CONVIVENDO CON I DEADLOCK

Introduzione
Modello del sistema
Caratterizzazione dei deadlock
Metodi per la gestione dei deadlock

Sammarco A.,De Caro A. – 2004/2005

INTRODUZIONE

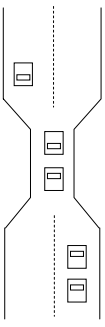
Protocollo di accesso alle risorse

- **Richiesta :** se la richiesta non può essere soddisfatta in modo immediato,causa diverso utilizzo della risorsa,il processo richiedente deve attendere fino alla possibilità di acquisire la risorsa richiesta
- **Utilizzo :** il processo può operare sulla risorsa
- **Rilascio :** il processo rilascia la risorsa

Il problema dei deadlock

- Operazioni parallele tra diversi dispositivi pilotate da processi concorrenti contribuiscono all'aumento significativo delle prestazioni. L'altra faccia della medaglia è la contesa delle risorse che può causare deadlock tra i processi interessati.
- *Defininizione di Deadlock* : è una situazione dove un gruppo di processi sono permanentemente bloccati e che competono per le risorse di sistema o comunicano fra loro .

Esempio : Attraversamento di un Ponte



- In ogni istante, sul ponte possono transitare autoveicoli solo in una direzione.
- Ciascuna sezione del ponte può essere immaginata come una risorsa.
- Se si verifica un deadlock, il *recovery* può essere effettuato se un'auto torna indietro (rilascia la risorsa ed esegue un *rollback*).
- In caso di deadlock, può essere necessario che più auto debbano tornare indietro.
- È possibile si verifichi starvation (attesa indefinita).

Modello del Sistema(Risorse)

Si distinguono due categorie di risorse :

- **Riutilizzabili**
- **Non Riutilizzabili**

Risorse Riutilizzabili

- Questo tipo di risorsa può essere usata in modo sicuro da un processo alla volta,ed essere riutilizzata in seguito da un altro processo
- Spesso la causa di possibili stalli risiede dentro la complessa logica del programma,quindi si necessita una strategia in vincoli sull'ordine delle richieste di risorse al livello di progettazione
- Esempio di Risorse Riutilizzabili : Processori,canali I/O,Memoria Centrale e Secondaria,Basi di Dati,Semafori,File,Dispositivi

Risorse Non Riutilizzabili

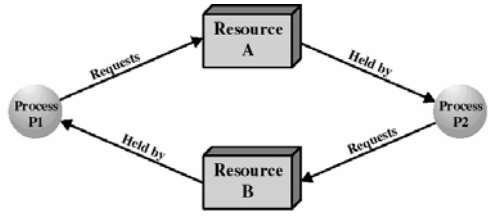
- Questo tipo di risorsa dopo essere stata creata,viene consumata (distrutta)
- Un processo produttore,se non bloccato,può produrre un numero illimitato di risorse di questo tipo
- Esempio di Risorse Non Riutilizzabili : interrupt,segnali,messaggi,dati contenuti nel buffer I/O

Caratterizzazione dei deadlock

Ci sono condizioni necessarie per le quali è possibile che si verifichi un deadlock:

- 1. **Mutua esclusione** — esiste almeno una risorsa non condivisibile, cioè un solo processo alla volta può usare quella risorsa.
- 2. **Possesso ed attesa** — un processo, che possiede almeno una risorsa, attende di acquisire ulteriori risorse possedute da altri processi.
- 3. **Impossibilità di prelazione** — una risorsa può essere rilasciata dal processo che la possiede solo volontariamente, al termine del suo compito.
- 4. **Attesa circolare** — esiste un insieme $\{P_0, P_1, \ldots, P_n\}$ di processi in attesa, tali che P_0 è in attesa di una risorsa che è posseduta da P_1, P_1 è in attesa di una risorsa posseduta da P_2, …, P_{n-1} è in attesa di una risorsa posseduta da P_n, e P_n è in attesa di una risorsa posseduta da P_0.

