Problemi classici di sincronizzazione

- Produttori – consumatori con buffer limitato
- Lettori – scrittori: dei processi lettori e scrittori usano una stessa area di memoria. Gli scrittori modificano la risorsa, quindi accedono in ME con tutti. I lettori non modificano la risorsa, quindi possono accedere alla memoria condivisa in parallelo.
  - Si utilizzano:
    - scrittura: semaforo di ME
    - una variabile intera condivisa numlettori che conta quanti processi stanno leggendo in questo momento
    - mutex: semaforo di ME per l'accesso a numlettori
- Cinque filosofi

lettori - scrittori

Lettore

```
forever {
        P(mutex);
        numlettori++;
        if (numlettori == 1)
           P(scrittori);
        V(mutex);

        <legge>

        P(mutex);
        numlettori--;
        if (numlettori == 0)
           V(scrittura);
        V(mutex);
}
```

Scrittore

```
forever {

        P(scrittura);

        <scrive>

        V(scrittura);

}
```

Inizialmente mutex vale 1, scrittura vale 1 e numlettori vale 0

scrittore

```
Scrittore

forever {
    P(scrittura);
    <scrive>
    V(scrittura);
}
```

uno scrittore accede all'area condivisa in ME con chiunque altro (lettori o scrittori) perché modifica la risorsa. Scrittura vale inizialmente 1

lettore

```
Lettore

forever {
     [A]  P(mutex);
          numlettori++;
          if (numlettori == 1)
             P(scrittori);
          V(mutex);

          <legge>

     [B]  P(mutex);
          numlettori--;
          if (numlettori == 0)
             V(scrittura);
          V(mutex);
}
```

un lettore usa due classi di sezione critica: una relativa all'area di memoria da cui legge e una relativa alla variabile condivisa numlettori. Mutex controlla l'accesso a quest'ultima

[A] e [B] parentesizzano la sezione critica relativa all'area da cui si legge. Implementano un controllo per cui un lettore può accedervi a patto che nessuno scrittore stia operando.

[A]:
se numlettori > 1 allora qualcun altro ha già fatto in modo tale che nessuno scrittore sia attivo. Il lettore può procedere nella lettura.

se numlettori == 1 sono l'unico lettore, tramite il semaforo scrittura controllo ed eventualmente attendo che non ci siano scrittori attivi. Poiché uso una P, quando riesco a passare blocco l'accesso ad eventuali scrittori

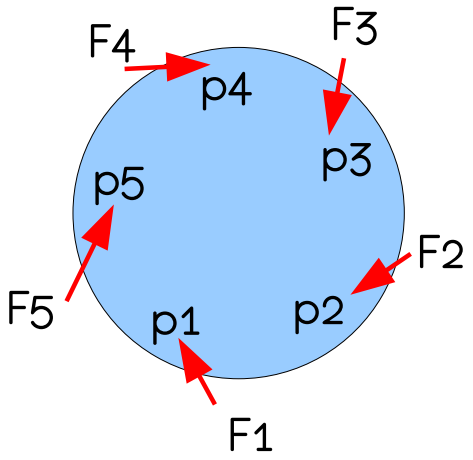
[B] decrementa numlettori ed eventualmente incrementa il semaforo scrittura

Problemi classici di sincronizzazione

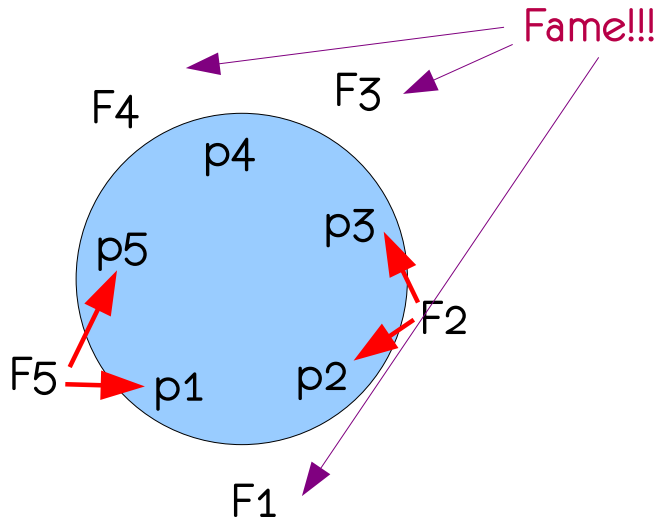
- Produttori – consumatori con buffer limitato
- Lettori – scrittori
- Cinque filosofi: 5 filosofi passano il tempo seduti intorno a un tavolo pensando e mangiando a fasi alterne. Per mangiare un filosofo ha bisogno di due posate ma ci sono solo cinque posate in tutto

