

Si possono gestire i deadlock in 3 modi:

- D. Ignore:

Si ignora il problema, fingendo che non si possa mai verificare.

- D. Prevention:

Si usa un protocollo per prevenire / evitare gli stalli assicurando che non ci siano mai.

- D. Avoidance:

Si accetta che il sistema possa entrare in stallo e se ci entra di risolverlo.

I S.O. moderni ignorano i deadlock per questioni di costi

DEADLOCK PREVENTION ➤ 8.5

La prevenzione si focalizza sull'evitare che tutte le condizioni di stallo siano vere:

- mutua esclusione:

Usando risorse condivisibili si nega la mutua esclusione:
es: file in sola lettura → risorsa con 100 istanze
che però non è sempre applicabile (es: mutex)

- Hold and wait :

Si può negare con

- Singola risorsa:

Ogni thread prima di richiedere altre risorse, libera quelle che ha acquisito.

- "richiesta atomica"

Ogni thread prova a richiedere tutte le risorse di cui ha bisogno e se non riesce le rilascia.

Entrambe le soluzioni presentano svantaggi:

- gest. non efficace delle risorse

- potenziale starvation per risorse sempre richieste

• No Preemption:

Dato un thread con 1+ risorse alle quali bisogna assegnarne altre non disponibili immediatamente le sue risorse sono messe in prelazione e potrà riprendere quando le ha tutte.

Ha come svantaggi:

- Potenziale cambio di programma per implement.
- Non è applicabile su tutte le risorse.

• Circular Wait

Si impone un ordine nelle risorse r_0, r_1, r_2, \dots . Ogni thread con 1+ risorse può richiedere solo risorse maggiori.

NR: L'ordinamento sui **lock** non garantisce gli stalli

```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
    acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
    release(lock2);
    release(lock1);
}
```

Figura 8.7 Esempio di stallo con ordinamento dei lock.

Le policy di prevention sono essenziali per programmi / sistemi critici

DEADLOCK AVOIDANCE > 8.6

Il kernel esamina se con la prossima assegnazione o richiesta si può entrare in una situazione di **stallo**.

Se sì, lo evita.

Per fare ciò, l'S.O. deve avere conoscenza delle risorse richieste. (es: ogni thread dice il mazzo di risorse per tipo) in un manifest esterno con sbarco al programmatore

SAFE STATE 8.6.1

Il sistema si divide in

unsafe state.

• safe state :

Se si può definire un ordine tra i thread (t_1, t_2, \dots, t_n) in cui per il thread t_i si possono ancora impiegare

- le risorse libere
- le risorse di t_j $\forall j \leq i$ (attesa)

• unsafe state:

Se non è safe. Nell'unsafe l'S.O. non è in grado di evitare i deadlock

Per garantire lo stato sicuro, si possono usare diversi algoritmi:

ALGO. WITH RESOURCE ALLOCATION 8.6.7

L'S.O. definisce nel grafo anche dei claim edge che rappresentano le possibili richieste che un thread può richiedere.

Questo prevede un censimento iniziale dei claim prima che il thread inizi oppure quando ha solo claim edges.

A questo punto, l'S.O. è in grado di rilevare ed evitare i cicli con n^2 operazioni $n = \# \text{ thread}$.

ALGO. DEL BANCULERE 8.6.3

Prevede che ogni thread dichiari il num. mass di istanze per tipo di risorse nella sua esecuzione.

Quando richiede risorse, deve specificare se la richiesta può portare in stato unsafe.

- se no, le risorse sono fornite subito
- se sì, il thread potrebbe aspettare che un numero accettabile di risorse si liberi.

REALIZZAZIONE: DATA STRUCTURES

Dati $t = \# \text{ di thread}$ e $r = \# \text{ di risorse}$

- available[r]: indica le istanze disponibili per la risorsa r
- max[t][r]: definisce la richiesta max di ciascun thread sulla risorsa r.
- assigned[t][r]: definisce le v risorse già assegnate dei thread
- need[t][r]: def. la necessità residua di istanze v di risorse

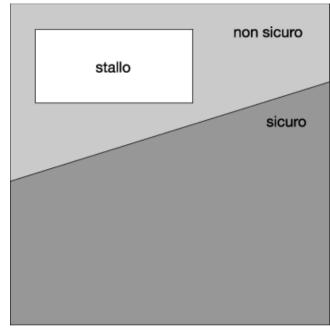


Figura 8.8 Spazi degli stati sicuri, non sicuri e di stallo.

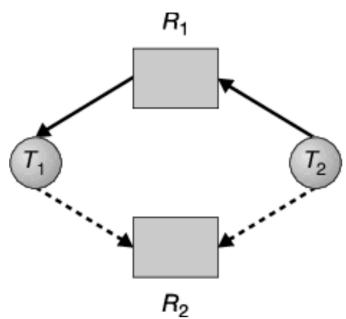


Figura 8.9 Grafo di assegnazione delle risorse per evitare le situazioni di stallo.

per ogni thread.

