Corso di Programmazione Concorrente

Programmazione Concorrente: Concetti di Base

Valter Crescenzi

http://crescenzi.inf.uniroma3.it

Sommario

- Primitive fork & join
  - Traduzione dei Diagrammi delle Precedenze
  - Espressività
- Condizioni di Bernstein
- Interferenza
- Errori dipendenti dal tempo
- Correttezza di una esecuzione concorrente
- Assunzione di progresso finito
- Starvation & Deadlock
- Fairness
- Divisibilità delle Istruzioni
- Strategie per la gestione dell'interferenza
- Tecniche per la gestione dell'interferenza
- Architettura di Riferimento

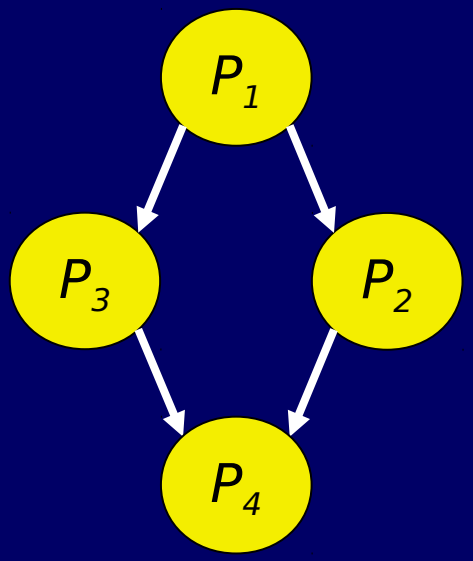
Fork & Join

- Per esprimere attività concorrenti si possono usare diversi costrutti. Nella forma più semplice, bastano:
  - *fde_id* = **fork** <*program*>

  crea un f.d.e. figlio di quello attuale mediante l'attivazione del programma specificato. Restituisce l'identificatore del f.d.e. figlio appena creato. Padre e figlio continuano indipendentemente il loro avanzamento.
  - **join** *fde_id*

  il f.d.e. corrente rimane bloccato sino a quando non termina il f.d.e. con l'identificatore specificato
- In genere le implementazioni dotano ciascun flusso di esecuzione di proprie aree di memoria dati (*record di attivazione*) mentre il codice può essere condiviso. In questo caso si parla di programmi o procedure *rientranti*

Traduzione di un Diagramma delle Precedenze con fork & join



concurrent Procedure P_1
begin <*corpo di* P_1>; **end**
concurrent Procedure P_2
begin <*corpo di* P_2>; **end**
...
begin
  var p_2_id, p_3_id : *f.d.e. id;*
  $P_1()$;
  p_2_id=**fork** P_2; p_3_id=**fork** P_3;
  join p_2_id; **join** p_3_id;
  $P_4()$;
end

Esercizi

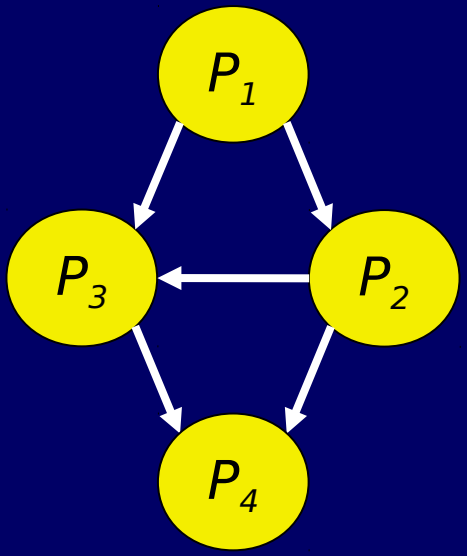
Esercizio: tradurre con fork e join il diagramma delle precedenze per il calcolo e la stampa delle prime quattro potenze di un dato numero.