è un problema interessante perché rappresenta un'ampia categoria di situazioni pratiche in cui diversi processi hanno bisogno di più risorse e bisogna evitare situazioni di deadlock e di starvation (es. quando riesco ad avere solo una parte delle risorse necessarie)

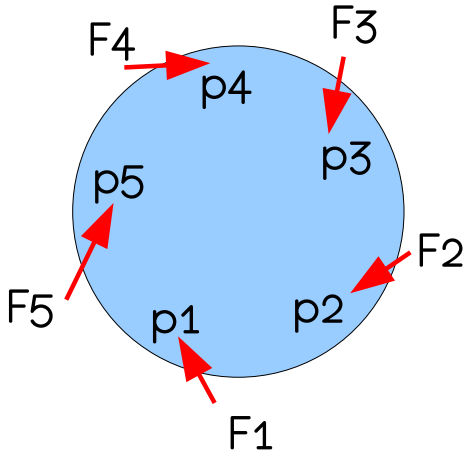
deadlock e altri guai



Deadlock: tutti hanno preso la posata di sinistra ma nessuno può prendere quella di destra



Starvation: due filosofi si accaparrano sempre le posate necessarie impedendo agli altri di mangiare



Livelock: tutti prendono la posata di sinistra ma nessuno può prendere quella di destra, quindi tutti rilasciano la posata di sinistra, poi ricominciano da capo. I processi non sono propriamente bloccati ma non c'è progresso.

una soluzione

Filosofo F_i

```
forever {
      P(posata[i]);
      P(posata[i+1]%5);

      <mangia>

      V(posata[i]);
      V(posata[i+1]%5);

      <pensa>
}
```

si usano 5 semafori, uno per ciascuna posata; tutti richiedono la posata di sinistra e poi quella di destra.

Deadlock? Sì

Starvation? Sì

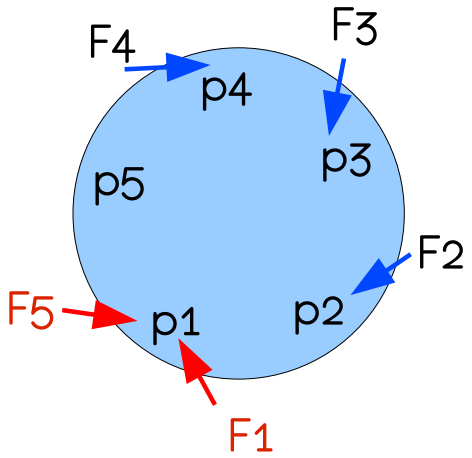
Prevenire il deadlock

Dijkstra propone una soluzione che consente di evitare il deadlock. A questo fine occorre rompere la simmetria nell'accesso alle posate. Introduzione di una semplice regola:

ogni filosofo deve prendere per prima la posata con indice minore

F1 prenderà quindi p1 prima di p2, ..., F4 prenderà p4 prima di p5.

Apparentemente al contrario F5 prenderà prima p1 e poi p5!!!



