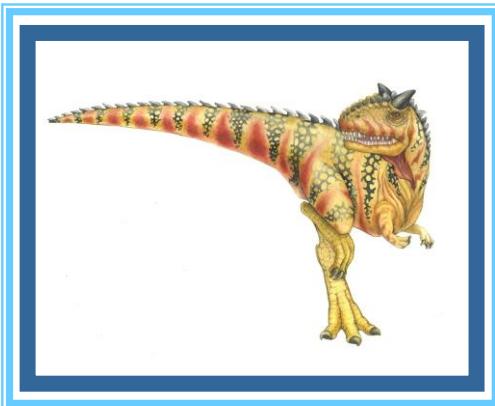


Deadlock

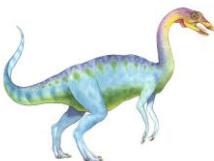




Obiettivi

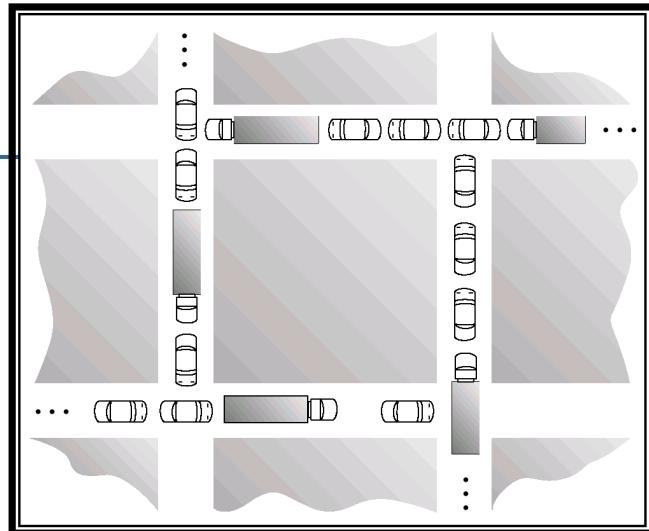
- ❖ Illustrare come e quando può verificarsi il deadlock
- ❖ Definire le quattro condizioni necessarie che lo caratterizzano
- ❖ Identificare il deadlock in un grafo di allocazione delle risorse
- ❖ Valutare diversi approcci per preveire il deadlock
- ❖ Applicare l'algoritmo del Banchiere per evitare il deadlock
- ❖ Applicare l'algoritmo di rilevamento dei deadlock
- ❖ Valutare gli approcci per il recupero da situazioni di deadlock

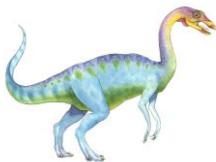




Sommario

- ❖ Il problema del deadlock
- ❖ Modello del sistema
- ❖ Deadlock nelle applicazioni multithread
- ❖ Caratterizzazione dei deadlock
- ❖ Metodi per la gestione dei deadlock
 - Prevenire i deadlock
 - Evitare i deadlock
 - Rilevare i deadlock
 - Ripristino da situazioni di deadlock





Il problema del deadlock – 1

- ❖ In un ambiente multiprogrammato più thread possono entrare in competizione per ottenere un numero finito di risorse
- ❖ **Deadlock** o *stallo* — insieme di thread bloccati, ciascuno dei quali “possiede” (almeno) una risorsa ed attende di acquisire una o più risorse allocate ad altri thread dell’insieme (che detengono risorse, ma ne attendono altre)
- ❖ Nella programmazione multithread, gli stalli si verificano più frequentemente, poiché più thread possono competere per un insieme (limitato) di risorse condivise





Il problema del deadlock – 2

- ❖ Difficoltà intrinseca nell'identificare deadlock che, generalmente, si verificano solo in particolari condizioni di scheduling
- ❖ Anche se esistono applicazioni che possono rilevare i programmi suscettibili di stallo, la maggior parte dei SO attuali non offre strumenti di prevenzione
- ❖ Progettare programmi che non provochino deadlock è responsabilità dei programmatori
 - Compito sempre più difficile nei sistemi multicore in cui, più in generale, tutti i problemi di liveness, si presentano con maggior frequenza e complessità





Esempi di deadlock

❖ *Ciascun processo in un insieme di processi attende un evento che può essere causato solo da un altro processo dell'insieme*

❖ Esempio 1

- Nel sistema sono presenti due stampanti
- T_1 e T_2 si sono assicurati una stampante ciascuno ed entrambi richiedono anche l'altra