- **Le prime tre condizioni sono necessarie ma non sufficienti** per avere uno stallo,e la quarta condizione è in realtà una conseguenza delle prime tre.
- Graficamente l' attesa circolare si può indicare

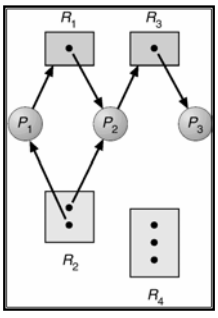


Grafo di allocazione delle risorse

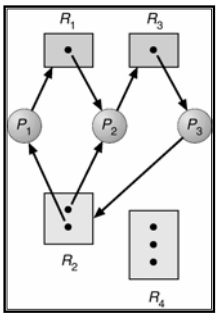
- Un grafo G(V,E) è costituito da un insieme di vertici (o nodi) V variamente connessi mediante un sistema di archi E
- L'insieme V è partizionato in due sottoinsiemi $P=\{P_1,P_2,\ldots,P_n\}$ costituito dall'insieme dei processi del sistema,ed un sottoinsieme $R=\{R_1,R_2,\ldots,R_m\}$ rappresentante delle risorse del sistema
- Arco di richiesta P_i -> R_j
- Arco di Assegnazione R_j -> P_i

Esempio Grafo di Allocazione

Grafo di allocazione risorse



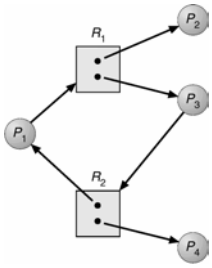
Grafo di allocazione risorse con deadlock



Un ciclo senza deadlock

- Ci sono istanze multiple per le risorse R1 ed R2
- P2 e P4 possono completare per liberare le risorse per P1 e P3

- Se il grafo non contiene cicli ⇒ no ci sono deadlock.
- Se il grafo contiene un ciclo ⇒
  - Se vi è una sola istanza per ogni tipo di risorsa, allora si ha un deadlock.
  - Se si hanno più istanze per tipo di risorsa, allora il deadlock è possibile (ma non certo).



Metodi per la gestione dei deadlock

- **Prevenzione dei deadlock**
  - Annullare uno delle tre condizioni necessarie (Mutua esclusione, Possesso e Attesa, Assenza di Prelazione) per il possibile verificarsi di un deadlock o prevenire il verificarsi dell'Attesa Circolare
- **Esclusione dei deadlock**
  - Non concedere una richiesta di risorse se quell'allocazione può portare a un deadlock
- **Rilevamento dei deadlock**
  - concedere le richieste di risorse quando possibile ma controllando periodicamente la presenza di deadlock e nel caso prendere opportune azioni per il ripristino da esso
- **Ignorare il problema**
  - fingere che i deadlock non avvengano mai nel sistema; impiegato dalla maggior parte dei SO, incluso UNIX.

Prevenzione dei deadlock

Consiste nell' annullare uno delle tre condizioni necessarie (Mutua esclusione, Possesso e Attesa, Assenza di Prelazione) per il possibile verificarsi di un deadlock o prevenire il verificarsi dell'Attesa Circolare

- **Mutua esclusione** — non è richiesta per risorse condivisibili; deve valere invece per risorse che non possono essere condivise
- **Possesso e attesa** — occorre garantire che, quando un processo richiede una risorsa, non ne possieda altre.
  - Richiedere ad un processo di stabilire ed allocare tutte le risorse necessarie prima che inizi l'esecuzione, o consentire la richiesta di risorse solo quando il processo non ne possiede alcuna.
  - Basso impiego delle risorse. È possibile che si verifichi l'attesa indefinita (starvation).

Prevenzione dei deadlock (2)

- **Assenza di prelazione —**
  - Se un processo, che possiede alcune risorse, richiede un'altra risorsa che non gli può essere allocata immmediatamente, allora rilascia tutte le risorse possedute.
  - Le risorse rilasciate (prelazionate al processo) vengono aggiunte alla lista delle risorse che il processo sta attendendo.
  - Il processo viene avviato nuovamente solo quando può ottenere sia le vecchie che le nuove risorse.
- **Attesa circolare** — si impone un ordinamento totale su tutti i tipi di risorsa e si pretende che ciascun processo richieda le risorse in ordine crescente. Questo protocollo previene i deadlock ma spesso nega le risorse riducendo le prestazioni.