F1 ed F5 concorrono all'uso di p1:

se vince F1, F5 aspetterà che p1 si liberi prima di richiedere p5, lasciando a F4 la possibilità di usarla

se vince F5, F1 aspetterà che p1 si liberi, lasciando a F5 la possibilità di mangiare richiedendo anche p5

NB: la starvation persiste

Chandy-Misra

Chandy-Misra propongono una soluzione “equa”, nel senso che quando due processi sono in competizione l'algoritmo non favorisce sempre lo stesso (causando starvation). Viene introdotto un concetto di precedenza fra processi, che però cambia nel tempo, è dinamica. Il dinamismo è ottenuto associando un concetto di stato alla risorsa.

1. Ogni forchetta può essere “sporca” o “pulita”, inizialmente sono tutte “sporche” e ogni filosofo (quello con id più basso fra i due vicini) ne ha una..

2. Quando un filosofo vuole mangiare invia ai suoi vicini dei messaggi per ottenere le forchette che gli mancano.

3. Quando un filosofo, che ha una forchetta, riceve una richiesta: se sta pensando, la cede altrimenti se la forchetta è pulita ne mantiene il possesso, se è sporca la cede. Quando passa una forchetta ne pone lo stato a “pulita”.

4. Dopo aver mangiato, tutte le forchette di un filosofo diventano “sporche”. Se risultano richieste pendenti per qualche forchetta, il filosofo la pulisce e la passa.

Uso errato dei semafori

- È molto facile introdurre errori involontari in programmi che fanno uso di semafori. consideriamo la semplice mutua esclusione:
  - V(mutex) <sez. cr.> P(mutex)
  - P(mutex) <sez. cr.> P(mutex)
  - <sez. critica>
- sono tutti esempi di programmi sbagliati. Basta un solo processo sbagliato per creare deadlock!!!
- una soluzione è incapsulare risorse di basso livello, come i semafori, in tipi di dati astratti, come i **monitor**, offerti poi come nuovi costrutti del linguaggio

Monitor

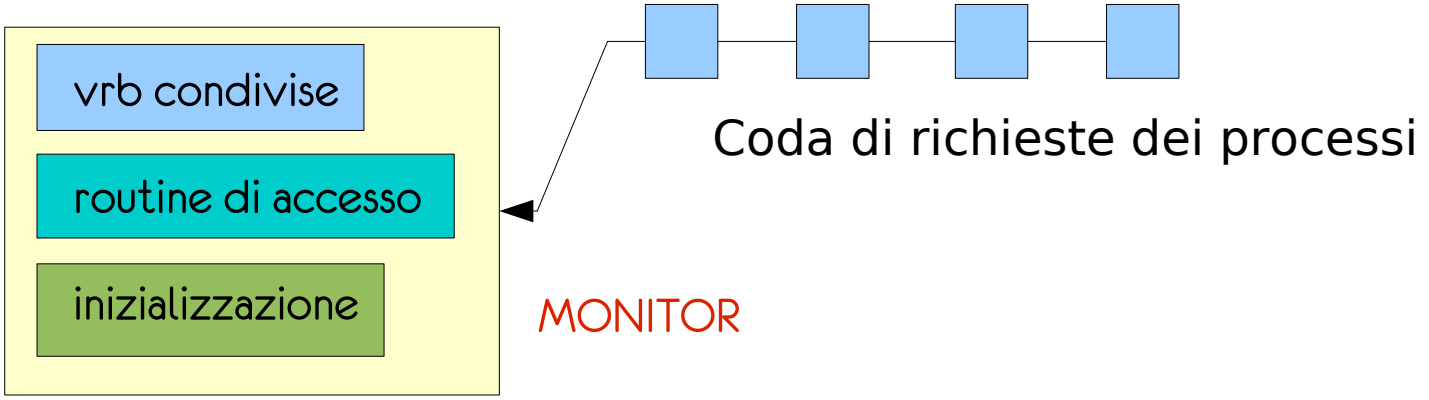
- **Monitor** (Dijkstra, Brinch Hansen) è un costrutto di sincronizzazione contentente i dati e le operazioni necessarie per allocare una risorsa condivisa usabile in modo seriale
- È un **Abstract Data Type**:
  incapsula dei dati mettendo a disposizione le operazioni necessarie per manipolarli
- Le variabili di un monitor sono **condivise** dai processi che usano quel monitor
- Un solo processo per volta può essere attivo all'interno di un certo monitor (**Mutua Esclusione**)