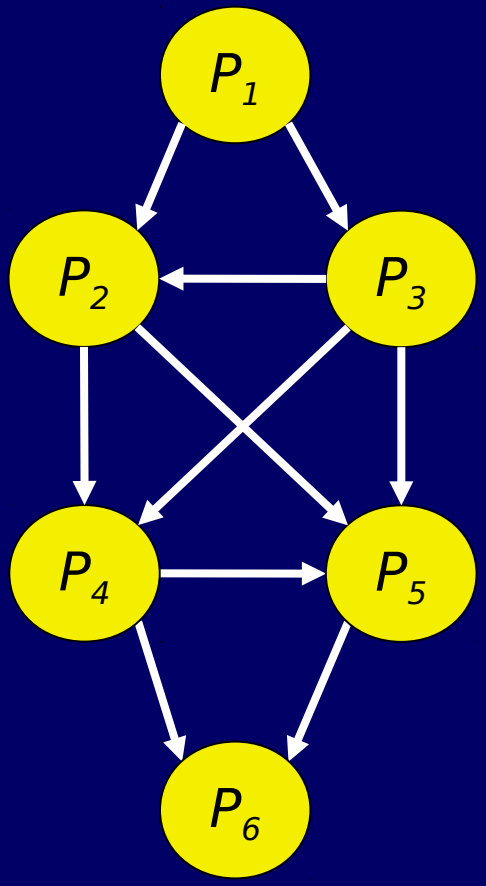
Esercizio: disegnare il diagramma delle precedenze per il calcolo e la stampa progressiva delle prime N potenze di un dato numero; tradurre con fork e join il diagramma delle precedenze trovato. Ragionare sulla presenza del ciclo di iterazione, come esprimerlo ed interpretarlo in un diagramma delle precedenze.

Esercizio: disegnare il diagramma delle precedenze per il calcolo del prodotto di due matrici interi; tradurre con fork e join il diagramma delle precedenze trovato.

Esercizi

Esercizio: tradurre con fork e join i diagrammi delle precedenze mostrati accanto.



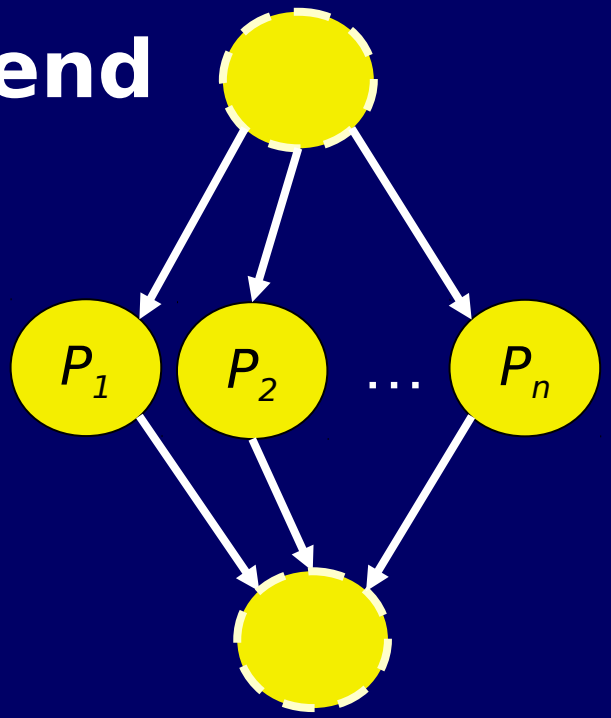


Fork & Join: Espressività

- Queste due primitive sono sufficienti a tradurre un qualsiasi diagramma delle precedenze
- Vantaggi:
  - flessibilità
  - espressività
- Svantaggi:
  - basso livello di astrazione
    - rispetto al diagramma delle precedenze, il programmatore è costretto a specificare la creazione dei flussi
  - non impongono alcuna particolare struttura al programma concorrente

Altri Costrutti per Esprimere Programmi Concorrenti

- **cobegin** P_1 || P_2 || … || P_n **coend**
- Esegue n istruzioni concorrentemente
- Non sono sufficientemente espressive per esprimere qualsiasi diagramma delle precedenze
- Risulteranno comode per esprimerne taluni



Quando Eseguire Concorrentemente?

