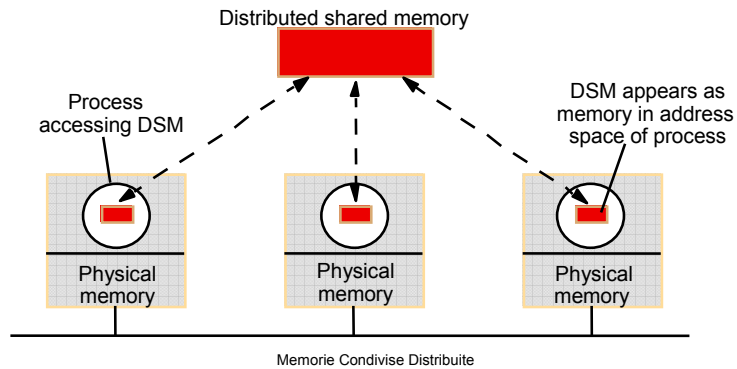
## Memorie Distribuite Condivise

Prof. Roberto Baldoni, Ing. Alessia Milani



# Memorie Distribuite Condivise

- Astrazione che permette la condivisione di dati tra più computer che non condividono un'unica memoria fisica.





# Modello di Memoria Condivisa

- N processi sequenziali  $P=\{p_1, p_2, \dots, p_n\}$  interagenti attraverso una memoria condivisa  $M$ .
- $M$  è composta da  $m$  locazioni  $x_1, x_2, \dots, x_m$
- Ogni locazione è inizializzata a  $\perp$
- Ogni processo  $p_i$  può eseguire le seguenti operazioni su  $M$ :
  - Lettura,  $r_i(x_h)v$ .  $p_i$  legge il valore  $v$  registrato nella locazione  $x_h$ .
  - Scrittura,  $w_i(x_h)v$ .  $p_i$  scrive un nuovo valore  $v$  nella locazione  $x_h$ .

Memorie Condivise Distribuite

3

Si assume che non possano esistere due scritture che scrivono uno stesso valore.  
(SEMPLIFICAZIONE DEL MODELLO)



---

# Criteri Di Consistenza: Specifica



# Process Order

- $h_i$  è l'insieme di operazioni eseguite dal processo  $p_i$
- **PROCESS ORDER**: Date  $o_1$  e  $o_2$  eseguite da  $p_i$ , se  $p_i$  esegue prima  $o_1$  e poi  $o_2$  si dice che  $o_1$  precede  $o_2$  nell'ordine del processo  $p_i$ ,  $o_1 \rightarrow_1 o_2$ .

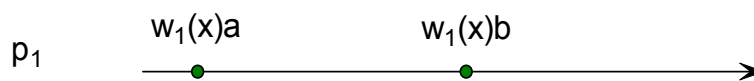


- $w_1(x)a \rightarrow_1 w_2(x)b$



## Local history

- Sia  $h_i$  l'insieme di operazioni eseguite da  $p_i$
- **DEF.** Una *local history*  $\hat{h}_i$  di un processo  $p_i$  è la sequenza di operazioni eseguite da  $p_i$  ordinata secondo  $\rightarrow_i$ .

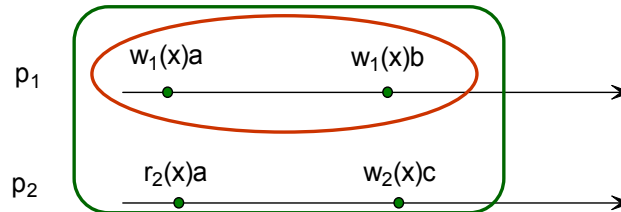


- $\hat{h}_1 = w_1(x)a, w_1(x)b$



# History

- **DEF.** Una **history H** è l'unione di tutte le  $h_i$  dei processi del sistema.



- $h_1 = \{w_1(x)a, w_1(x)b\}$   
 $h_2 = \{r_2(x)a, w_2(x)c\}$   
 $H = h_1 \cup h_2 = \{w_1(x)a, w_1(x)b, r_2(x)a, w_2(x)c\}$

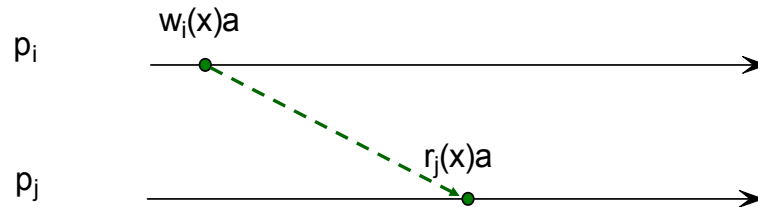


## Read-from order

- La relazione di *read-from order*,  $\rightarrow_{ro}$  lega due operazioni  $o_1, o_2$  eseguite da processi distinti del sistema,  $p_i$  e  $p_j$ .
- $o_1 \rightarrow_{ro} o_2$  se e solo se:
  - $o_1 = w_i(x)v$  e  $o_2 = r_j(x)v$
  - $\forall o_2$  esiste **al più una**  $o_1$  t.c.  $o_1 \rightarrow_{ro} o_2$
  - se  $o_2 = r_j(x)v$  e **non esiste** un  $o_1$  t.c.  $o_1 \rightarrow_{ro} o_2$  allora  **$v = \perp$** .



# Read-from order scenario



■  $w_i(x)a \rightarrow_{ro} r_j(x)a$



# Execution History

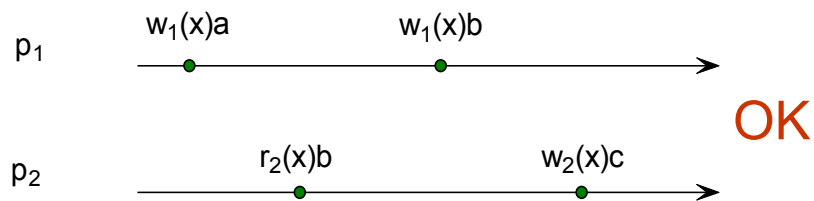
- **DEF.** Un' **Execution History** è un ordinamento parziale  $\hat{H}=(H, \rightarrow_H)$  tale che:
  - $H = \cup_i h_i$
  - $o_1 \rightarrow_H o_2$  se una delle tre condizioni è soddisfatta:
    - $\exists p_i : o_1 \rightarrow_i o_2$
    - $o_1 \rightarrow_{r_0} o_2$
    - $\exists o_3 : o_1 \rightarrow_H o_3 \text{ e } o_3 \rightarrow_H o_2$



## Lettura Legale (1)

- **DEF.** Data una  $\hat{H}=(H, \rightarrow_H)$ , una lettura  $r(x)v \in \hat{H}$  è **legale** se :

1.  $\exists w(x)v$  tale che  $w(x)v \rightarrow_H r(x)v$
2.  $\neg \exists w'(x)v : w(x)v \rightarrow_H w'(x)v \rightarrow_H r(x)v$

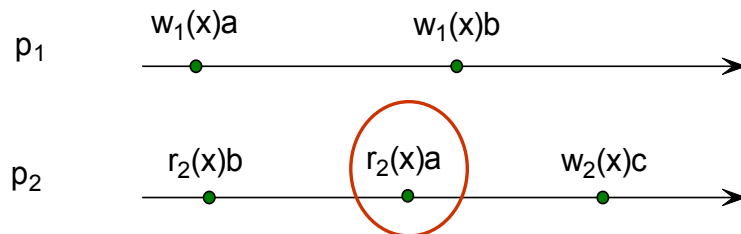


Criteria di Consistenza: Notazioni

11



## Lettura Legale (2)



- $r_2(x)a$  **NON** è una scrittura **LEGALE**



## Execution History Legale

- **DEF.** Una **Execution History**  $\hat{H} = (H, \rightarrow_H)$  è legale se tutte le sue letture sono legali.
- In una **Execution History** legale nessuna operazione di lettura può restituire un valore sovrascritto.