Monitor

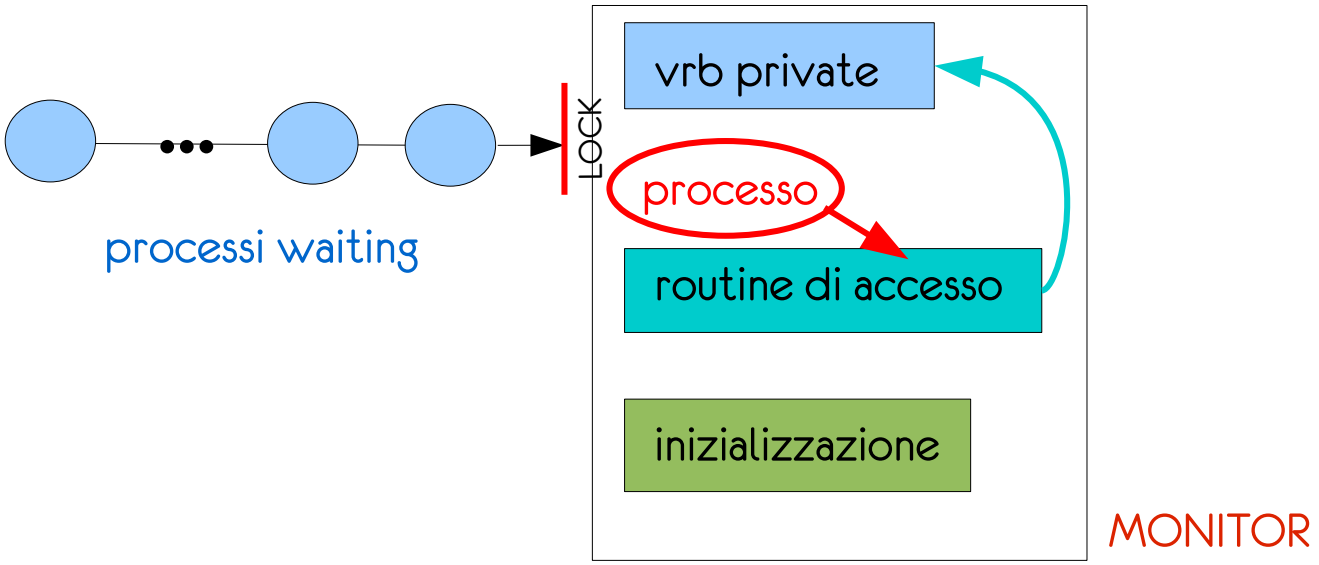
- Brinch-Hansen: “You can imagine the (monitor) calls as a **queue of messages** being served one at a time. The monitor will receive a message and try to carry out the request as defined by the procedure and its input parameters. If the request can immediately be granted the monitor will return parameters . . . and allow the calling process to continue. However, if the request cannot be granted, the monitor will prevent the calling process from continuing, and enter a reference to this transaction in a queue local to itself. This enables the monitor [...] to inspect the queue and decide which interaction should be completed now. From the point of view of a process a monitor call will look like a procedure call. The calling process will be delayed until the monitor consults its request. The monitor then has a set of scheduling queues which are completely local to it, and therefore protected against user processes.”
- Fonte: P. Brinch Hansen, Monitors and Concurrent Pascal: A personal history. 2nd ACM Conference on the History of Programming Languages, Cambridge, MA, April 1993. In c 1993, Association for Computing Machinery, Inc.