- Dato un programma sequenziale, non è difficile costruire un equivalente diagramma delle precedenze
- Tuttavia è opportuno stabilire un criterio generale per capire se due istruzioni possono essere eseguite concorrentemente o meno:
  - per ottenere diagrammi delle precedenze che esprimono il massimo grado di parallelismo possibile
  - per automatizzare il calcolo dei vincoli che esprimono

Quando è lecito eseguire concorrentemente due istruzioni i_a e i_b ?

Dominio e Rango

- Indichiamo con A, B, … X, Y, … un'area di memoria
- Sia *i* una istruzione
  - dipende da una o più aree di memoria che denotiamo domain(*i*), ovvero dominio di *i*
  - altera il contenuto di una o più aree di memoria che denotiamo range(*i*) di *i*, ovvero rango (o codominio) di *i*

- Ad es. per la procedura **P**

```
procedure P
begin
  X ← A + X;
  Y ← A * B;
end
```

domain(**P**) = {A, B, X}

range(**P**) = {X, Y}

Condizioni di Bernstein

Quando è lecito eseguire concorrentemente due istruzioni i_a e i_b ?

- se valgono le seguenti condizioni, dette Condizioni di Bernstein:
  - $range(i_a) \cap range(i_b) = \emptyset$
  - $range(i_a) \cap domain(i_b) = \emptyset$
  - $domain(i_a) \cap range(i_b) = \emptyset$

Condizioni di Bernstein (2)

- Si osservi che non si impone alcuna condizione su

  domain(i_a) ∩ domain(i_b)
- Sono banalmente estendibili al caso di tre o più istruzioni

- Esempi di violazione per le due istruzioni:
  - *X ← Y + 1; X ← Y – 1; (violano la 1.)*
  - *X ← Y + 1; Y ← X - 1 ; (violano la 2. e la 3.)*
  - *scrivi X; X ← X + Y; (violano la 3.)*

Effetti delle Violazioni

- Quando un insieme di istruzioni soddisfa le condizioni di Bernstein, il loro esito complessivo sarà sempre lo stesso indipendentemente dall'ordine e dalle velocità relative con cui vengono eseguite
  - in altre parole, indipendentemente dalla particolare sequenza di esecuzione seguita dai processori
  - ovvero, *sarà sempre equivalente ad una loro esecuzione seriale*

- Al contrario, in caso di violazione, gli errori dipendono dall'ordine e dalle velocità relative generando il fenomeno dell'*interferenza*

Nozioni di Correttezza per Esecuzioni Concorrenti

- Ne esistono diverse, e sono state oggetto di intensi studi anche da parte di più comunità scientifiche, incluse quelle interessate ai:
  - Sistemi Concorrenti/Distribuiti
  - Sistemi Operativi
  - Database
- Maurice Herlihy, Jeannette M. Wing: *Linearizability: A Correctness Condition for Concurrent Objects*. ACM Trans. Program. Lang. Syst. 12(3): 463-492 (1990)
- Gray, J.N., Lorie, R.A., Putzulo, G.R., Traiger, I.L. *Granularity of Locks and Degrees of Consistency in a Shared Database*. Research Report RJ1654, IBM, September, 1975.
- Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, Irving L. Traiger: The Notions of Consistency and Predicate Locks in a Database System. Commun. ACM (CACM) 19(11):624-633 (1976)
- Ben-Ari, M. *Principles of Concurrent Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1982.
- Leslie Lamport, "*How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*", IEEE Trans. Comput. C-28,9 (Sept. 1979), 690-691.