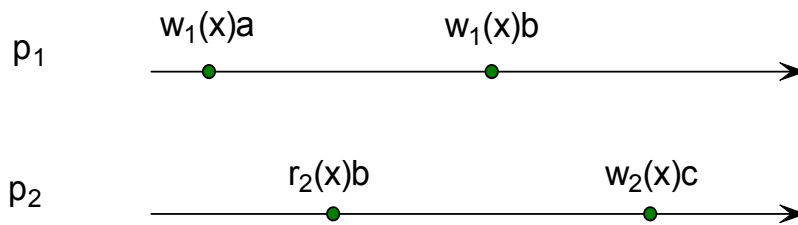


## Estensione Lineare

- **DEF.**  $\hat{S}=(S, \rightarrow_S)$  è un'estensione lineare di un ordinamento parziale  $\hat{H}=(H, \rightarrow_H)$  se:
  1.  $S=H$
  2.  $o_1 \rightarrow_H o_2 \Rightarrow o_1 \rightarrow_S o_2$  ( $\hat{S}$  mantiene l'ordinamento di ogni coppia di operazioni in  $\hat{H}$ )
  3.  $\rightarrow_S$  definisce un ordinamento totale



# Estensione Lineare Esempio



■  $\hat{S} = w_1(x)a, w_1(x)b, r_2(x)b, w_2(x)c$



# Sequential Consistency (1)

---

- Proposta da Lamport nel 1979 per definire un criterio di correttezza per sistemi di memorie condivise multiprocessore.
- “Il risultato di un’esecuzione è lo stesso di quello che si otterrebbe se le operazioni di tutti i processi fossero eseguite in un qualche ordine sequenziale, mantenendo per ogni processo l’ordinamento specificato dal suo programma.”

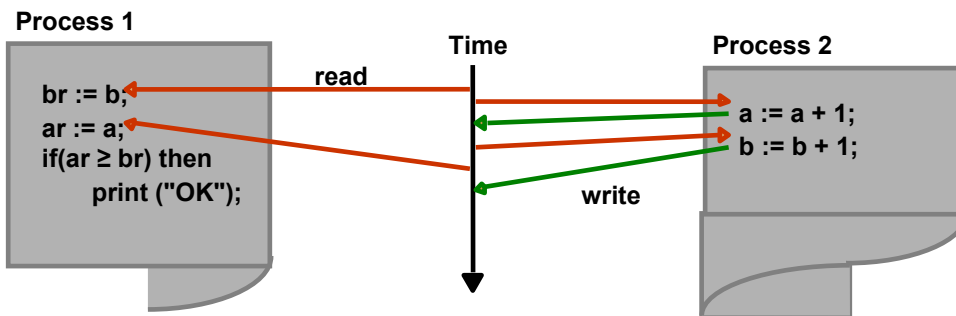


## Sequential Consistency (2)

- **DEF.** Un'execution History  $\hat{H}=(H, \rightarrow_H)$  è Sequential Consistent se esiste un'estensione lineare  $\hat{S}$  in cui tutte le letture sono legali.



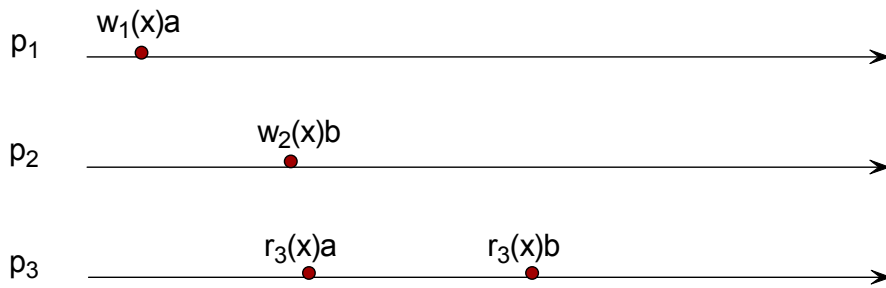
# Sequential Consistency (3)



- a e b inizializzati a 0.
- La combinazione  $ar=0$  e  $br=1$  non può verificarsi se il sistema impone la Sequential Consistency.



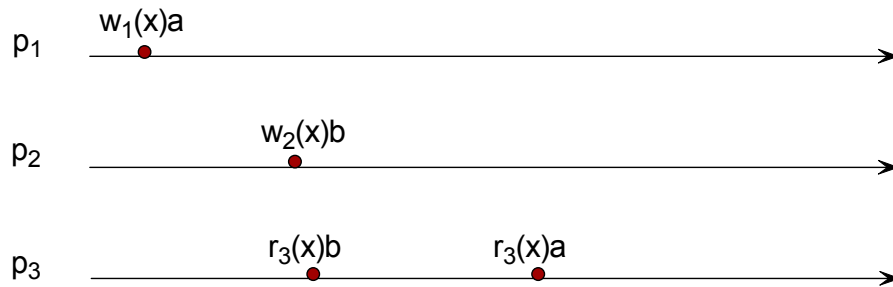
# Esempio 1



■  $\hat{S} : w_1(x)a, r_3(x)a, w_2(x)b, r_3(x)b.$



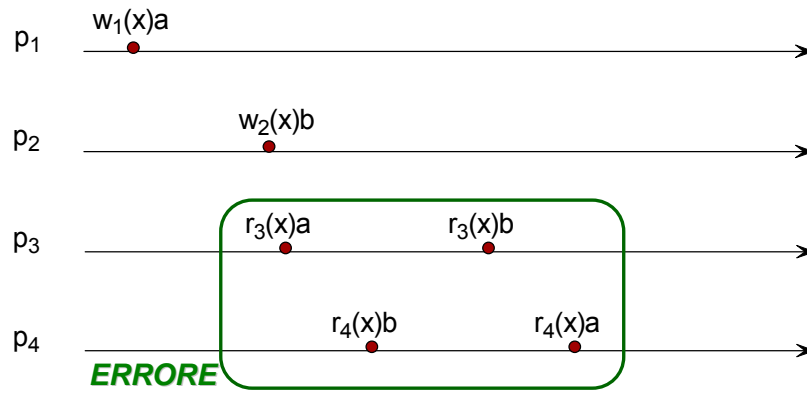
## Esempio 2



■  $\hat{S} : w_2(x)b, r_3(x)b, w_1(x)a, r_3(x)a .$

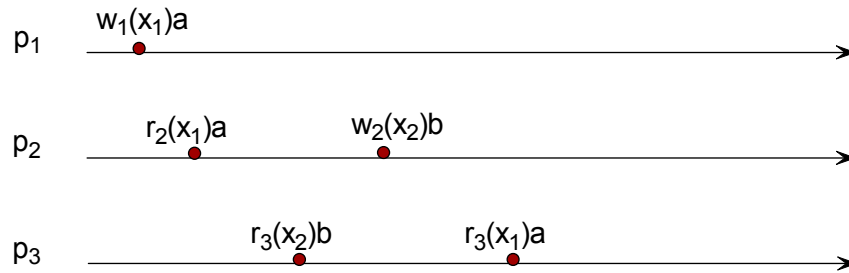


# Esempio 3





## Esempio 4 (due variabili)



■  $\hat{S} = w_1(x_1)a, r_2(x_1)a, w_2(x_2)b, r_3(x_2)b, r_3(x_1)a$



## Causal Consistency (Ahamad 1991)

- **DEF.**  $\hat{H}_i$  è il sottoinsieme di  $\hat{H}$  costituito da tutte le scritture di  $\hat{H}$  e da tutte le letture invocate dal processo  $p_i$ .
- **DEF.** Data una  $\hat{H}=(H, \rightarrow_H)$ ,  $\hat{H}$  è *causalmente consistente* se, per ogni processo  $p_i$ , tutte le operazioni di lettura di  $\hat{H}_i$  sono legali.
- Ogni processo può vedere una differente estensione lineare  $\hat{S}_i$  dell'ordinamento parziale indotto da  $\rightarrow_H$ .