Monitor

- Per accedere alle variabili condivise un processo deve eseguire una routine di accesso al monitor (possono essercene diverse)
- un solo processo per volta può essere attivo all'interno di un certo monitor (Mutua Esclusione)
- es. i monitor sono usati da Java per realizzare la ME dei thread (prog. III)



Monitor



un processo che riesce ad eseguire una routine di accesso al monitor acquisisce un **lock** sul monitor. Tutti gli altri processi che cercano di eseguire routine di accesso vengono sospesi in una coda di attesa esterna al monitor. Quando il lock viene rilasciato uno dei processi waiting viene riattivato

Monitor

- oltre che la ME, i monitor consentono anche di effettuare alcuni tipi di **sincronizzazione fra i processi**
- es. monitor per produttori-consumatori: un consumatore che ha avuto accesso al monitor si accorge che non c'è nulla da consumare ... non c'è un modo per far sì che i consumatori possano entrare solo se è il caso?
- vengono introdotte variabili di tipo **condition**, su cui si possono eseguire solo le operazioni:
  - **wait(x)**: l'esecutore viene sospeso se la condition x è falsa
  - **signal(x)**: se qualche processo è sospeso sulla condition x, uno viene scelto e risvegliato, altrimenti non ha effetto
  - NB: signal è diversa dalla V dei semafori: la V incrementa sempre il valore del semaforo, ha sempre un effetto!

Monitor: esempio

monitor per un allocatore di risorse

```
boolean inUso = false;

condition disponibile;

monitorEntry void prendiRisorsa() {
    if (inUso) wait(disponibile);
    inUso = true;
}

monitorEntry void rilasciaRisorsa() {
    inUso = false;
    signal(disponibile);
}
```

se una qualche risorsa di interesse risulta usata da qualcun altro, allora mi sospendo sulla condition "disponibile". Quando mi risveglio continuo dalla linea di codice successiva: setto a true inUso.

Memento: il monitor è eseguito in ME quindi tutte le monitorEntry sono atomiche, a meno di sospensioni volontarie

quando rilascio una risorsa, faccio una notifica sulla condition "disponibile". Se qualcuno era in attesa si risveglia, altrimenti la notifica viene dimenticata

Signal e M.E.

Quando un processo (thread) esegue signal rischiamo di avere due processi (thread) attivi nel monitor!! Quello che ha eseguito signal e quello risvegliato

SOLUZIONI

Segnalare e Attendere

- P attende
- Q riprende ed esegue

Segnalare e Proseguire

- P continua
- Q aspetta che P finisca

5 filosofi realizzati coi monitor 1/2

```
Monitor 5filosofi {

    enum { pensa, affamato, mangia } stato[5];
    condition aspetta[5];

    Prende (int i) {
        stato[i] = affamato;
        verifica(i);
        if (stato[i] != mangia) aspetta[i].wait();
    }

    Posa (int i) {
        stato[i] = pensa;
        verifica((i+4) % 5);
        verifica((i+1) % 5);
    }

    ...
```

Può essere “mangia” solo se i due vicini non sono settati a “mangia”

Consente ai filosofi di sospendersi quando non possono mangiare

5 filosofi realizzati coi monitor 2/2

```
    ...

    Verifica (int i) {
       if (stato[(i+4)%5] != mangia) &&
           stato[i] == affamato      &&
           (stato[(i+1)%5] != mangia))
       {
          stato[i] = mangia;
          aspetta[i].signal();
       }
    }

    Codice di inizializzazione {
       for (int i=0; i<5; i++)
           stato[i] = pensa;
    }

}
```

M.E. e transazioni

- In talune circostanze applicative (e.g. basi di dati, contabilizzazione) il concetto di esecuzione in ME non è sufficiente. Si vorrebbe realizzare un'idea di atomicità più forte, per cui o si riesce ad eseguire un intero insieme di istruzioni con successo oppure non se ne deve eseguire nessuna. Devono essere eseguite come una sola operazione non interrompibile.
- Esempio:
  - acquisisci stampante,
  - contabilizza pagine da stampare,
  - stampa,
  - rilascia stampante

e se la carta finisce prima che io abbia potuto stampare tutte le pagine desiderate?

Mi verranno fatte pagare anche quelle non stampate?

