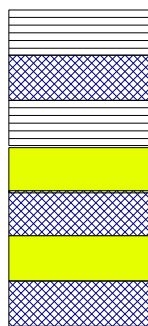


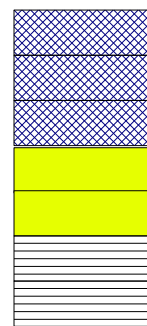
Transazioni atomiche concorrenti

- Per ora abbiamo trattato le transazioni senza preoccuparci del fatto che possano essere concorrenti, **la concorrenza => interleaving delle istruzioni ...**
- Come si combinano **concorrenza** ed **atomicità** (**atomicità a livello logico**, non di esecuzione sulla CPU)?
- **Idea**: garantire in qualche modo la **proprietà di serializzabilità**!
- **Serializzabilità di un insieme di transazioni**: proprietà per cui la loro esecuzione concorrente è equivalente alla loro esecuzione in una sequenza arbitraria

a ogni colore
corrisponde
una transazione
diversa: ogni
rettangolo è un'
operazione



concorrenza



esecuz. sequenziale

si potrebbero eseguire le
transazioni in ME, serializ-
zandole sul serio tramite
un semaforo mutex ma si
tratta di una soluzione
troppo rigida

Esempio

A	B
3	5

Siano T1 e T2 due transazioni concorrenti che utilizzano gli stessi dati condivisi, A e B, aventi valori iniziali 3 e 5

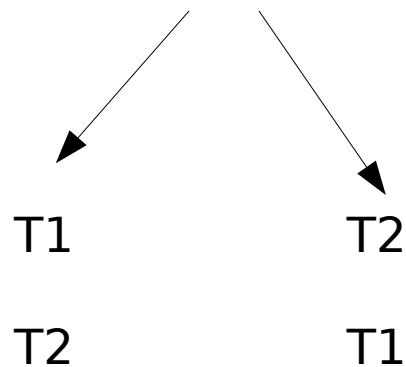
T1	T2
read(A)	read(A)
write(A)	write(A)
read(B)	read(B)
write(B)	write(B)

Sono **funzionalmente atomiche**: a livello logico le quattro operazioni costituiscono una funzionalità unica

Esempio

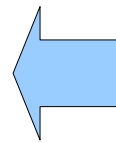
A	B
3	5

Gli **unici stati corretti** per il sistema sono quelli raggiungibili eseguendo T1 e T2 in modo sequenziale



Possibili alternative

T1	T2
read(A)	read(A)
write(A)	write(A)
read(B)	read(B)
write(B)	write(B)



Sono **funzionalmente atomiche**: a livello logico le quattro operazioni costituiscono una funzionalità unica