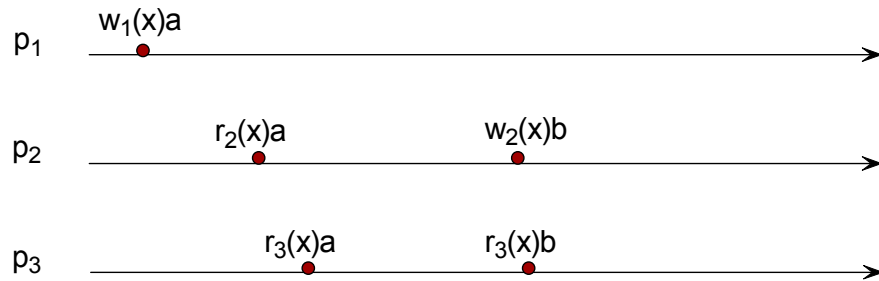


## Causal Consistency: Causal Order

- $o_1 \rightarrow_{co} o_2$  se una delle seguenti condizioni è verificata:
  - $o_1 \rightarrow_i o_2$  per qualche  $p_i$
  - $o_1 \rightarrow_{ro} o_2$  ( $o_1$  legge il valore scritto da  $o_2$ )
  - $\exists o_3 : o_1 \rightarrow_{co} o_3$  e  $o_3 \rightarrow_{co} o_2$
- Se  $\neg(o_1 \rightarrow_{co} o_2)$  e  $\neg(o_2 \rightarrow_{co} o_1)$ , le due operazioni sono causalmente concorrenti,  $o_1 \parallel_{co} o_2$ .



# Esempio 1



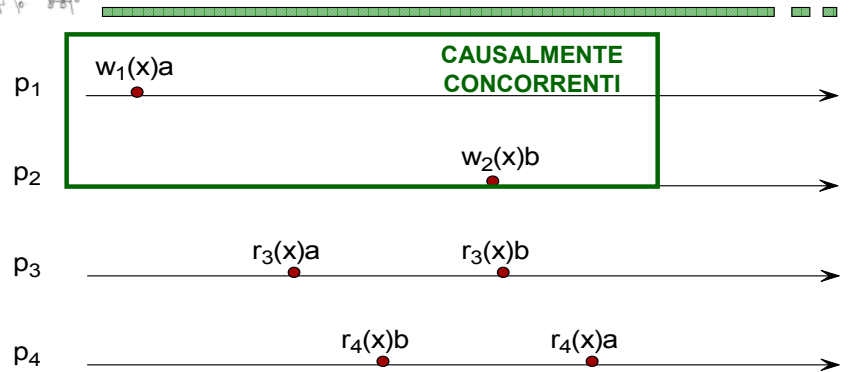
- $\hat{S}_1 = w_1(x)a, w_2(x)b$
- $\hat{S}_2 = w_1(x)a, r_2(x)a, w_2(x)b$
- $\hat{S}_3 = w_1(x)a, r_3(x)a, w_2(x)b, r_3(x)b$

Criteria di Consistenza: Causal Consistency

25



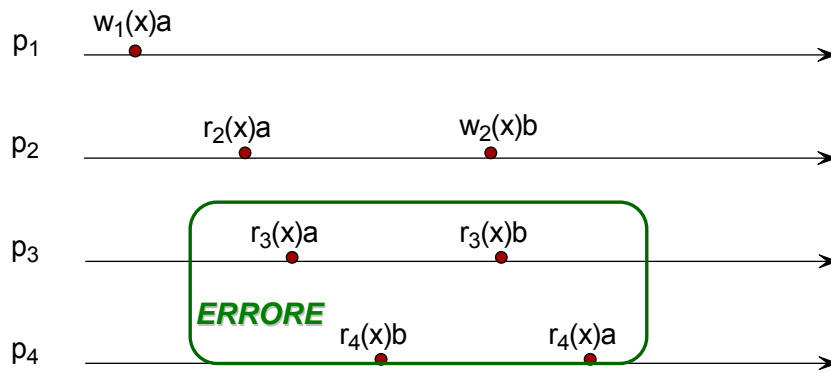
## Esempio 2



- Scritture causalmente concorrenti possono essere viste da processi diversi in un qualsiasi ordine.



## Esempio 3



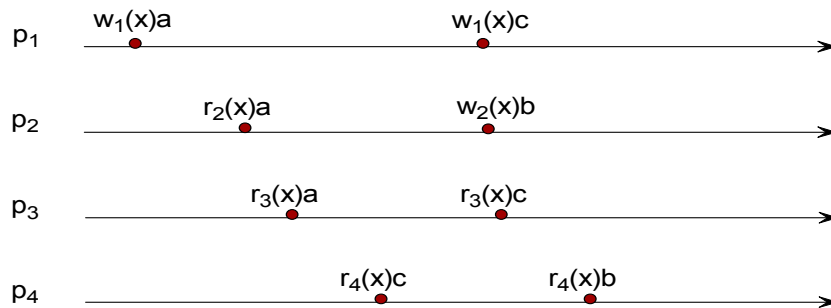
- $w_1(x)a \rightarrow_{co} w_2(x)b \Rightarrow$  ogni processo dovrà leggere prima **a** e poi **b** (NO il contrario)

Criteria di Consistenza: Causal Consistency

27



## Esempio 4



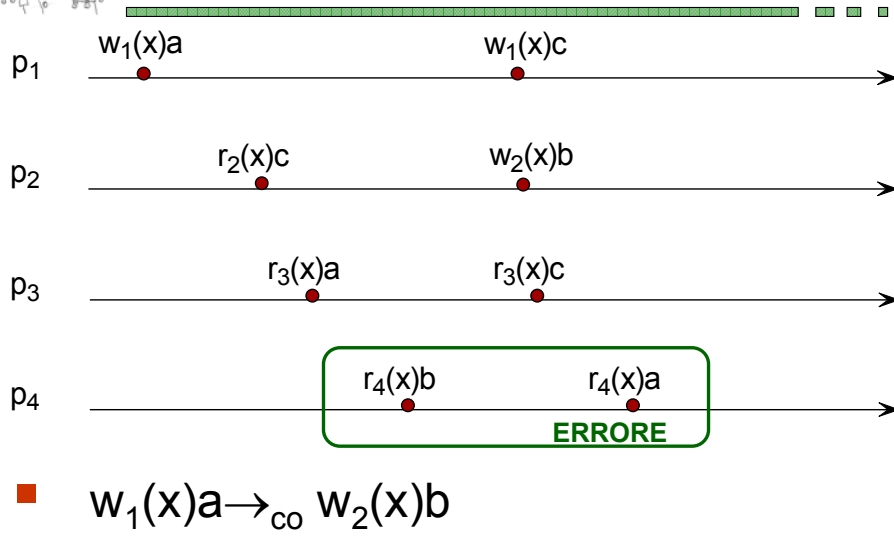
- $\hat{S}_1 = w_1(x)a, w_1(x)c, w_2(x)b$
- $\hat{S}_2 = w_1(x)a, r_2(x)a, w_1(x)c, w_2(x)b$
- $\hat{S}_3 = w_1(x)a, r_3(x)a, w_1(x)c, r_3(x)c, w_2(x)b$
- $\hat{S}_4 = w_1(x)a, w_1(x)c, r_4(x)c, w_2(x)b, r_4(x)b$