Esclusione dei deadlock

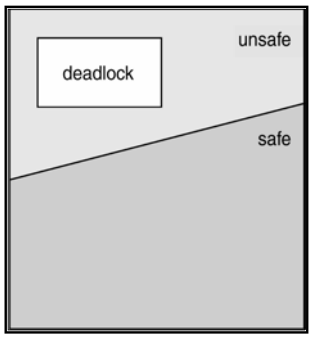
- Si accettano le tre condizioni necessarie (Mutua esclusione, Possesso e Attesa, Assenza di Prelazione) per il possibile verificarsi di un deadlock ma si adottano scelte "giudiziose" per assicurare che non si raggiunga il deadlock.
- I due approcci:
  - non iniziare un processo se tutte le sue richieste conducono ad uno stato di deadlock ("Process Initiation Denial")
  - non concedere un incremento di richieste di risorse se l'allocazione delle stesse conduce ad uno stato di deadlock ("Resource Allocation Denial")
- In ogni caso, il numero massimo di richieste per risorse deve essere dichiarato in anticipo

Resource Allocation Denial

- Quando un processo richiede una risorsa disponibile, il sistema deve decidere se l'allocazione immediata lasci il sistema in **stato sicuro**.
- Il sistema si trova in uno stato sicuro se esiste una **sequenza sicura** di esecuzione di tutti i processi.
- La sequenza $<P_1, P_2, \ldots, P_n>$ è sicura se, per ogni P_i, le risorse che P_i può ancora richiedere possono essergli allocate sfruttando le risorse disponibili + le risorse possedute da tutti i P_j, con $j < i$.
  - Se le richieste di P_i non possono essere soddisfatte immediatamente, allora P_i può attendere finché P_j ha terminato.
  - Quando P_j ha terminato, P_i può ottenere le risorse richieste, eseguire i suoi compiti, restituire le risorse allocate e terminare.
  - Quando P_i termina, P_{i+1} può ottenere le risorse richieste, etc.

Stato Sicuro

- Se un sistema è in stato sicuro ⇒ non si evolve verso il deadlock.
- Se un sistema è in stato non sicuro ⇒ si può evolvere in deadlock.
- In stato non sicuro, il SO non può impedire ai processi di richiedere risorse la cui allocazione porta al deadlock.
- *Deadlock–avoidance* ⇒ garantisce che un sistema non entri mai in stato non sicuro.



Algoritmo del banchiere

- Ciascun processo deve dichiarare a priori il massimo impiego di risorse.
- Quando un processo richiede una risorsa può aver bisogno di attendere.
- Quando un processo prende tutte le sue risorse deve restituirle in un tempo finito.

Strutture dati

Sia *n* = numero di processi, e *m* = numero di tipi di risorse.

□ *Available:* Vettore delle disponibilità di lunghezza *m*.

A	B	C
3	3	2

□ *Max: Matrice[nxm]* delle richieste massime per risorsa di un processo.

□ *Allocation : Matrice[nxm]* delle risorse allocate ad un processo.

□ *Need : Matrice[nxm]* delle risorse ancora necessarie ad un processo.

Need [*i,j*] = *Max* [*i,j*] – *Allocation* [*i,j*]

	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Algoritmo di richiesta risorse

□ $Request_i$: Vettore delle richieste per il processo P_i.

A	B	C
1	0	2

if ($Request_i > Need_i$)
    throw new IllegalArgumentException("Invalid Request")
while ($Request_i > Available$)
    wait() // P_i deve attendere, le risorse non sono disponibili
$Available = Available – Request_i$
$Allocation_i = Allocation_i + Request_i$
$Need_i = Need_i – Request_i$

- *Se lo stato è sicuro ⇒ le risorse vengono definitivamente allocate a P_i.*
- *Se lo stato è non sicuro ⇒ P_i deve attendere, e viene ripristinato il vecchio stato di allocazione delle risorse.*

Algoritmo di verifica della sicurezza

Sia *n* = numero di processi, e *m* = numero di tipi di risorse.

define *Work*[m] = *Available;* define *Finish*[n] = false
while (true) {
    int processIndex = -1
    for (int i = 0 ; i < n ; i++)
        if (*Finish*[i] == false && $Need_i <= Work$)
            processIndex = i; break;
    if (processIndex != -1) {
        $Work = Work + Allocation_{processIndex}$
        Finish[processIndex] = *true*
    } else break
}
return (*Finish[0..n-1] == true)*