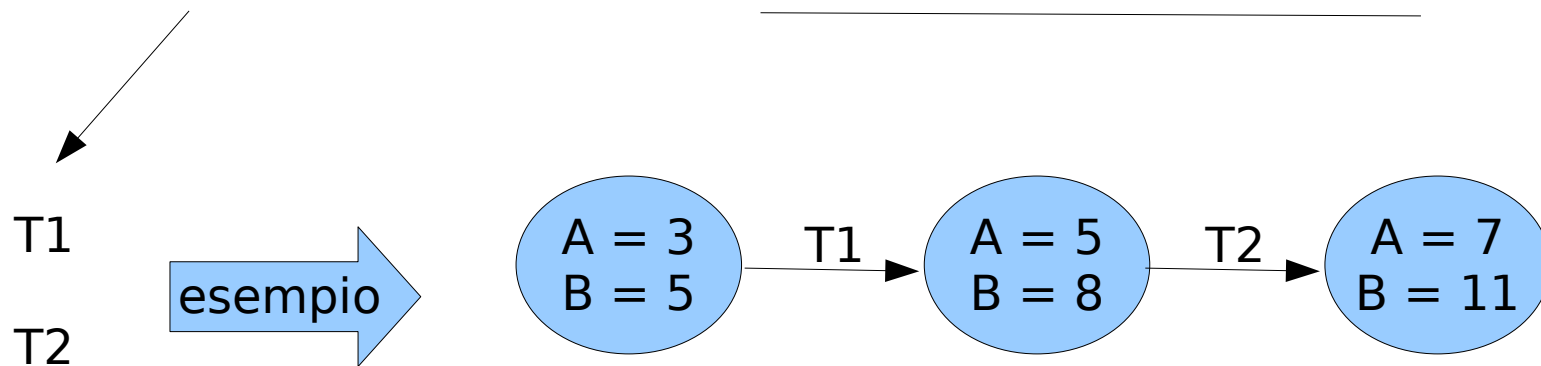
Esempio

A	B
3	5

Gli **unici stati corretti** per il sistema sono quelli raggiungibili eseguendo T1 e T2 in modo sequenziale

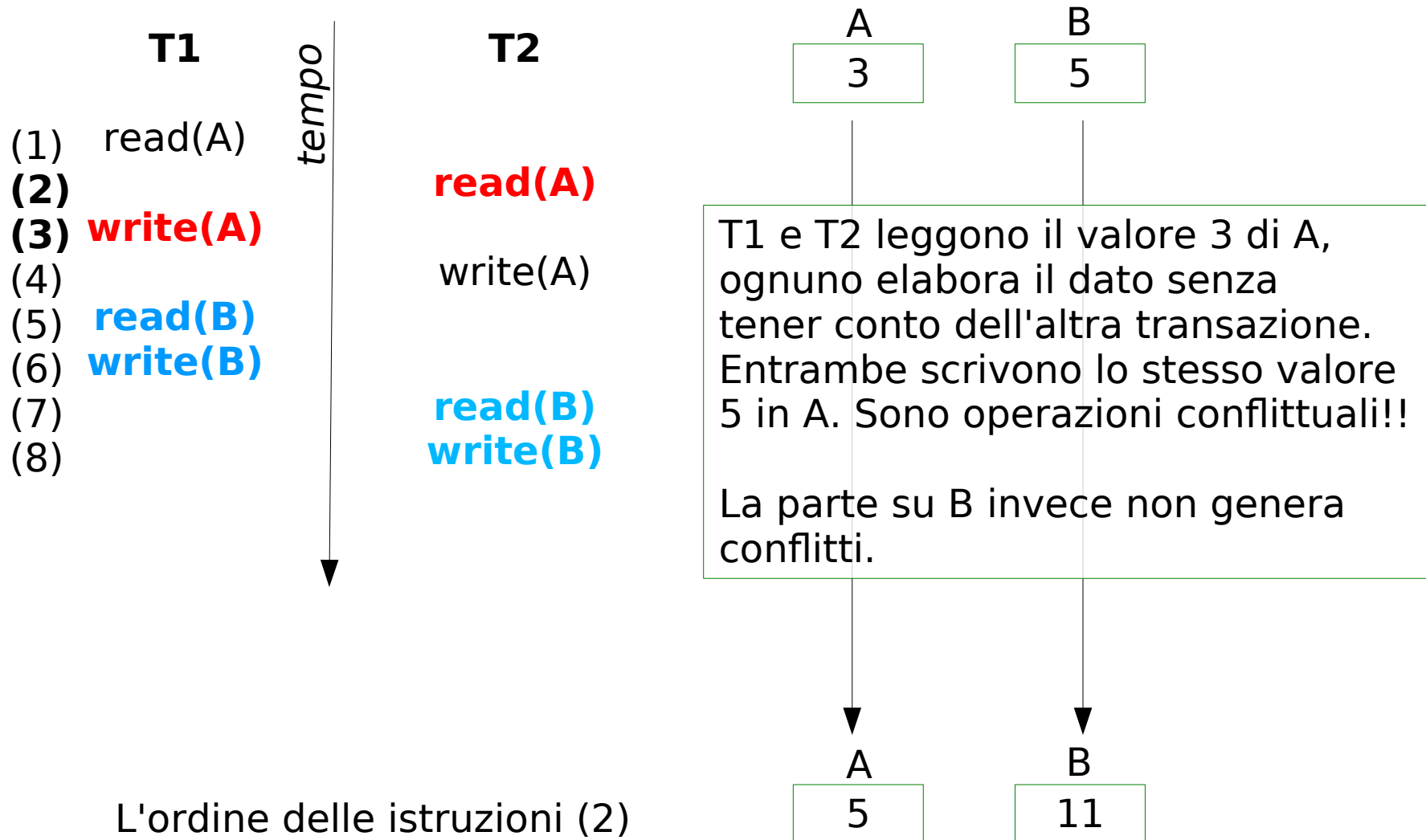
T1	T2
read(A)	read(A)
A=A+2	A=A+2
write(A)	write(A)
read(B)	read(B)
B=B+3	B=B+3
write(B)	write(B)