Criteria di Consistenza: Causal Consistency

28



## Esempio 5

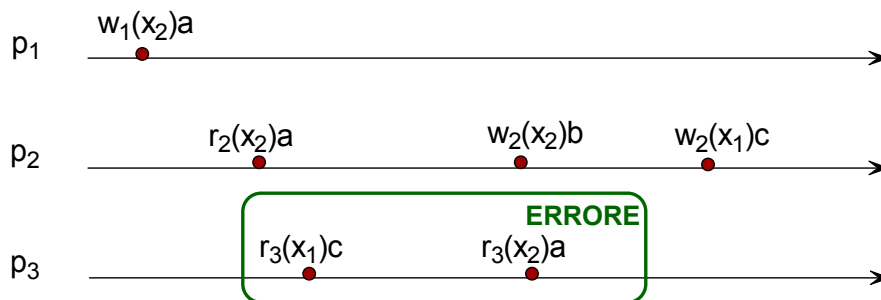


Criteria di Consistenza: Causal Consistency

29



## Esempio 6 (2 variabili)



- $w_1(x_2)a \rightarrow_{co} w_2(x_2)b \rightarrow_{co} w_2(x_1)c$
- Quando leggo  $x_1=c$ ,  $x_2=a$  è già stato sovrascritto da  $x_2=b$ .

Criteri di Consistenza: Causal Consistency

30



# PRAM (Pipelined RAM)

- **DEF.** Data  $\hat{H}=(H, \rightarrow_H)$  definiamo  $\hat{H}'=(H', \rightarrow_{H'})$  nel seguente modo:
  - $H'=H$
  - $o_1 \rightarrow_{H'} o_2$  se una delle seguenti condizioni è verificata:
    1.  $\exists p_i : o_1 \rightarrow_i o_2$
    2.  $o_1 = w_i(x)v$  e  $o_2 = r_j(x)v$
    3.  $\exists o_3 : o_1 \rightarrow_i o_3$  e  $o_3 \rightarrow_i o_2$  (**DIFFERENTE** da  $\rightarrow_H$ )

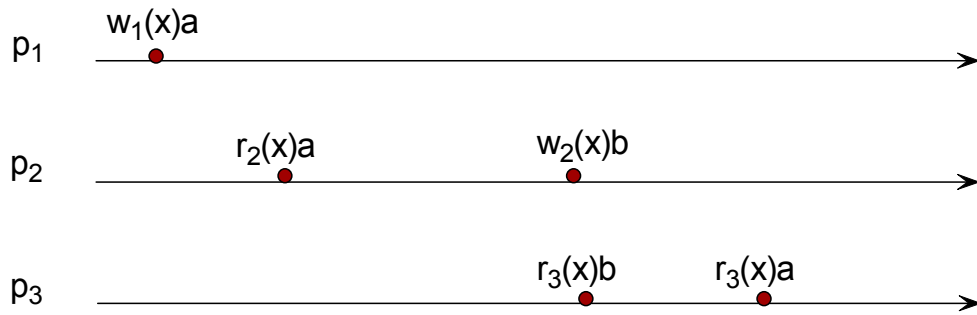


## PRAM (2)

- Sia  $\hat{H}'_i$  la sotto-history ottenuta da  $\hat{H}'$  eliminando tutte le operazioni di lettura non invocate da  $p_i$ .
- **DEF.**  $\hat{H}$  è PRAM consistente se, per ogni processo  $p_i$ , tutte le letture di  $\hat{H}'_i$  sono legali.



# Esempio 1



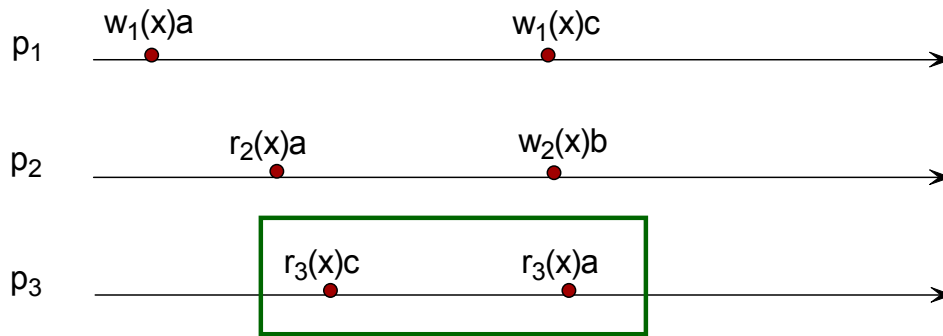
- $\hat{S}_1 = w_1(x)a, w_2(x)b$
- $\hat{S}_2 = w_1(x)a, r_2(x)a, w_2(x)b$
- $\hat{S}_3 = w_2(x)b, r_3(x)b, w_1(x)a, r_3(x)a$

Criteria di Consistenza: PRAM Consistency

33



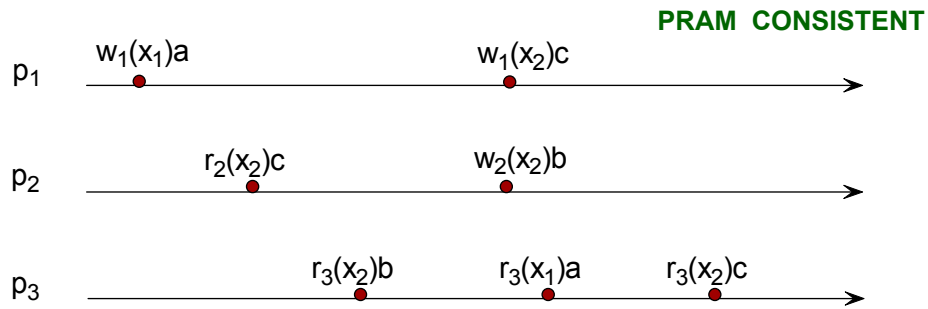
## Esempio 2



**NO PRAM CONSISTENT**



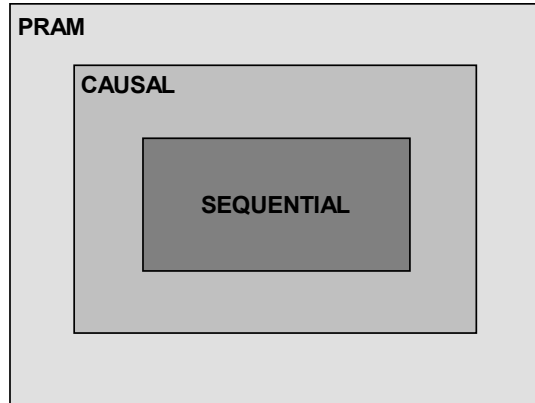
## Esempio 3 (2 variabili)



- $\hat{S}_1 = w_1(x_1)a, w_1(x_2)c, w_2(x_2)b$
- $\hat{S}_2 = w_1(x_1)a, w_1(x_2)c, r_2(x_2)c, w_2(x_2)b$
- $\hat{S}_3 = w_2(x_2)b, r_3(x_2)b, w_1(x_1)a, r_3(x_1)a, w_1(x_2)c, r_3(x_2)c$