❖ Esempio 2

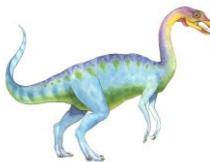
- Semafori A e B, inizializzati a 1:

T_1

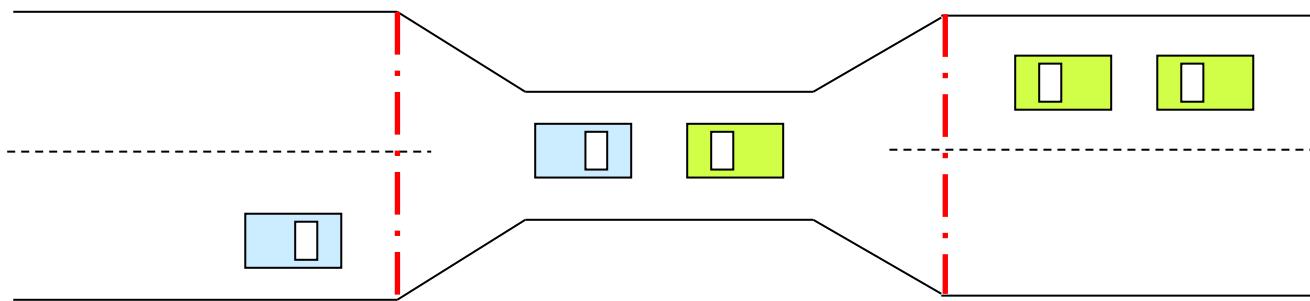
T_2

- | | |
|----------------------|---------------------|
| 1) wait (A) ; | 2) wait(B) ; |
| 3) wait (B) ; | 4) wait(A) ; |





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 (come in figura), il *recovery* può essere effettuato se un'auto torna indietro (rilascia la risorsa ed esegue un roll back)
- ❖ In generale, in caso di deadlock, può essere necessario che più auto debbano tornare indietro
- ❖ È possibile si verifichi starvation (attesa indefinita)

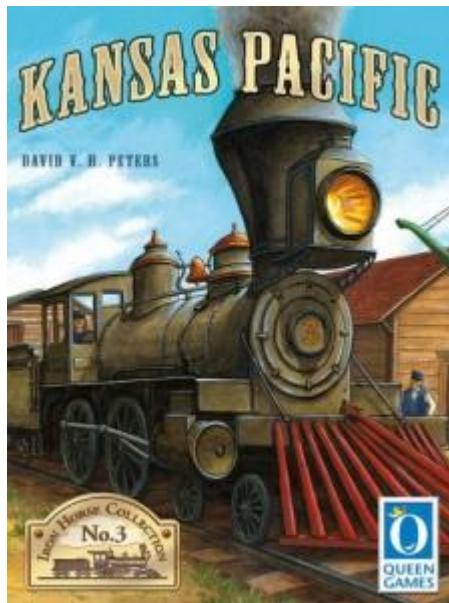




Esempio: leggi ferroviarie in Kansas

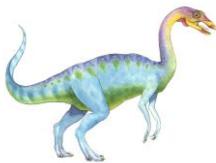
- ❖ Testo parziale della proposta di legge per la regolamentazione delle ferrovie del Kansas (Aprile 1913):

"Quando due treni convergono ad un incrocio, ambedue devono arrestarsi, e nessuno dei due può ripartire prima che l'altro si sia allontanato"



When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other is gone

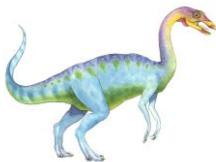




Modello di sistema – 1

- ❖ Tipi di risorse R_1, R_2, \dots, R_m
Cicli di CPU, spazio di memoria, file, dispositivi di I/O, lock, semafori
- ❖ Le risorse di una classe vengono dette istanze della classe
- ❖ Ciascun tipo di risorsa R_i ha W_i istanze
- ❖ Il numero di risorse in una classe viene detto molteplicità del tipo di risorsa
- ❖ Un processo non può richiedere una specifica risorsa, ma solo una risorsa di una specifica classe
- ❖ Una richiesta per una classe di risorse può essere soddisfatta da qualsiasi istanza di quel tipo
- ❖ Risorse ad *assegnazione statica*
 - Avviene al momento della creazione del processo e rimane valida fino alla terminazione
 - **Esempio:** descrittore del processo