Esempio di elaborazione fatta dai processi che eseguono T1 e T2 su A e B



Problema: i sistemi ad alte prestazioni non possono permettersi di proibire l'esecuzione concorrente delle transazioni. Occorre consentire un po' di interleaving però in modo tale che lo stato raggiunto sia uno degli stati leciti (stesso risultato in tempo più rapidi)

Esempio



L'ordine delle istruzioni (2) e (3) non doveva essere invertito

Non è uno stato lecito!!!!

Esempio

CC_cl1 == 100
Prezzo == 50

READ CC_cl1
READ CC_vend
WRITE CC_cl1 CC_cl1 - 50
WRITE CC_vend CC_vend + 50

T1

CC_cl2 == 100
Prezzo == 30

READ CC_cl2
READ CC_vend
WRITE CC_cl2 CC_cl2 - 30
WRITE CC_vend CC_vend + 30

T2

T1 e T2 sono due esecuzioni dello stesso codice, catturano l'acquisto di oggetti diversi da parte di due clienti, effettuato presso lo stesso venditore, il cui CC vale inizialmente 100.

Senza interleaving si raggiunge lo stato finale (corretto) in cui CC_vend vale 180. E se consento l'inteleaving? Consideriamo un esempio ...

Esempio

CC_cl1 == 100
Prezzo == 50

READ CC_cl1

READ CC_vend

WRITE CC_cl1 CC_cl1 - 50
WRITE CC_vend CC_vend + 50

CC_cl2 == 100
Prezzo == 30

READ CC_cl2
READ CC_vend

WRITE CC_cl2 CC_cl2 - 30
WRITE CC_vend CC_vend + 30

▼ tempo

Esempio

CC_cl1 == 100
Prezzo == 50

READ CC_cl1

Ha letto il valore 100

READ CC_vend

WRITE CC_cl1 CC_cl1 - 50
WRITE CC_vend CC_vend + 50

Calcola l'espression
a partire dal valore 100

CC_cl2 == 100
Prezzo == 30

READ CC_cl2
READ CC_vend

Ha letto il valore 100

WRITE CC_cl2 CC_cl2 - 30
WRITE CC_vend CC_vend + 30

Calcola l'espression
a partire dal valore 100

**ERRORE! Valore finale di
CC_vend pari a 150 !**

tempo

Serializzabilità

- In gnerale dati N elementi, possiamo costruire $N!$ (N fattoriale) sequenze diverse
- Quindi date N transazioni possiamo quindi trovare $N!$ sequenze di esecuzione seriale
- Vogliamo consentire un po' di concorrenza per aumentare l'efficienza complessiva dell'esecuzione però ci interessano delle sequenze di esecuzione non seriale che sono equivalenti (dal punto di vista degli effetti) a quelle seriali, che ci danno la garanzia di produrre risultati consistenti