# Confronto Tra i Criteri

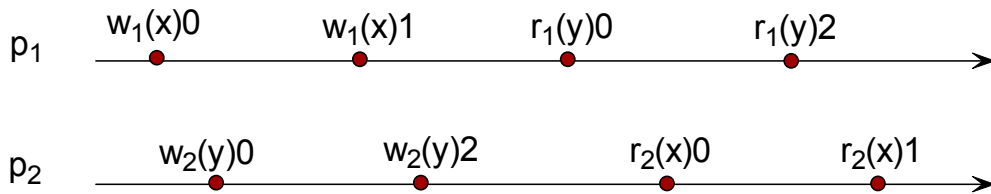


Criteri di Consistenza

36



## Confronto tra i criteri: Esempio 1

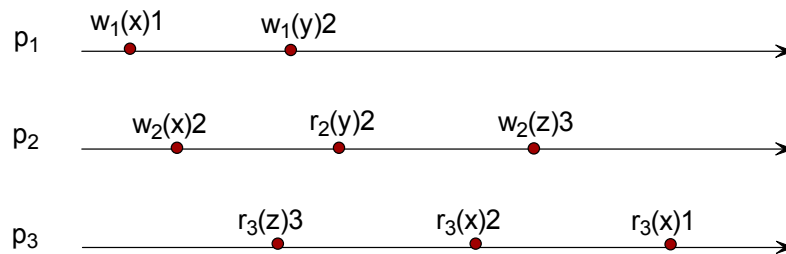


- $\hat{S}_1 = w_1(x)0, w_1(x)1, w_2(y)0, r_1(y)0, w_2(y)2, r_1(y)2$
- $\hat{S}_2 = w_1(x)0, w_2(y)0, w_2(y)2, r_2(x)0, w_1(x)1, r_2(x)1$

**NO SEQUENTIAL, SI CAUSAL E PRAM**



## Confronto tra i criteri: Esempio 2



- $\hat{S}_1 = w_1(x)1, w_1(y)2, w_2(x)2, w_2(z)3$
- $\hat{S}_2 = w_1(x)1, w_1(y)2, w_2(x)2, r_2(y)2, w_2(z)3$
- $\hat{S}_2 = w_2(x)2, w_2(z)3, r_3(z)3, r_3(x)2, w_1(x)1, r_3(x)1, w_1(y)2$

**NO SEQUENTIAL E CAUSAL, SI PRAM**



---

# Criteri Di Consistenza Protocolli



## Modello di sistema

- N processi (N computer su ognuno dei quali è in esecuzione un singolo processo)
- Ogni processo ha una copia locale dello spazio di indirizzi condivisi (replicazione totale)
- I processi comunicano attraverso una rete di connessione
- Modello a scambio di messaggi
- Canali affidabili e asincroni (perfect link)
- No guasti

Criteria di Consistenza: Protocolli

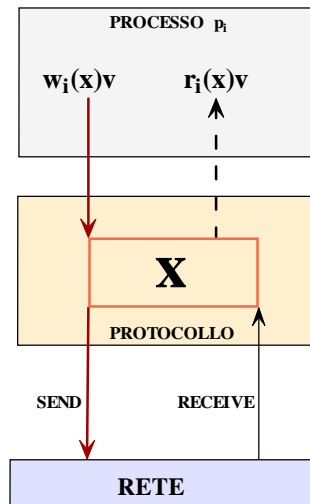
40

Il modello di memoria condivisa viene implementato attraverso un insieme finito di processi sequenziali  $\Pi = \{p_1, p_2, \dots, p_n\}$ . Ogni processo è in esecuzione su un computer distinto. I processi comunicano scambiando messaggi attraverso una rete di connessione. I canali di comunicazione sono:

1. **affidabili**: un qualsiasi messaggio spedito verrà ricevuto esattamente una volta e non sarà corrotto;
2. **asincroni**: il ritardo subito da un messaggio non è noto a priori.



# Architettura Di Processo



41

L'esecuzione di ogni operazione da parte di un processo  $p_i$  genera un insieme di eventi localmente ad uno o più processi. Gli eventi possibili sono:

- **send<sub>i</sub>(w<sub>i</sub>(x)v)**: spedizione (locale a  $p_i$ ) del messaggio di aggiornamento relativo a  $w_i(x)v$ ,
  - **receive<sub>j</sub>(w<sub>i</sub>(x)v)**: ricezione (locale a  $p_j$ ) del messaggio di aggiornamento relativo a  $w_i(x)v$ ,
  - **apply<sub>j</sub>(w<sub>i</sub>(x)v)**: applicazione dell'aggiornamento relativo a  $w_i(x)v$  sulla locazione  $x$  della memoria locale a  $p_j$ ,
  - **return<sub>i</sub>(x,v)** : restituzione a  $p_i$  del valore registrato nella sua copia locale della variabile  $x$ .
1. Quando  $p_i$  esegue un'operazione di scrittura  $w_i(x)v$  viene generato il seguente evento di spedizione **send<sub>i</sub>(w<sub>i</sub>(x)v)**, cioè viene spedito a tutti gli altri processi nel sistema il messaggio di aggiornamento corrispondente alla scrittura invocata. Tale evento di broadcast segna l'inizio della propagazione di  $w_i(x)v$  all'interno del sistema. Quando un processo  $p_j$  riceve la notifica di  $w_i(x)v$ , un evento di **receive<sub>j</sub>(w<sub>i</sub>(x)v)** è generato a  $p_j$ . A questo punto  $p_j$  schedula l'applicazione della scrittura sulla sua memoria locale. Quando  $p_j$  applica la scrittura, un evento di **apply<sub>j</sub>(w<sub>i</sub>(x)v)** viene generato.
  2. Quando  $p_i$  esegue un'operazione di lettura  $r_i(x)v$ , viene generato un evento di **return<sub>i</sub>(x,v)**.

In pratica, una **local history** produce una sequenza di eventi localmente ad ogni processo  $p_i$ , denotata  $E_i$ , ordinati secondo la relazione di "happened-before" introdotta da Lamport.

Sia  $E = \cup E_i$ , definiamo computazione distribuita, denotata  $\hat{E} = (E, \rightarrow)$ , l'ordinamento parziale indotto dalla happened-before su  $E$ .



# Protocollo di Ahamad

Implementa una memoria  
condivisa distribuita a  
**Consistenza Causale.**

Protocollo di Ahamad

42

Per garantire la consistenza causale, il protocollo deve assicurare che gli aggiornamenti corrispondenti a scritture legate da una relazione di causalità ( $w_i(x)a \rightarrow_{co} w_j(x)b$ ) siano applicati localmente ad ogni processo in modo tale da garantire che tutti i processi le vedano nello stesso ordine e che tale ordine rispetti le precedenze causali (tutti i processi dovranno prima applicare  $w_i(x)a$  e poi  $w_j(x)b$ ).

**IDEA:** Date due scritture  $w_i(x)a$ ,  $w_j(x)b$  tali che  $w_i(x)a \rightarrow_{co} w_j(x)b$  in  $\hat{H}$  allora si avrà  $send_i(w_i(x)a) \rightarrow send_j(w_j(x)b)$  nella corrispondente  $\hat{E}$ .

Quindi:

se dati due eventi di spedizione  $send_i(w_i(x)a)$ ,  $send_j(w_j(x)b)$  tali che hanno lo stesso processo destinatario  $p_k$  e tali che  $send_i(w_i(x)a) \rightarrow send_j(w_j(x)b)$ , il protocollo garantisce una consegna dei corrispondenti messaggi a  $p_k$  che rispetti l'ordine di spedizione (deve essere consegnato prima il messaggio corrispondente a  $w_i(x)a$  e poi quello relativo a  $w_j(x)b$ ) allora il protocollo assicura il criterio di consistenza causale.



## Strutture Dati Locali ai Processi

- Ogni processo  $p_i$  mantiene :
  - Una copia privata della memoria condivisa (astrazione)  $M$
  - Un vector clock  $Write_{co}[1..n]$ : array di interi di dimensione  $n$ , inizializzato a tutti 0. Ogni scrittura è associata ad un  $Write_{co}(w_i(x)a.Write_{co})$ . Usato come timestamp per i messaggi di aggiornamento.
    - Dati due vector clock  $W_1$  e  $W_2$ ,  $W_1 \leq W_2 \Leftrightarrow \forall i W_1[i] \leq W_2[i]$
    - Dati due vector clock  $W_1$  e  $W_2$ ,  $W_1 < W_2 \Leftrightarrow \forall i W_1[i] \leq W_2[i]$  ed  $\exists j$  tale che  $W_1[j] < W_2[j]$ .