Cosa me ne faccio delle sole prime 2 (per es.) pagine?

in questo caso vorrei poter effettuare il rollback delle istruzioni eseguite, disfarne gli effetti, tornare allo stato precedente: *cancella pagine stampate, sottrai loro numero da totale*

Es. journaling del file system

Esempio: transazione commerciale

CC_cl == 100 // conto corrente del cliente
CC_vend == 100 // c.c. del venditore
Prezzo == 50 // prezzo oggetto acquistato

READ CC_cl

READ CC_vend

WRITE CC_cl CC_cl - 50

WRITE CC_vend CC_vend + 50

Esempio: transazione commerciale

CC_cl == 100 // conto corrente del cliente
CC_vend == 100 // c.c. del venditore
Prezzo == 50 // prezzo oggetto acquistato

READ CC_cl

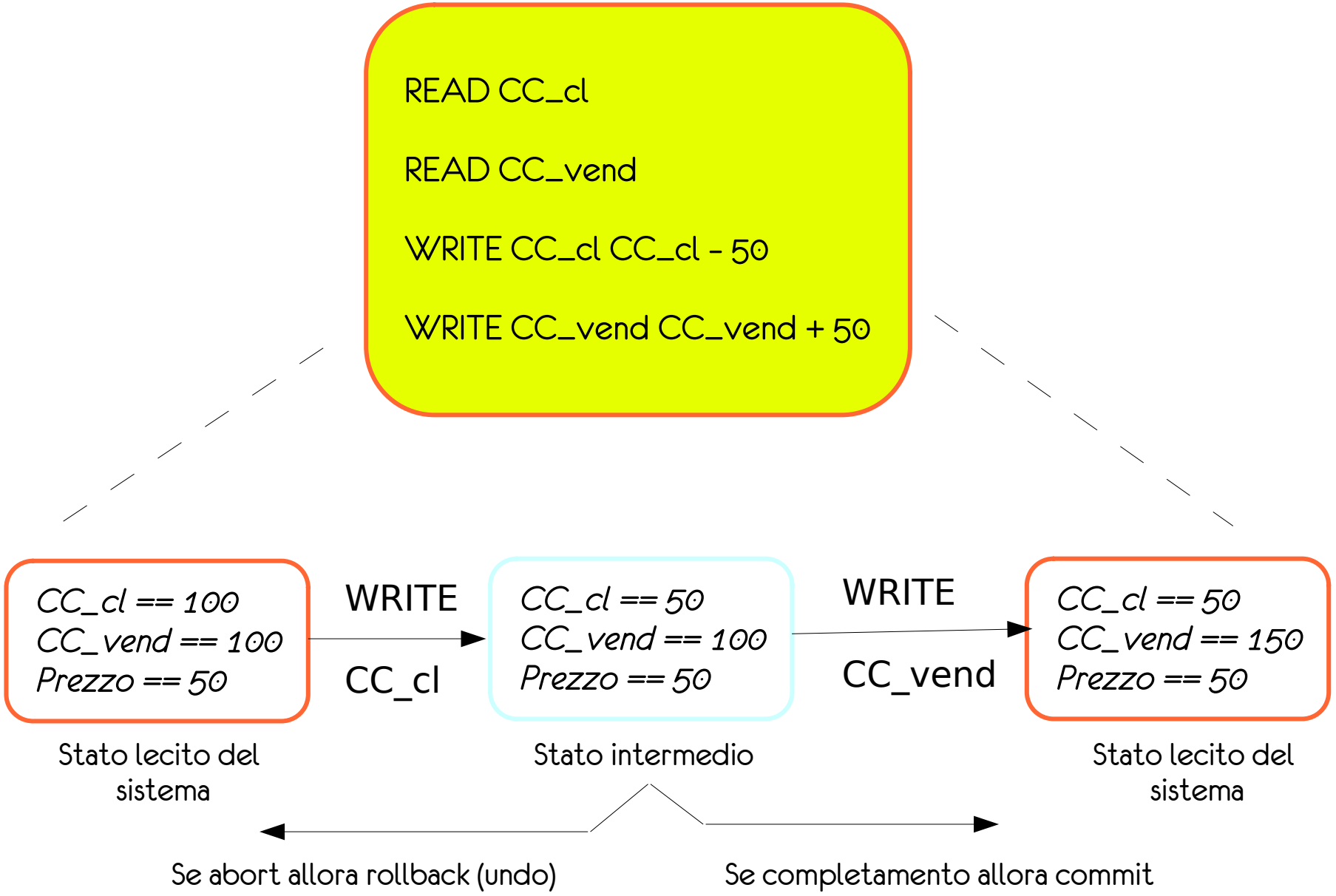
READ CC_vend

WRITE CC_cl CC_cl - 50

WRITE CC_vend CC_vend + 50

Se l'esecuzione si interrompesse a questo punto avremmo uno stato inconsistente: il CC del cliente è stato decrementato ma la cifra non è stata aggiunta al CC del venditore

Esempio: transazione commerciale



Transazione

- Transazione concetto che indica un insieme di istruzioni che esegue una singola funzione logica
- Ovvero una sequenza di **read** e **write** che si conclude con un **commit** o con un **abort**:
  - **commit**: la transazione si è conclusa con successo
  - **abort**: la transazione è fallita. Potrebbe avere alterato dei dati! Bisogna disfarne gli effetti (*rollback* o *compensazione*)

Transazione

- Come si fa a disfare una transazione abortita?
- Ovvero come si fa a garantire l'atomicità di una transazione?
- Dipende dai dispositivi di memoria usati per mantenere i dati elaborati dalla medesima:
  - **memorie volatili** (es. cache e registri): staccando la corrente si cancellano
  - **non volatili** (es. dischi, EEPROM): sono persistenti ma non danno garanzie di eternità
  - **stabili**: sono supporti di memorizzazione a cui si aggiungono politiche e strumenti di duplicazione che rendono “eterni” i dati memorizzati

Memoria volatile

- Vedremo solo il caso di transazioni abortite per cancellazione di memoria volatile (es. crash di sistema)