Serializzabilità

- Introduciamo un concetto nuovo: **operazioni conflittuali**:

date due **transazioni** T1 e T2 e le due **operazioni** O1 (appartenente a T1) e O2 (appartenente a T2), se O1 e O2 appaiono in successione, accedono agli stessi dati e almeno una delle due operazioni è una **write**, allora O1 e O2 sono **operazioni conflittuali**

- **NB:**

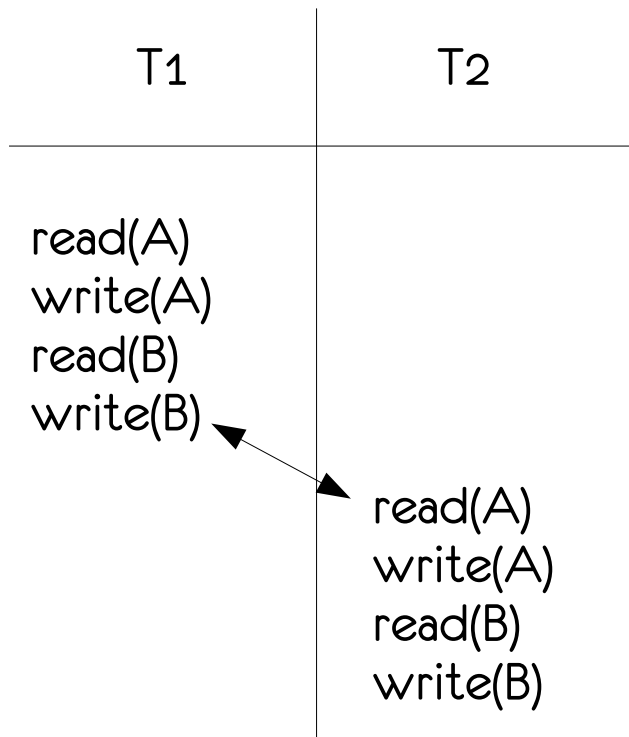
- 1) l'ordine di esecuzione di due operazioni conflittuali incide sul risultato
- 2) due operazioni non conflittuali possono essere scambiate

esempio

T1	T2
read(A) write(A) read(B) write(B)	read(A) write(A) read(B) write(B)

esecuzione seriale di
T1 e T2

esempio

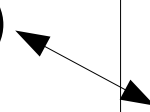


esecuzione seriale di
T1 e T2

Proviamo a introdurre un po' di interleaving
write(B) e read(A) sono conflittuali?

esempio

T1	T2
read(A) write(A) read(B) write(B)	read(A) write(A) read(B) write(B)