Protocollo di Ahamad

43

Per tener traccia delle dipendenze di happened-before tra gli eventi della computazione distribuita viene usato un sistema di vector clock adattati.

Ogni processo  $p_i$  gestisce localmente un proprio vector clock  $Write_{co}[1..n]$  inizializzato a tutti zeri. Tale vector clock viene usato come timestamp nei messaggi di aggiornamento associati alle scritture. Il  $Write_{co}$  inserito nei messaggi di aggiornamento è denotato  $W_{co}$

Il processo  $p_i$  **aggiorna il suo  $Write_{co}$  nel seguente modo:**

1. Quando  $p_i$  scrive, incrementa di uno la locazione  $i$ -esima del vector clock,  $Write_{co}[i] := Write_{co}[i] + 1$ .
2. Quando  $p_i$  applica la scrittura  $w_j(x)v$  aggiorna la  $j$ -esima locazione del suo vector clock ponendola uguale al valore della  $j$ -esima locazione del vector clock contenuto nel corrispondente messaggio di aggiornamento,  $Write_{co}[j] := W_{co}[j]$ .



# Procedure Eseguite da $p_i$

## ■ Scrittura

```
WRITE ( $x_h, v$ )  
1  $Write_{co}[i] := Write_{co}[i] + 1;$            % tracking  $\mapsto p_{o_i}$  %  
2 send [ $m(x_h, v, W_{co})$ ] to  $\Pi - p_i;$      % send event %  
3 apply( $v, x_h$ );                             % apply event %
```

## ■ Lettura

```
READ ( $x_h$ )  
1 return( $x_h$ )
```

Protocollo Ahamad

44

### **SCRITTURA**

Quando un processo  $p_i$  invoca un'operazione di scrittura  $w_i(x)v$  esegue atomicamente i passi 1,2,3 della procedura di scrittura. Nel dettaglio:

1. incrementa la  $i$ -esima locazione del suo vector clock per tener traccia del process order
2. invia il messaggio di aggiornamento a tutti gli altri processi nel sistema. Il messaggio di aggiornamento contiene le seguenti informazioni: la variabile che deve essere aggiornata,  $x_h$ , il valore con il quale aggiornarla,  $v$ , e il valore corrente del vector clock locale a  $p_i$ ,  $Write_{co}$  (denotato  $W_{co}$ )
3. applica l'aggiornamento alla copia locale della memoria.

### **LETTURA**

Quando un processo  $p_i$  invoca un'operazione di lettura  $r_i(x)v$ , gli viene restituito il valore registrato nella locazione  $x$  della copia locale della memoria.



# Thread di Sincronizzazione

- 1 Upon the arrival of  $m(x_h, v, W_{co})$  from  $p_u$
- 2 wait until  $((\forall t \neq u \in W_{co}, W_{co}[t] \leq Write_{co}[t]) \text{ and } (Write_{co}[u] = W_{co}[u] + 1))$
- 3  $Write_{co}[u] = W_{co}[u];$
- 4  $apply(v, x_h).$

Protocollo Ahamad

45

## **SINCRONIZZAZIONE**

Ogni volta che un messaggio di aggiornamento  $m(x_h, v, W_{co})$  associato a  $w_u(x_h)v$  e inviato dal processo  $p_u$  arriva al destinatario  $p_i$ , viene attivato un nuovo thread di sincronizzazione. Il codice di tale thread è quello mostrato nella slide. Nel dettaglio:

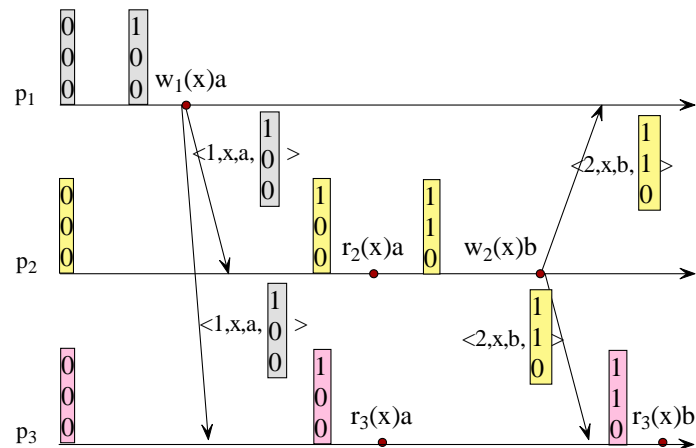
il thread attende fino a quando la condizione alla linea 2 non è verificata, in modo tale da garantire il rispetto della relazione di causalità tra le operazioni. In particolare, si verifica che ogni componente del vector clock locale a  $p_i$ , fatta eccezione per la  $u$ -esima, sia maggiore o uguale alla corrispondente locazione del vector clock contenuto nel messaggio di aggiornamento:  $W_{co}[t] \leq Write_{co}[t] \quad t \neq u$ . In questo modo, si verifica che il processo  $p_i$  prima di applicare  $w_u(x_h)v$  abbia applicato tutte le scritture applicate dal processo  $p_u$  prima di invocare  $w_u(x_h)v$ .

Analogamente il controllo della condizione  $Write_{co}[u] = W_{co}[u] + 1$  serve a garantire il process order. In pratica il processo  $p_i$  può applicare la  $k$ -esima scrittura di  $p_u$  solo se ha già applicato la  $(k-1)$ -esima scrittura di  $p_u$  e per induzione tutte quelle che la precedono causalmente.

Una volta verificata la condizione, viene aggiornata la  $u$ -esima locazione del vector clock per tener traccia della nuova scrittura applicata e viene infine aggiornata la copia della locazione  $x_h$  locale a  $p_i$ .



## Scenario Di Funzionamento (1)



Protocollo Ahamad

46

Dato lo scenario in figura abbiamo:

$$\hat{h}_1 = w_1(x)a;$$

$$\hat{h}_2 = r_2(x)a, w_2(x)b;$$

$$\hat{h}_3 = r_3(x)a, r_3(x)b;$$

Poiché  $w_1(x)a \rightarrow_{ro} r_2(x)a$  e  $r_2(x)a \rightarrow_2 w_2(x)b$  si ha  $w_1(x)a \rightarrow_{co} w_2(x)b$ .

**Nessun processo deve poter leggere prima  $x=b$  e poi  $x=a$ .**

Quando il processo  $p_1$  scrive  $w_1(x)a$  incrementa la prima locazione del suo vector clock locale che diventa  $[1,0,0]$  e lo inserisce nel messaggio di aggiornamento che spedisce agli altri processi. I processi  $p_2$  e  $p_3$  possono applicare l'aggiornamento contenuto nel messaggio in quanto il corrispondente vector clock soddisfa la condizione di wait del thread di sincronizzazione (pag 45 linea 2).

Aggiornano la copia locale della memoria e i loro vector clock, ponendoli uguali a  $[1,0,0]$  (per tener traccia della scrittura applicate alle proprie copie locali della memoria).

Analogamente quando il processo  $p_2$  scrive, incrementa di uno la locazione 2 del suo vector clock locale (per tener traccia del process order), che diventa  $[1,1,0]$ .