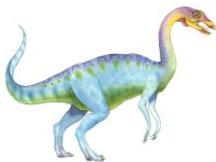




Modello di sistema – 2

- ❖ Il protocollo di accesso dei thread alle risorse ad *assegnazione dinamica* prevede...
 - **Richiesta** – se la richiesta non può essere soddisfatta immediatamente, causa diverso utilizzo della risorsa, il thread richiedente deve attendere fino al rilascio della risorsa da parte del thread che la detiene
 - **Utilizzo**
 - **Rilascio**
- ❖ Richiesta/rilascio realizzati con apposite system call, quali **request/release** device, **open/close** file, **allocate/free** memory, **wait/signal** semaphore, **acquire/release** lock mutex





Modello di sistema – 3

- ❖ Ogni volta che si usa una risorsa gestita dal kernel, il SO controlla che il thread utente ne abbia fatto richiesta e che questa gli sia stata accordata
- ❖ Una tabella di sistema registra lo stato di ogni risorsa e, se è allocata, indica il thread relativo
- ❖ Ogni risorsa può avere una coda d'attesa



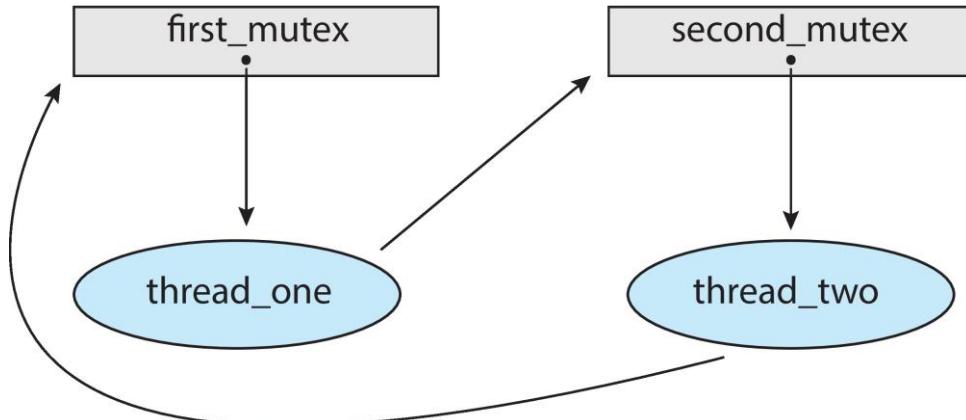


Deadlock nelle applicazioni multithread

- ❖ Si creano e si inizializzano due lock mutex

```
pthread_mutex_t first_mutex;  
pthread_mutex_t second_mutex;  
  
pthread_mutex_init(&first_mutex,NULL);  
pthread_mutex_init(&second_mutex,NULL);
```

Stallo se **thread_one** acquisisce
first_mutex e **thread_two**
acquisisce **second_mutex**



```
/* thread_one runs in this function */  
void *do_work_one(void *param)  
{  
    pthread_mutex_lock(&first_mutex);  
    pthread_mutex_lock(&second_mutex);  
    /**  
     * Do some work  
     */  
    pthread_mutex_unlock(&second_mutex);  
    pthread_mutex_unlock(&first_mutex);  
  
    pthread_exit(0);  
}
```

```
/* thread_two runs in this function */  
void *do_work_two(void *param)  
{  
    pthread_mutex_lock(&second_mutex);  
    pthread_mutex_lock(&first_mutex);  
    /**  
     * Do some work  
     */  
    pthread_mutex_unlock(&first_mutex);  
    pthread_mutex_unlock(&second_mutex);  
  
    pthread_exit(0);  
}
```



Stallo attivo – 1

- ❖ Il **livelock** o **stallo attivo** è una situazione simile al deadlock in cui, tuttavia, i thread non sono effettivamente bloccati, ma di fatto non progrediscono
- ❖ Gli stati dei thread coinvolti nel livelock cambiano costantemente l'uno rispetto all'altro
- ❖ Pertanto, il livelock può considerarsi una forma speciale di starvation
- ❖ Come il deadlock, il livelock può verificarsi solo in determinate condizioni di scheduling