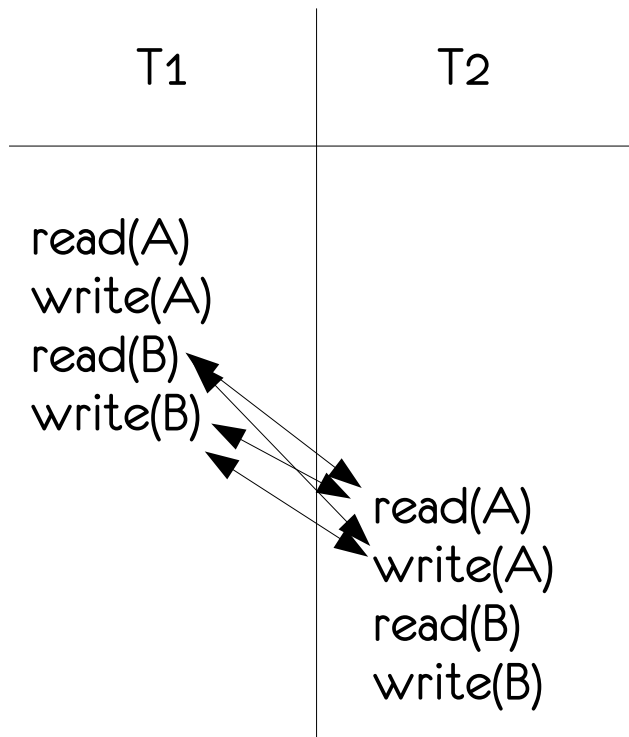


esecuzione seriale di
T1 e T2

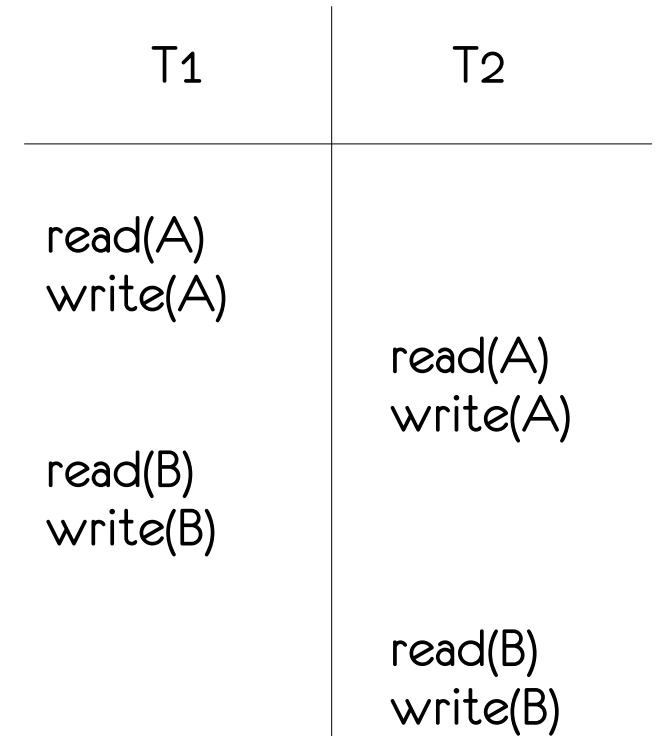
T1	T2
read(A) write(A) read(B) write(B)	read(A) write(A) read(B) write(B)

esecuzione concorrente di
T1 e T2 equivalente a
quella seriale

esempio



esecuzione seriale di
T1 e T2



esecuzione concorrente di
T1 e T2 equivalente a
quella seriale

Dopo qualche iterazione ...

esempio

T1	T2
read(A) write(A) read(B) write(B)	read(A) write(A) read(B) write(B)

conflitto!

Sono operazioni
conflittuali!!!

esecuzione seriale di
T1 e T2

T1	T2
read(A) write(A)	read(A) write(A)
read(B) write(B)	
	read(B) write(B)