Esempio di Interferenza (1)

- La disponibilità di un volo di una compagnia aerea è memorizzata in *POSTI=1*. Due signori nel medesimo istante ma da due postazioni distinte, chiedono rispettivamente di prenotare l'ultimo posto e di disdire la prenotazione già effettuata
- Le due richieste vengono tradotte in queste sequenze di istruzioni elementari indivisibili:

procedure Prenota
begin
  $R_a \leftarrow POSTI - 1;$
  $POSTI \leftarrow R_a;$
end

procedure Disdici
begin
  $R_b \leftarrow POSTI + 1;$
  $POSTI \leftarrow R_b;$
end

Esempio di Interferenza (2)

- L'esecuzione concorrente da luogo ad una qualsiasi delle possibili sequenze di interleaving. Consideriamo un campione di tre sequenze:

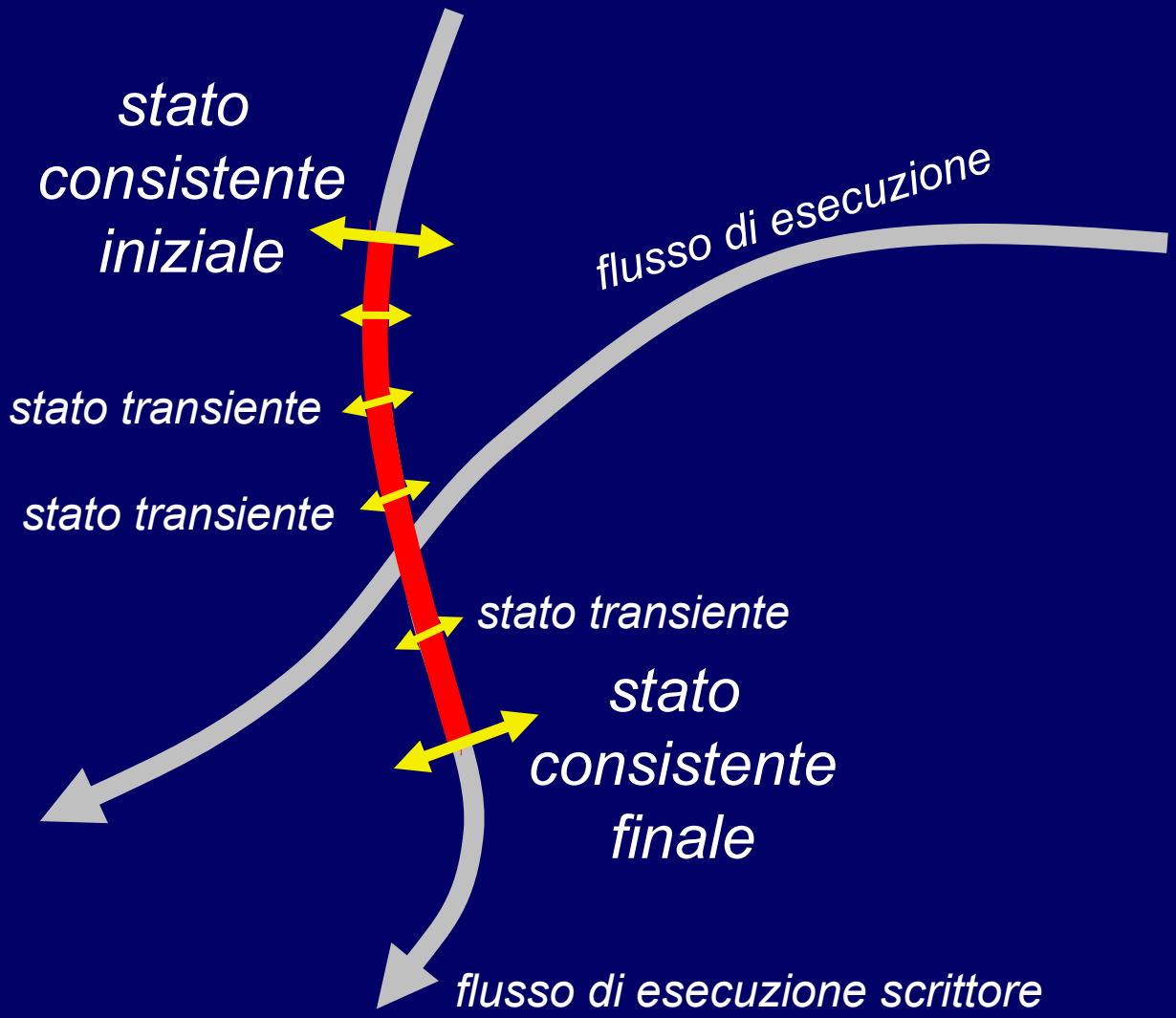
$R_a \leftarrow POSTI - 1;$	$R_a \leftarrow POSTI - 1;$	$R_b \leftarrow POSTI + 1;$
$R_b \leftarrow POSTI + 1;$	$POSTI \leftarrow R_a;$	$R_a \leftarrow POSTI - 1;$
$POSTI \leftarrow R_b;$	$R_b \leftarrow POSTI + 1;$	$POSTI \leftarrow R_a;$
$POSTI \leftarrow R_a;$	$POSTI \leftarrow R_b;$	$POSTI \leftarrow R_b;$
(POSTI=0)	(POSTI=1)	(POSTI=2)