ALGORITMO DI VERIFICA DELLA SICUREZZA:

Si definiscono $\text{work}[0, \dots, r] = \text{available}$ e

$\text{finish}[0, \dots, t] = \text{false}$:

1. Cerco un thread $0 \leq i \leq t$ tale che

- $\text{finish}[i] = \text{false}$: non ha finito

- $\text{need}_i \leq \text{work}_i$: le cui risorse per terminare sono disponibili.

1.T: trovo $i \rightarrow$ Ne simulo la completazione

$\text{work} = \text{work} + \text{assigned}_i$; $\text{finish}[i] = \text{true}$

e ripeto il ciclo con il prossimo thread

1.F: $\exists i$ che soddisfa i criteri \rightarrow interrizzo ciclo

2. se $\text{finish}[i] = \text{true} \forall i \in \{0, \dots, t\}$, allora è in stato sicuro, altrimenti no.

Questa verifica può richiedere $t^2 \cdot r$

ALGORITMO DI RICHIESTA 8.6.3.2

Dato il thread $0 \leq i \leq t$, si definisce $\text{request}[t][r]$ la richiesta di istanze per risorsa:

1.? Verifico che la richiesta sia in linea con la necessità residua: $\text{request}_i \leq \text{need}_i$

1.T se si \rightarrow vai avanti

1.F se no \rightarrow riporta errore per num. mass. di req.

2.? Verifico che la richiesta non superi le risorse disp: $\text{request}_i \leq \text{available}_i$

2.T se si \rightarrow vai avanti

2.F se no \rightarrow il thread i deve attendere le risorse

3. simulo l'assegnazione delle risorse:

$\text{available} \leftarrow \text{available} - \text{request}_i$: decrem. le ris. disponib

$\text{assigned}_i \leftarrow \text{assigned}_i + \text{request}_i$: increm. le ris. disponib

need ; ← need ; - request ; : decrem. le necessità

4.7 chiamo l'algoritmo di sicurezza con i dati simulati

4.7.T: se è in safe state : si persistono le modifiche.

4.7.F: altrimenti : si annullano le modifiche.

DEADLOCK DETECTION > 8.7

Si possono rilevare a partire dal wait-for graph, ovvero un grafico creabile dal resource allocation che descrive le varie dipendenze senza le risorse.

L'S.O. chiamerà ogni tanto qualche algoritmo per creare il grafo.

- vantaggi: costo minore
- svantaggi: bisogna avere il resource graph.

(costo di logging)

La frequenza dipende dall'S.O., che porterà al rollback dei vari thread coinvolti.

RIPRISTINO DEADLOCK 8.8.1

Soluzioni possibili possono essere:

- terminazione di tutti i processi
- terminazione di 1 processo alla volta, che può essere deciso per priorità, tempo, ecc. (costo minimo)

ALGORITMO DI RILEVAZIONE 8.7.2

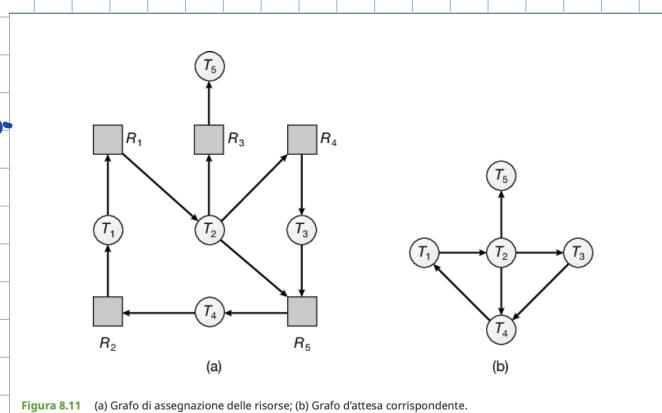
Si definiscono

- available[0,...,r]
- request[t][r]
- assigned[t][r]

L'algo definisce:

- work = available → vettore di supporto

- finish[t] = $\begin{cases} \text{assigned}_t \neq 0 \Rightarrow \text{false} \\ \text{assigned}_t = 0 \Rightarrow \text{true} \end{cases}$ → Il thread t è finito se non ha risorse



1. Cerco un thread $0 \leq i \leq t$ tale che

- $\text{finish}[i] = \text{false}$: non ha finito
- $\text{req}_k \subseteq \text{work}$: le cui risorse per terminare sono disponibili.

1.T: trovo $i \rightarrow$ Ne simulo la completazione

$\text{work} = \text{work} + \text{assigned}; \text{finish}[i] = \text{true}$

e riproto il ciclo con il prossimo thread

1.F: $\exists i$ che soddisfa i criteri \rightarrow interrompo ciclo

2.? se esiste un thread i che non è finito

$\exists i \text{ finish}[i] = \text{false}$

2.T Il thread i è in deadlock.

2.F Il sistema non ha deadlock.

Questa verifica può richiedere $t^2 \cdot r$