esecuzione concorrente di
T1 e T2 equivalente a
quella seriale

Protocollo di gestione dei lock

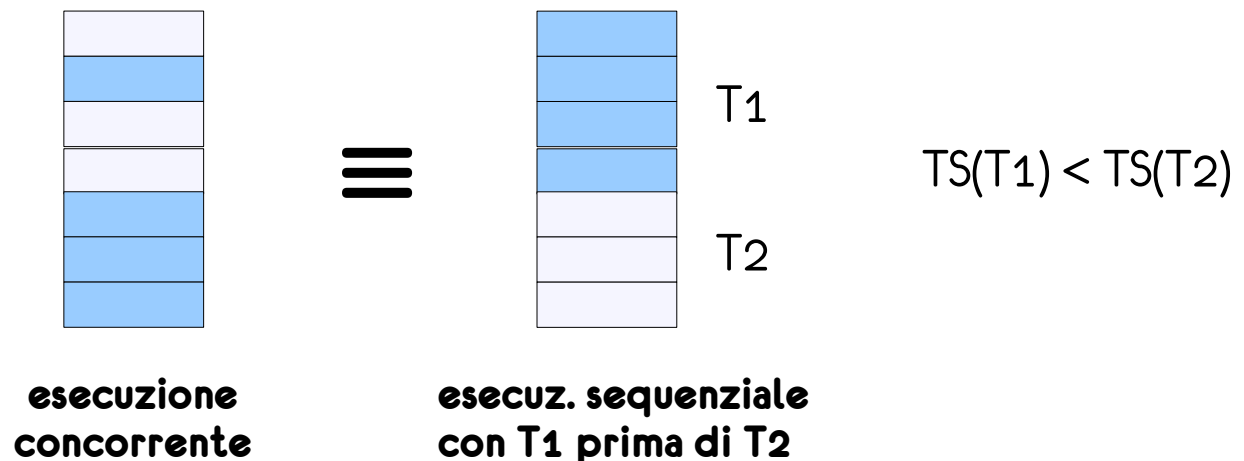
- Per garantire la serializzabilità si può usare un meccanismo di **lock** in cui:
 - a ogni dato soggetto a transazione si associa un lock
 - il lock di un dato può essere:
 - **S** (indica la possibilità di condivisione): la transazione può leggere il dato ma non scriverlo
 - **X** (indica esclusività): la transazione può sia leggere che modificare un certo dato
- una transazione che intenda usare un certo dato deve richiederne il lock appropriato, ed eventualmente attendere che un'altra transazione lo rilasci
- <1> protocollo di gestione dei lock a due fasi
- <2> protocolli di ordinamento dei timestamp

Gestione dei lock a due fasi

- una transazione si può trovare in fase di crescita oppure di riduzione
- inizialmente tutte le transazioni sono in fase di crescita
- una transazione in fase di crescita può ottenere nuovi lock ma non ne rilascia mai nessuno (acquisisce tutte le risorse necessarie)
- una transazione in fase di riduzione può rilasciare lock ma non può richiederne di nuovi (rilascia via via quelle che non le servono più)
- **Commenti:**
 - si può avere deadlock
 - non tutte le sequenze di esecuzione lecite delle operazioni possono essere trovate

Protocolli basati su timestamp

- Cos'è un **timestamp**? È una rappresentazione univoca di un istante temporale, è anche detta marker temporale
- Il sistema assegna a ogni transazione un timestamp prima che questa inizi ad eseguire. Indichiamo il timestamp associato a T con $TS(T)$
- **Idea che vogliamo realizzare**: se due transazioni hanno timestamp $TS(T1) < TS(T2)$ allora il sistema deve garantire che la sequenza di esecuzione delle istruzioni di T1 e T2 sia equivalente all'esecuzione sequenziale in cui T1 viene eseguita prima di T2



Esempio di protocollo

- **Premessa**
- in questo protocollo ogni dato D soggetto a transazione ha due timestamp:
 - $R(D)$: denota il valore più alto di timestamp associato a una transazione che ha letto il dato D
 - $W(D)$: denota il valore più alto di timestamp associato a una transazione che ha eseguito un'operazione di scrittura su D
- questi due valori cambiano nel tempo a seconda degli accessi effettuati a D
- **Regole che definiscono il protocollo**
 - . . .

Regole che definiscono il protocollo

- **<1>** se una transazione **T** desidera **leggere D**:
 - **<1.a>** se $TS(T) < W(D)$: ho una transazione vecchia che cerca di leggere un valore sovrascritto da una più recente -> **azione**: si annulla la lettura richiesta e si esegue una sequenza di undo per annullare T
 - **<1.b>** se $TS(T) > W(D)$: ok -> **azione**: si effettua la lettura e si aggiorna $R(D)$ assegnando $\max(R(D) , TS(T))$

Regole che definiscono il protocollo

- **<2>** se una transazione T desidera **scrivere D** :
 - **<2.a>** se $TS(T) < R(D)$: problema, si vuole fare un aggiornamento che avrebbe dovuto precedere l'ultima lettura -> **azione**: si annulla la write e si effettuano degli undo per annullare T
 - **<2.b>** se $TS(T) < W(D)$: problema, si vuole fare un aggiornamento obsoleto -> **azione**: si annulla la write e si effettuano degli undo per annullare T
 - **<2.c>** si esegue la write e si aggiorna $W(D)$ a $TS(T)$