esecuzione sequenziale

Interferenza

- Si ha interferenza in presenza di
  - due o più flussi di esecuzione
  - almeno un flusso di esecuzione scrivente
- Perché
  - un flusso esegue un cambio di stato dell'area di memoria in maniera non atomica
  - gli stati *transienti* che intercorrono tra quello iniziale a quello finale sono visibili a flussi di esecuzione diversi da quello che li sta producendo
  - le piattaforme di riferimento non offrono la possibilità di fare aggiornamenti atomici

Origine dei Fenomeni di Interferenza



Istruzioni Divisibili

- Le piattaforme a cui faremo riferimento non offrono la possibilità di fare aggiornamenti atomici

  (cfr. *Transactional Memory*)
- Salvo e*splicito* avviso in senso contrario, tutte le istruzioni di tutte le piattaforme utilizzate saranno considerate *divisibili*
- Si tratta di una ragionevole astrazione delle piattaforme moderne, che offrono principalmente istruzioni senza fornire garanzie sulla loro atomicità, tranne casi molto particolari e ben documentati
- N.B. Una scelta in senso contrario renderebbe il corso dipendente dalla piattaforma

Errori Dipendenti dal Tempo

- L'interferenza causa errori particolarmente temibili perché dipendenti dalla sequenza di interleaving effettivamente eseguita

- Temibili perché:
  - ciascuna sequenza di esecuzione può produrre effetti diversi
  - la scelta della particolare sequenza adottata è (dal punto di vista del programmatore) casuale

Caratteristiche degli Errori Dipendenti dal Tempo

- irriproducibili: possono verificarsi con alcune sequenze e non con altre
- indeterminati: esito ed effetti dipendono dalla sequenza
- latenti: possono presentarsi solo con sequenze rare
- difficili da verificare, e testare: perché le tecniche di verifica e testing si basano sulla riproducibilità del comportamento

Il Programmatore e gli Errori Dipendenti dal Tempo

- Il programmatore non può fare alcuna assunzione:
  - sulla particolare sequenza di interleaving eseguita, ovvero
  - sulle velocità relative dei vari processori virtuali
  - su un qualsiasi altro tipo di sincronismo legato alla specifica implementazione dei processori virtuali
  - sulla indivisibilità delle istruzioni
- Un programma che implicitamente od esplicitamente basa la propria correttezza su ipotesi circa la velocità relativa dei vari processori, è *scorretto*
- Esiste una sola assunzione che possono fare i programmatori sulla velocità dei processori virtuali…

Assunzione di Progresso Finito

Tutti i processori virtuali hanno una velocità finita non nulla

- Questa assunzione è l'unica che si può fare sui processori virtuali e sulle loro velocità relative

Starvation & Deadlock (1)