- Per ripristinare uno stato precedente occorre tener traccia delle operazioni eseguite => memorizzare in un file (mantenuto in memoria stabile e detto **logfile** o **log**) cosa viene fatto
- **Contenuto del log**:
  - per ogni transazione T si registra il suo inizio:

    <T, start>
  - si registra una sequenza di tuple, ciascuna relativa ad una operazione di write prossima da eseguire (write-ahead logging), siffatte:

    <ID transazione, ID dato modificato, valore precedente, nuovo valore>
  - si registra il successo della transazione T:

    <T, commit>

Ripristino/abort

- A seguito di un crash di sistema, il SO controlla il log e per ogni transazione T registrata nel log:
  - se a <T, start> non corrisponde un <T, commit> il SO esegue l'operazione undo(T) che ripristina tutti i valori modificati dalla transazione eseguita in modo parziale
  - se a <T, start> corrisponde <T, commit> il SO verifica che le modifiche registrate siano state effettivamente eseguite. È possibile che al crash alcune modifiche registrate nel log non fossero ancora state apportate ai dati veri e propri, in questo caso il SO esegue un redo(T), attua le modifiche registrate.

Ripristino/abort

- A seguito di un crash di sistema, il SO controlla il log e per ogni transazione T registrata nel log:
  - se a <T, start> non corrisponde un <T, commit> il SO esegue l'operazione undo(T) che ripristina tutti i valori modificati dalla transazione eseguita in modo parziale
  - se a <T, start> corrisponde <T, commit> il SO verifica che le modifiche registrate siano state effettivamente eseguite. È possibile che al crash alcune modifiche registrate nel log non fossero ancora state apportate ai dati veri e propri, in questo caso il SO esegue un redo(T), attua le modifiche registrate.
- Questo genere di meccanismo è usato dai sistemi di journaling proposti dai moderni SO per verificare velocemente la consistenza del file system dopo un crash
- La registrazione dei dati introduce un overhead di lavoro, rende un po' meno efficiente l'esecuzione, però fornisce garanzie irrinunciabili in certe applicazioni

Tutto il logfile?