Esempio di applicazione

- 5 processi, da P_0 a P_4; 3 tipi di risorse: *A* (10 istanze), *B* (5 istanze), e *C* (7 istanze).
- Istantanea al tempo T_0:

Available

A	B	C
3	3	2

Allocation

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

Max

	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Need = Max - Allocation

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Verifica Stato Sicuro

Available

A	B	C
3	3	2

Work

A	B	C
10	8	8

	Finish
P_0	F
P_1	F
P_2	F
P_3	F
P_4	F

Allocation

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

Need

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

$< P_1 P_3 P_4 P_2 P_0 >$

Rilevamento dei deadlock

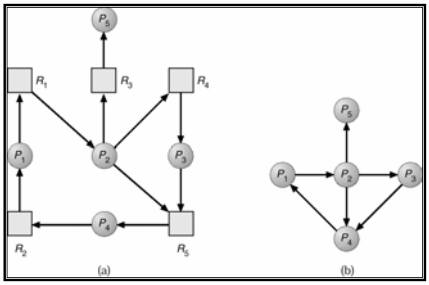
- Si permette al sistema di entrare in uno stato di deadlock
- Sono necessari:
  - Algoritmo di rilevamento del deadlock
  - Algoritmo di ripristino dal deadlock

Rilevamento: Singola Istanza

- Impiega un ***grafo di attesa***:
  - I nodi rappresentano i processi.
  - L'arco $P_i \rightarrow P_j$ esiste se P_i è in attesa che P_j rilasci una risorsa che gli occorre.
- Periodicamente viene richiamato un algoritmo che verifica la presenza di cicli nel grafo.
- Un algoritmo per rilevare cicli in un grafo richiede un numero di operazioni dell'ordine di n^2, dove n è il numero di vertici del grafo.

Grafo di allocazione risorse e grafo di attesa

Un arco $P_i \rightarrow P_j$ esiste nel grafo di attesa, se e solo se il corrispondente grafo di allocazione risorse contiene i due archi $P_i \rightarrow R_q$, $R_q \rightarrow P_j$, per una qualche risorsa R_q.



Grafo di allocazione risorse Grafo di attesa corrispondente

Impiego dell'algoritmo di rilevamento

- Quando e quanto spesso richiamare l'algoritmo di rilevamento dipende da:
  - Frequenza (presunta) con la quale si verificano i deadlock;
  - Numero di processi che vengono eventualmente influenzati dal deadlock (e sui quali deve essere effettuato un *rollback*):
- Se l'algoritmo viene richiamato in momenti arbitrari, possono essere presenti molti cicli nel grafo delle risorse $\Rightarrow$ non si può stabilire quale dei processi coinvolti nel ciclo abbia "causato" il deadlock.
- Alternativamente, l'algoritmo può essere richiamato ogni volta che una richiesta non può essere soddisfatta immediatamente o quando l'utilizzo della CPU scende al di sotto del 40%. (E' possibile usare criteri fuzzy?)

Ripristino: Terminazione di processi

- Terminazione in abort di tutti i processi in deadlock.
- Terminazione in abort di un processo alla volta fino all'eliminazione del ciclo di deadlock.
- In quale ordine si decide di terminare i processi? I fattori significativi sono:
  - La priorità del processo.
  - Il tempo di computazione trascorso e il tempo ancora necessario al completamento del processo.
  - Quantità e tipo di risorse impiegate.
  - Risorse ulteriori necessarie al processo per terminare il proprio compito.
  - Numero di processi che devono essere terminati.
  - Tipo di processo: interattivo o batch.

Ripristino: Prelazione di risorse

- **Selezione di una vittima** — È necessario stabilire l'ordine di prelazione per minimizzare i costi.
- **Rollback** — Un processo a cui sia stata prelazionata una risorsa deve ritornare ad uno stato sicuro, da cui ripartire.
- **Starvation** — Alcuni processi possono essere sempre selezionati come vittime della prelazione: includere il numero di rollback nel fattore di costo.