- Esistono due diverse situazioni che possono invalidare l'assunzione di progresso finito
  - starvation: quando un f.d.e. rimane in attesa di un evento che pure si verifica infinite volte
    - un sistema di f.d.e. che garantisce contro questa evenienza si dice che gode della proprietà di *fairness*
  - deadlock (o stallo): quando due o più f.d.e. rimangono in attesa di eventi che non potranno mai verificarsi a causa di condizioni cicliche nei f.d.e. e nella richiesta di risorse
    - esempio classico: un processo P_a possiede una risorsa R_1 e richiede una risorsa R_2 già posseduta da un altro processo P_b; quest'ultimo a sua volta richiede l'uso di R_1

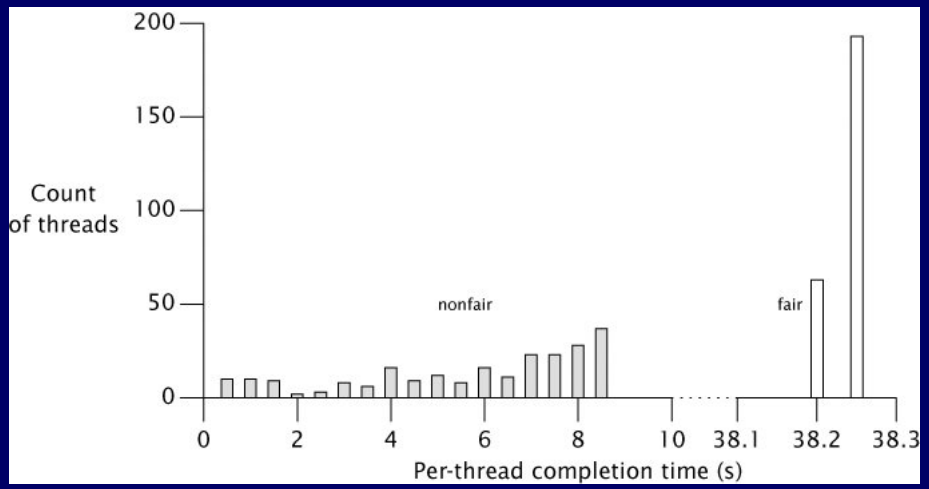
Starvation & Deadlock (2)

- proprietà di *fairness*
  - Accezione stringente, non bastano argomentazioni probabilistiche: se esiste anche una sola sequenza di esecuzione ammissibile in cui un flusso non avanza mai un algoritmo è considerato *unfair*
  - Chiamiamo *unfair* anche una simile sequenza
- Osservazioni: un algoritmo *unfair* ma che dal punto di vista probabilistico sembra produrre solo seq. *unfair* con probabilità tendenti allo zero potrebbe nella pratica causare uno di questi scenari:
    - Starvation: le seq. *unfair* diventa probabile a causa di fattori trascurati nella modellazione
    - Forte varianza nei tempi di attesa di un f.d.e.
  - >> deadlock (o stallo)

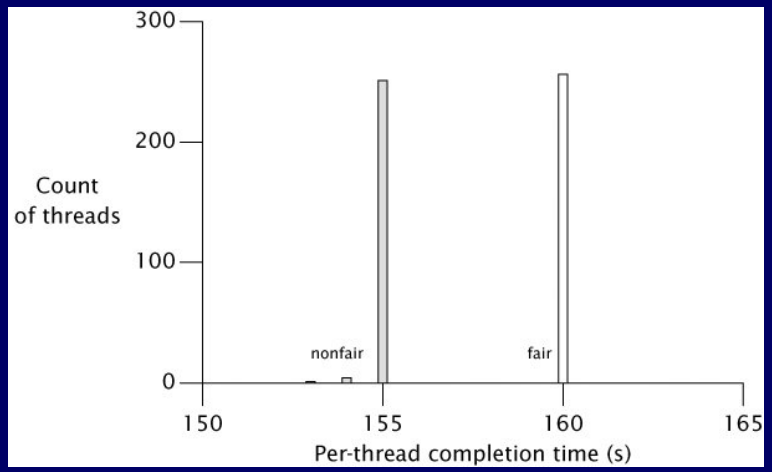
Unfairness: conseguenze

- Forte impatto nelle performance
- Varianza dei tempi di esecuzione
- Il contesto detta la scelta più opportuna