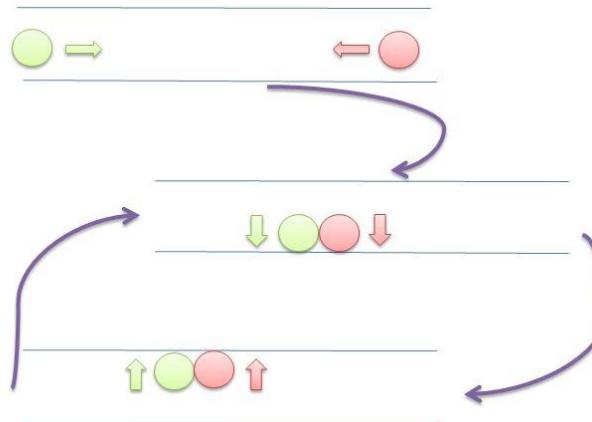




Stallo attivo – 2

❖ Esempi

- Due persone si incontrano in un corridoio e, cercando di passare, si spostano ripetutamente da un lato all'altro del corridoio stesso



- Due thread effettuano polling (busy waiting) per verificare l'uno lo stato dell'altro e non fanno progressi (**livelock** mutuo), ma non sono in deadlock dato che comunque effettuano il polling
- Collisioni su rete Ethernet
 - ▶ Soluzione: ritrasmettere un pacchetto dopo un intervallo di tempo casuale



Caratterizzazione del deadlock

- ❖ Una situazione di deadlock può verificarsi solo se valgono simultaneamente le seguenti condizioni
 - **Mutua esclusione:** nel sistema, esiste almeno una risorsa non condivisibile, cioè utilizzabile da un solo thread alla volta
 - **Possesso ed attesa:** un thread, che possiede almeno una risorsa, attende di acquisire ulteriori risorse possedute da altri thread
 - **Impossibilità di prelazione:** una risorsa può essere rilasciata dal thread che la detiene solo volontariamente, al termine del suo utilizzo
 - **Attesa circolare:** esiste un insieme $\{T_0, T_1, \dots, T_n\}$ di thread in attesa, tali che T_0 è in attesa di una risorsa che è posseduta da T_1 , T_1 è in attesa di una risorsa posseduta da T_2, \dots , T_{n-1} è in attesa di una risorsa posseduta da T_n e T_n è in attesa di una risorsa posseduta da T_0





Grafo di assegnazione delle risorse – 1

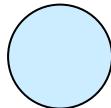
- ❖ Un grafo è costituito da un insieme di vertici (o nodi) V variamente connessi per mezzo di un insieme di archi E
- ❖ Nel **grafo di assegnazione delle risorse**...
 - L'insieme V è partizionato in due sottoinsiemi:
 - ▶ $T = \{T_1, T_2, \dots, T_n\}$ è l'insieme costituito da tutti i thread attivi nel sistema
 - ▶ $R = \{R_1, R_2, \dots, R_m\}$ è l'insieme di tutti i tipi di risorse presenti nel sistema
 - ▶ **Arco di richiesta:** $T_i \rightarrow R_j$
 - ▶ **Arco di assegnazione:** $R_j \rightarrow T_i$