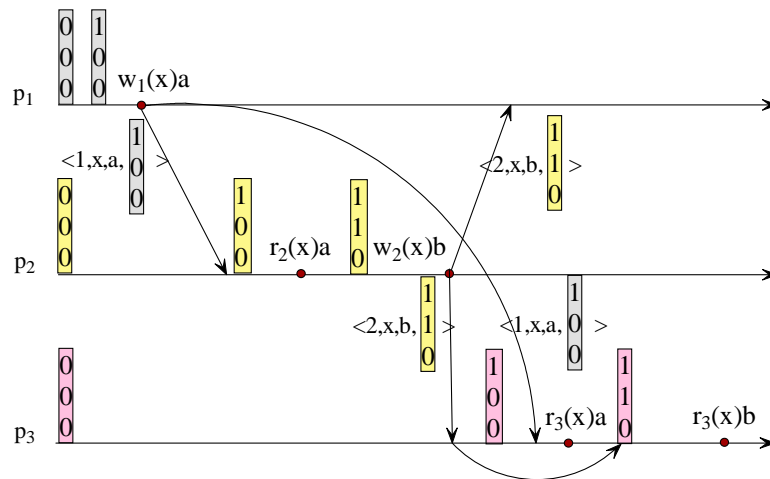
Successivamente spedisce il messaggio di aggiornamento a tutti gli altri processi.

Quando  $p_1$  e  $p_3$  ricevono il messaggio di  $p_2$ , prima di poter applicare il corrispondente update, devono assicurarsi di aver effettuato tutti gli aggiornamenti applicati da  $p_2$  alla sua copia locale della memoria fino all'istante in cui ha invocato  $w_2(x)b$ . In pratica,  $p_1$  e  $p_3$  devono aver applicato almeno le stesse scritture applicate da  $p_2$  fino all'invocazione di  $w_2(x)b$ , quindi anche  $w_1(x)a$ . In questo modo, quando un qualsiasi processo applicherà il valore  $b$  alla sua copia locale della memoria, il valore  $a$  sarà stato **sovrascritto** e quindi **nessuno potrà leggere prima  $a$  e poi  $b$** .

Si noti che in questo scenario i messaggi di aggiornamento arrivano ordinati.



## Scenario Di Funzionamento (2)



Protocollo Ahamed

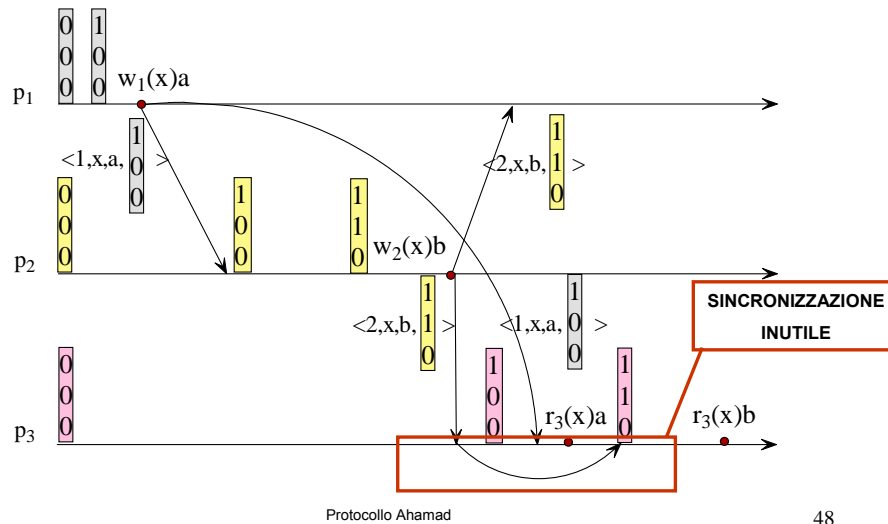
47

La computazione mostrata in figura genera la stessa **Execution History** di quella generata nella computazione di pag 46. Questa volta però i messaggi di aggiornamento non arrivano in maniera ordinata.

In particolare, quando il processo  $p_3$  riceve il messaggio di aggiornamento relativo a  $w_2(x)b$  non può applicarlo in quanto dall'analisi del vector clock si rende conto di aver perso una scrittura del processo  $p_1$ ,  $w_1(x)a$ , quindi bufferizza il messaggio e attende fino a quando non avrà applicato  $w_1(x)a$ . Una volta applicato l'aggiornamento corrispondente a  $w_1(x)a$ ,  $p_3$  potrà applicare anche  $w_2(x)b$ . Questa sincronizzazione assicura che la computazione genererà soltanto **Execution History causalmente consistenti**.



## Scenario Di Funzionamento (3)



La computazione mostrata in figura non dà origine alla stessa **Execution History** di pag 46,47 in quanto la history  $H$  è la stessa, ma variano le relazioni di ordinamento tra le operazioni. In particolare  $w_1(x)a$  e  $w_2(x)b$  sono **causalmente concorrenti**.

Si noti che il protocollo, anche in questo caso, impone che un qualsiasi processo nel sistema potrà leggere **prima  $x=a$  e poi  $x=b$** , mai il contrario. Questa situazione dipende dal fatto che il processo  $p_2$  prima di invocare  $w_2(x)b$  ha applicato l'aggiornamento corrispondente a  $w_1(x)a$ . Poiché il processo **potrebbe** aver letto il valore applicato,  $x=a$ , il protocollo assicura l'ordinamento tra le scritture a prescindere dall'effettiva esistenza o meno del legame (nel dubbio che il legame ci sia assicuro l'ordinamento). Si noti però, che il legame tra le scritture  $w_1(x)a$ ,  $w_2(x)b$  si forma ( $w_1(x)a \rightarrow_{co} w_2(x)b$ ) se  $p_2$  legge  $x=a$  altrimenti no ( $w_1(x)a \parallel_{co} w_2(x)b$ ). Quindi basterebbe tener traccia delle letture per sapere con certezza se il legame esiste. Per avere queste ulteriori informazioni, però, si deve complicare il protocollo e aumentare le strutture dati locali ad ogni processo.



## Limiti di Ahamad

- Protocollo non ottimale: presenza di sincronizzazioni inutili
- Riduce le run possibili
- Limita la concorrenza

Il protocollo di Ahamad non è ottimale perché come mostrato nello scenario di pag 48 impone sincronizzazioni non necessarie, limitando le possibili run generabili dal sistema e riducendo quindi la concorrenza.

I limiti del protocollo di Ahamad sono legati alle regole secondo le quali il sistema di vector clock viene aggiornato. Gli eventi in relazione ai quali il vector clock è aggiornato non sono collegati alla semantica del criterio di consistenza causale.



## Come Evitare Sincronizzazioni Inutili

- Strutture dati locali addizionali al processo  $p_i$ :
  - $\text{Apply}[1..n]$ : un array di interi inizializzato a tutti 0.  $\text{Apply}[j]$  è il numero di scritture invocate da  $p_j$  e applicate da  $p_i$ .
  - $\text{LastWriteOn}[1..m]$ : un array di vettori.  $\text{LastWriteOn}[h]$  è il  $\text{Write}_{\text{co}}$  relativo all'ultima scrittura che ha aggiornato la locazione  $x_h$  a  $p_i$ .

50

Si propone un protocollo in grado di superare i limiti del protocollo di Ahamad.

IDEA: usare un sistema di vector clock le cui regole di aggiornamento siano tali da tener traccia esclusivamente dei legami di causalità tra le varie scritture. Come nel caso di Ahamad ogni scrittura è associata ad un vector clock,  $w_k(x)v.\text{Write}_{\text{co}}$ .