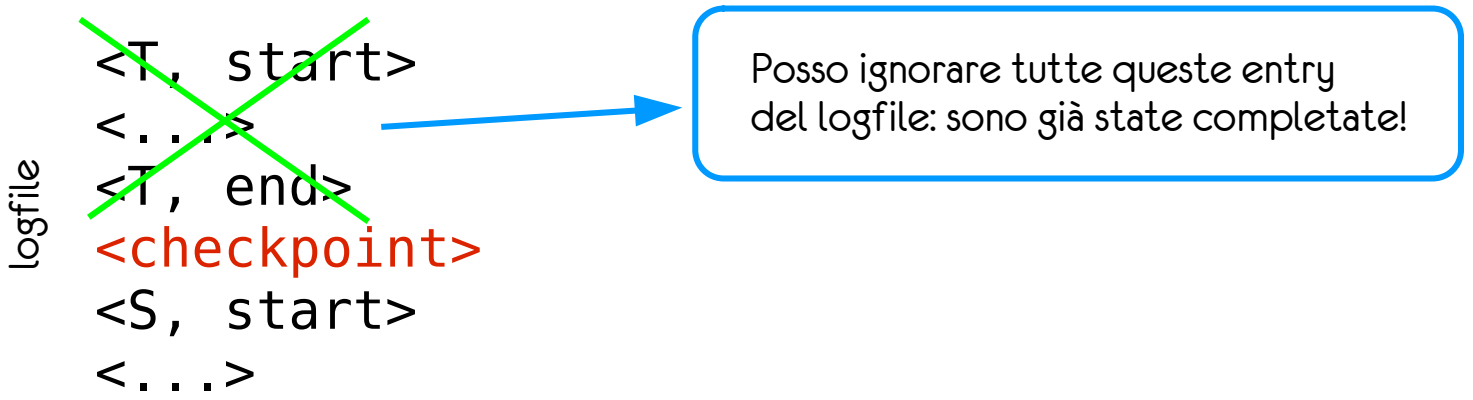
- Una domanda lecita è se ad ogni crash sia proprio necessario scorrere l'intero logfile, prendendo in considerazione anche transazioni ormai felicemente concluse da tempo ...
- Minore è il numero di transazioni considerate maggiore l'efficienza del ripristino!

Tutto il logfile?

- Una domanda lecita è se ad ogni crash sia proprio necessario scorrere l'intero logfile, prendendo in considerazione anche transazioni ormai felicemente concluse da tempo ...
- Minore è il numero di transazioni considerate maggiore l'efficienza del ripristino!
- **Checkpoint**:
  - si introduce una entry <checkpoint> nel logfile dopo quelle transazioni tali che:
    1) tutti i record del logfile relativi alla transazione sono stati riportati in memoria stabile
    2) tutte le operazioni di scrittura registrate nel logfile sono state applicate con successo

Uso dei checkpoint

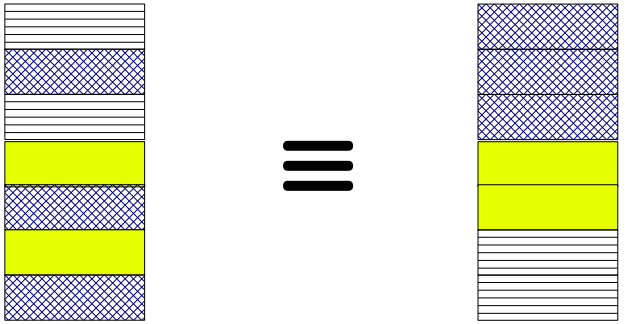
- Quando si ha un crash di sistema, si scorre a ritroso il logfile fino a identificare la prima occorrenza di **<checkpoint>**
- Tutto le operazioni che precedono questa etichetta possono essere ignorate perché per definizione di checkpoint tutte le modifiche sono già in memoria stabile
- Le operazioni di undo e redo vengono solo applicate alle transazioni successive al checkpoint



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, serializzandole sul serio tramite un semaforo mutex ma si tratta di una soluzione troppo rigida