Varianza dei tempi di esecuzione



Tempo di esecuzione



Divisibilità delle Istruzioni ed Assunzione di Progresso Finito

- Salvo diverso ed esplicito avviso in senso contrario, assumeremo che tutte le istruzioni di cui faremo uso, in qualsiasi linguaggio di programmazione utilizzato nel corso, astratto o concreto, siano divisibili
  - ✔ N.B. anche le più elementari

- Ipotesi diverse richiederebbero una conoscenza di dettaglio dei linguaggi e delle piattaforme utilizzate

Interazione tra Flussi di Esecuzione Concorrenti

- Due flussi di esecuzione possono essere:
  - disgiunti
  - interagenti
    - competizione: due o più flussi di esecuzione chiedono l'uso di una risorsa comune riusabile e di molteplicità finita
    - cooperazione: due o più flussi di esecuzione cooperano per raggiungere un obiettivo comune

Competizione

- La competizione occorre ogni qualvolta che c'è una risorsa riusabile condivisa e di molteplicità finita (spesso seriale)
  - incrocio stradale
  - sportellista alle poste
  - stampante dipartimentale
  - rete ethernet
- In presenza di competizione è necessario "gestire" i possibili fenomeni di interferenza

Caratteristiche Rilevanti dell'Esecutore

- Quale strategia risulta più opportuna per gestire l'interferenza dipende largamente
  - dalla tollerabilità degli effetti
  - dalla rilevabilità dei fenomeni di interferenza
  - dalla recuperabilità degli effetti eventualmente cancellando, ripetendo e disfacendo alcune operazioni
- Ad esempio i DBMS moderni possiedono interi sotto-sistemi unicamente dedicati alla gestione della concorrenza con diverse politiche

Strategie per Gestire l'Interferenza

- La strategia migliore per gestire l'interferenza dipende fortemente dal contesto, ed in particolare almeno da questi elementi
  - La natura delle conseguenze
  - La possibilità di recuperarla
  - La possibilità di rilevare l'interferenza
  - Il livello di competizione

- Le strategie si classificano in
  - Ottimistiche/Progressiste/*try-and-see*
  - Pessimistiche/Conservatrici/*check-and-act*

Strategie per Gestire l'Interferenza

- Conseguenze
  - inaccettabili
    - ad es. incrocio stradale
  - trascurabili
    - applicazioni non critiche
  - rilevabili e controllabili
    - ad es. iteratori
  - rilevabili e recuperabili
    - ad es. rete ethernet
- …

- Strategie
  - evitare qls interferenza
    - conservatrici
  - ignorare
    -
  - rilevare ed evitare
    - fail-fast
  - rilevare e ripetere
    - ottimistiche
- …

Tecniche per la Gestione dell'Interferenza

Le strategie trovano concretezza in alcune tecniche di programmazione per la scrittura di codice privo di interferenza

- immutabilità delle aree di memoria
- confinamento degli aggiornamenti
  - per flusso di esecuzione
  - per aree di memoria …
- esclusione delle sequenze di interleaving (sincronizzazione)

Programmazione Concorrente ed Architetture degli Elaboratori

- Architetture più diffuse
  - **SMP**
  - Cluster di elaboratori
  - Data-flow, per il calcolo vettoriale
    - –

- Scenario più comune e di riferimento per questo corso: pochi processori, f.d.e. a “grana grossa”
  - flussi di esecuzione sequenziali debolmente connessi

    ...tuttavia i framework per la decomposizione parallela discussi a fine corso hanno senso in scenari dviersi >>
  - solo alcuni cenni a tecniche utili nello scenario esattamente opposto (GPGPU programming) >>