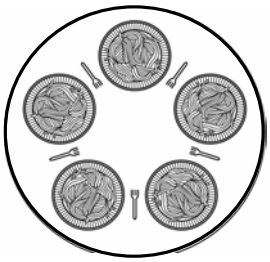
Strategia Integrata per i deadlock

- Si combinano i tre approcci di base:
  - **prevenzione**
  - **esclusione**
  - **rilevamento**

  permettendo l'uso dell'approccio ottimale per ciascuna risorsa del sistema.
- Si suddividono le risorse in classi gerarchicamente ordinate.
- Si impiegano le tecniche più appropriate per gestire i deadlock in ciascuna classe.

Il problema dei 5 filosofi

- 5 filosofi, 5 piatti di spaghetti, 5 forchette e una tavola circolare. Ognuno ha bisogno di due forchette per mangiare. Vediamo come soddisfano le precondizioni per il deadlock.
  - **Mutua Esclusione** : Forchette condivise.
  - **Possesso e Attesa**: Un filosofo potrebbe avere la forchetta sinitra ed essere in attesa della destra per sempre.
  - **Assenza di Prelazione**: Un filosofo potrebbe non voler restituire la forchetta che ha.
  - **Attesa Circolare**: Un filosofo attende un altro lungo la tavola.



Deadlocks in programmi Java

- I deadlocks possono accadere in Java perchè la parola chiave synchronized causa il blocco del thread in esecuzione in attesa del lock, o monitor, associato ad uno specifico oggetto.
- L'esempio illustra una situazione che potenzialmente può causare deadlock. Entrambi i metodi acquisiscono due lock sugli oggetti cacheLock e tableLock, prima di procedere.

```
public static Object cacheLock
= new Object();
  public static Object tableLock
= new Object();

  public void oneMethod() {
    synchronized (cacheLock) {
      synchronized (tableLock) {
        doSomething();
      }
    }
  }

  public void anotherMethod() {
    synchronized (tableLock) {
      synchronized (cacheLock) {
        doSomethingElse();
      }
    }
  }
```

Listato 1

Deadlocks non sono sempre così ovvi

- Il listato 1 può potenzialmente andare in deadlock perchè ogni metodo acquisisce due lock in ordine differente. Garantire che i locks siano sempre acquisiti in un ordine consistente, permette al programma di evitare situazioni di deadlock.
- Evidenziata l'importanza dell'ordine dei lock, è facile valutare la potenzialità di deadlock di un certo codice. Ci sono casi in cui è meno evidente la potenzialità.

Deadlocks non sono sempre così ovvi (2)

- Facciamo un esempio: esiste un metodo per trasferire fondi da un conto ad un altro. Si desidera acquisire locks su entrambi i conti prima di effettuare il trasferimento per assicurare l'atomicità dell'operazione.

```
public void transferMoney(Account fromAccount,
                          Account toAccount,
                          DollarAmount amountToTransfer) {
  synchronized (fromAccount) {
    synchronized (toAccount) {
      if (fromAccount.hasSufficientBalance(amountToTransfer) {
          fromAccount.debit(amountToTransfer);
          toAccount.credit(amountToTransfer);
      }
    }
  }
}
```

Deadlocks non sono sempre così ovvi (3)

- Consideriamo cosa succede se il thread A esegue la chiamata:

  transferMoney(accountOne, accountTwo, amount);

- mentre nello stesso istante il thread B esegue:

  transferMoney(accountTwo, accountOne, anotherAmount);

- In questo caso i due threads stanno provando ad acquisire gli stessi due lock ma in ordine differente. Il rischio di deadlock è lo stesso, ma è meno evidente la potenzialità.

Soluzione

```
public void transferMoney(Account fromAccount, Account toAccount,
                          DollarAmount amountToTransfer) {
 Account firstLock, secondLock;

 if (fromAccount.accountNumber() == toAccount.accountNumber())
   throw new Exception("Cannot transfer from account to itself");
 else if (fromAccount.accountNumber() < toAccount.accountNumber()) {
     firstLock = fromAccount;
     secondLock = toAccount;
 } else {
     firstLock = toAccount;
     secondLock = fromAccount;
 }

 synchronized (firstLock) {
   synchronized (secondLock) {
     if (fromAccount.hasSufficientBalance(amountToTransfer) {
         fromAccount.debit(amountToTransfer);
         toAccount.credit(amountToTransfer);
     }
   }
 }
}
```