In particolare:

1. Quando un processo  $p_i$  **scrive** incrementa di 1 la  $i$ -esima locazione del suo vector clock (**process order**)
2. Quando un processo  $p_i$  **legge** un valore  $v$  aggiorna il vector clock  $\text{Write}_{\text{co}}$  nel seguente modo:  $\forall j$

$\text{Write}_{\text{co}}[j] := \max(\text{Write}_{\text{co}}[j], w_k(x)v.\text{Write}_{\text{co}}[j])$  (**read-from order e transitività**)

Si noti che quando un processo legge deve conoscere il vector clock associato alla scrittura corrispondente al valore letto. In tal senso serve una struttura dati locali aggiuntiva che permetta di registrare il vector clock relativo all'ultima scrittura che ha aggiornato la data locazione. **LastWriteOn[1...m]** è un array di vector clock, di dimensioni pari al numero **m** di variabili condivise.



## Procedura di Scrittura Eseguita da $p_i$

```
WRITE( $x_h, v$ )  
1   $Write_{co}[i] := Write_{co}[i] + 1;$     % tracking  $\mapsto po_i$  %  
2  send [ $m(x_h, v, Write_{co})$ ] to  $\Pi - p_i$ ; % send event %  
3  apply( $v, x_h$ );                    % apply event %  
4   $Apply[i] := Apply[i] + 1;$   
5   $LastWriteOn[h] := Write_{co};$     % storing  $w_i(x_h)v.Write_{co}$  %
```

51

Quando il processo  $p_i$  scrive,  $w_i(x_h)v$ , incrementa sia la locazione  $i$ -esima del vector clock  $Write_{co}$  (process order) sia la locazione  $i$ -esima della struttura dati  $Apply$  (per tener traccia di tutte le scritture applicate alla memoria locale). Registra infine il  $Write_{co}$  associato a  $w_i(x_h)v$  (linea 5 dello pseudo-codice).



## Procedura di Lettura Eseguita da $p_i$

```
READ( $x_h$ )  
1  $\forall k \in [1..n], Write_{co}[k] := \max(Write_{co}[k], LastWriteOn[h].Write_{co}[k]);$   
2 return( $x_h$ );
```

- Prima di restituire il valore viene aggiornato il  $Write_{co}$  per tener traccia dei legami di read-from order.



## Thread di Sincronizzazione

- 1 Upon the arrival of  $\mathbf{m}(x_h, v, W_{co})$  from  $p_u$
- 2 **wait until**  $((\forall t \neq u \in W_{co}, W_{co}[t] \leq Apply[t]) \text{ and } (Apply[u] = W_{co}[u] - 1))$ ;
- 3 **apply** $(v, x_h)$ ;
- 4  $Apply[u] := Apply[u] + 1$ ;
- 5  $LastWriteOn[h] := W_{co}$ ;

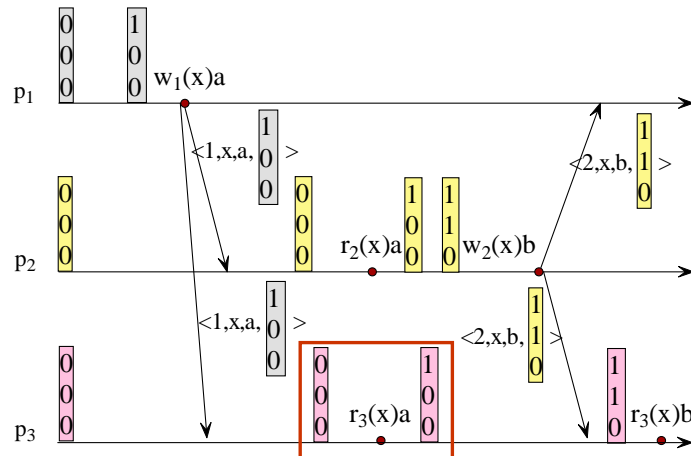
- 1) Evento di ricezione
- 4) Registrazione del numero di scritture applicate
- 5) Registrazione del  $Write_{co}$  relativo a  $w_u(x_h)v$

53

Si noti che nella **linea 2** del codice il confronto non viene fatto tra il vector clock locale  $Write_{co}$  e quello contenuto nel messaggio  $W_{co}$ , ma tra il vector clock nel messaggio  $W_{co}$  e una struttura dati aggiuntiva che tiene traccia di tutte le scritture applicate localmente ad un processo, **Apply[1..n]**.



# Scenario Di Funzionamento (1)



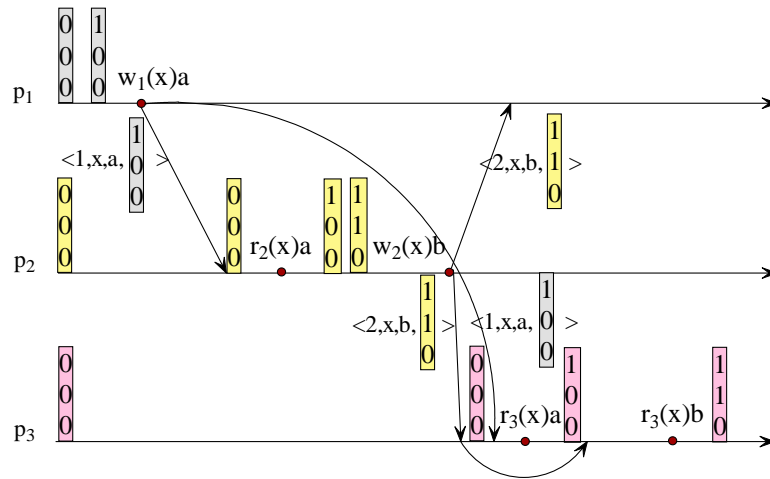
54

Notare che il vector clock non viene aggiornato quando si applica la scrittura alla memoria locale del processo, ma quando il processo stesso legge il valore aggiornato.

Nello scenario in figura, quando il processo  $p_3$  applica la scrittura  $w_1(x)a$  alla sua memoria aggiorna **Apply** ponendolo uguale a  $[1, 0, 0]$  mentre il vector clock **Write<sub>co</sub>** resta  $[0, 0, 0]$ . Successivamente quando legge  $r_3(x)a$  aggiorna **Write<sub>co</sub>** ponendolo uguale a  $[1, 0, 0]$  ( per ogni componente ha scelto il massimo tra le corrispondenti in **Write<sub>co</sub>** ( $[0, 0, 0]$ ) e  $r_3(x)a$ . **Write<sub>co</sub>** ( $[1, 0, 0]$ ) ).



## Scenario Di Funzionamento (2)

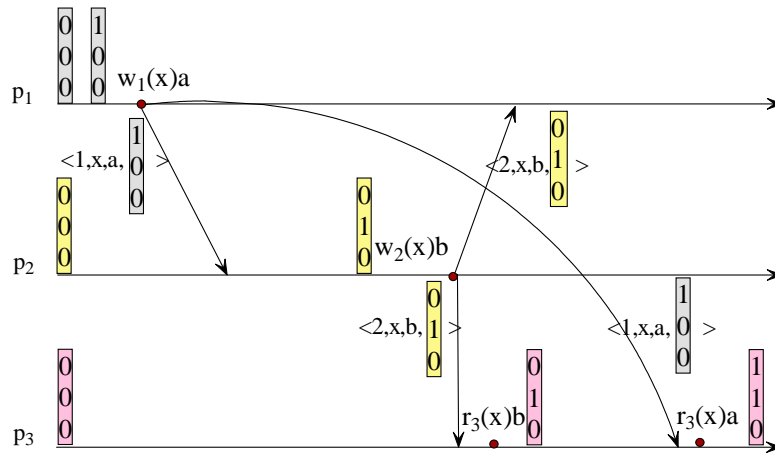


55

Il protocollo garantisce consistenza causale sincronizzando.



## Scenario Di Funzionamento (3)



56

Non avviene alcuna sincronizzazione, in quanto le due scritture sono causalmente concorrenti.

Il vector clock associato a  $w_2(x)b$  non porta alcuna informazione relativa a  $w_1(x)a$