NB: a una transazione fallita e disfatta viene assegnato un nuovo timestamp, poi la si riavvia

Esempio

Partenza: abbiamo due transazioni, **T1 con $TS(T1) = 1$ e T2 con $TS(T2) = 2$**
Le due transazioni sono descritte qui sotto:

T1	T2
read(A) read(B) write(A) read(B)	write(B) read(A) write(A)

Memento:

anche se non eseguite in modo atomico, vogliamo lo stesso risultato ottenuto eseguendole in modo atomico!!!

Proviamo a simularne un'esecuzione (immaginando un certo interleaving) e vediamo come si comporta l'algoritmo basato su timestamp

Vogliamo che l'effetto finale sia identico a quello che si avrebbe in una esecuzione sequenziale di T1 e T2, sia esse T1 e poi T2 oppure T2 e poi T1

Simulazione 1/3

T1	T2
read(A)	write(B)
read(B)	read(A)
write(A)	write(A)
read(B)	

Dati da gestire con le transazioni:

A inizialmente $R(A) = 0$ $W(A) = 0$
B inizialmente $R(B) = 0$ $W(B) = 0$

inizia T1

read(A): è eseguibile? Applichiamo l'algoritmo

$TS(T1) < W(A)?$ $1 < 0?$ no!
eseguo read(A) e pongo $R(A) = \max(R(A), TS(T1)) = 1$

read(B): è eseguibile?

$TS(T1) < W(A)?$ $1 < 0?$ no!
eseguo read(B) e pongo $R(B) = 1$

Supponiamo che ora subentri T2 ...

Simulazione 2/3

T1	T2
read(A)	write(B)
read(B)	read(A)
write(A)	write(A)
read(B)	

Dati da gestire con le transazioni:

A attualmente $R(A)=1$ $W(A)=0$
B attualmente $R(B)=0$ $W(B)=0$

inizia T2

write(B): è eseguibile?

$TS(T2) < R(B)?$ $2 < 0?$ no!

$TS(T2) < W(B)?$ $2 < 0?$ no!

eseguo write(B) e pongo $W(B) = TS(T2) = 2$

read(A): è eseguibile?

$TS(T2) < W(A)?$ $2 < 0?$ no!

eseguo read(A) e pongo $R(A) = 2$

Simulazione 3/3

T1	T2
read(A)	write(B)
read(B)	read(A)
write(A)	write(A)
read(B)	

Dati da gestire con le transazioni:

A attualmente $R(A)=2$ $W(A)=0$
B attualmente $R(B)=0$ $W(B)=2$

continua T1

write(A): è eseguibile?

$TS(T1) < R(A)?$ $1 < 2?$ sì!

sto cercando di assegnare ad A un valore ormai obsoleto

<1> non eseguo write(A)

<2> rollback di T1

<3> assegno a T1 un nuovo TS (es. 3) e la riavvio

Osservazioni

- L'algoritmo non impone un particolare ordinamento (considerato corretto) fra le transazioni
- L'unico scopo è far sì che tutte le volte che una transazione legge un dato questo abbia o il valore che aveva all'inizio della transazione stessa oppure sia stato modificato dalla transazione stessa
- In questo consiste la garanzia di atomicità funzionale, diversa dall'atomicità di esecuzione:
 - **atomicità funzionale:** interleaving consentito
 - **atomicità di esecuzione:** interleaving disabilitato
- **dove si ha il guadagno?** Transazioni che non interferiscono l'una con l'altra perché o usano dati diversi o l'una non modifica i dati usati dall'altra possono essere eseguite in modo concorrente
- **l'algoritmo non è preventivo:** identifica situazioni problematiche e le aggiusta