Grafo di assegnazione delle risorse – 2

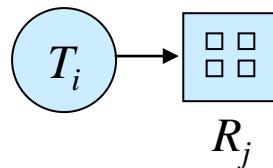
- ❖ Thread



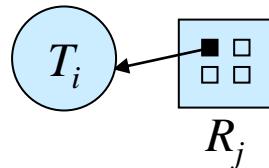
- ❖ Tipo di risorsa con 4 istanze



- ❖ T_i richiede un'istanza di R_j



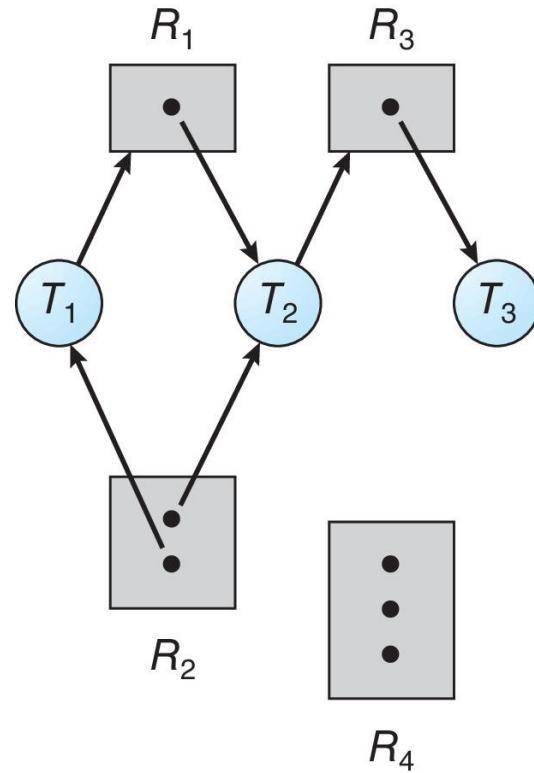
- ❖ T_i possiede un'istanza di R_j



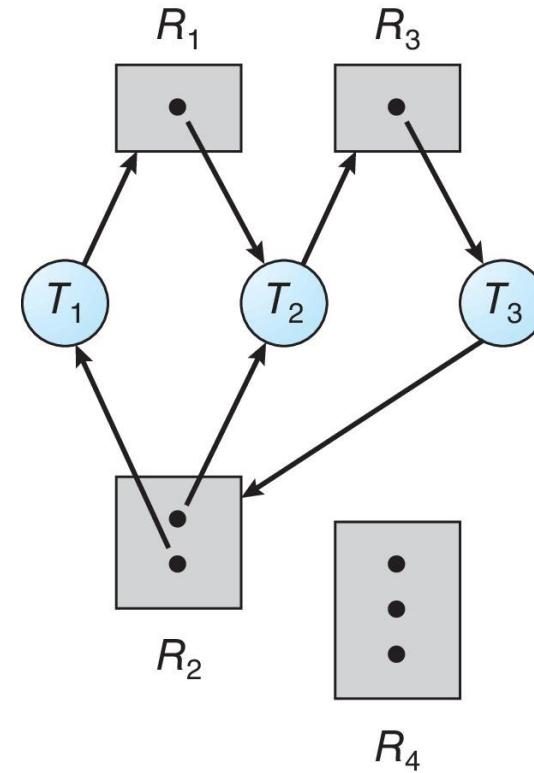


Esempi di grafo di assegnazione risorse – 1

Grafo di assegnazione risorse

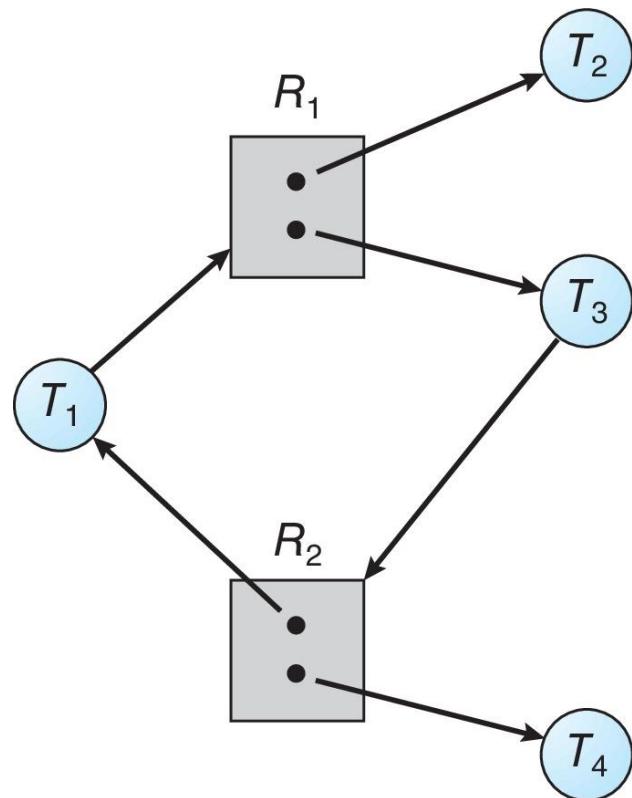


Grafo di assegnazione risorse con deadlock





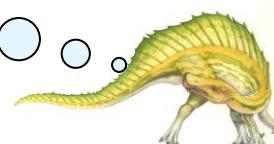
Esempi di grafo di assegnazione risorse – 2



Grafo di assegnazione risorse con un ciclo, ma privo di deadlock

- ❖ Se il grafo non contiene cicli
 - ⇒ Non 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)

Il ciclo c'è, ma
il deadlock no...

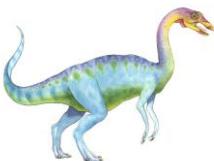




Esempio 1

- ❖ Si consideri un sistema con tre thread (T_1, T_2, T_3), e due tipologie di risorse (R_1, R_2), con la disponibilità di due risorse di tipo R_1 e una risorsa di tipo R_2
- ❖ Si consideri la seguente cronologia di richieste:
 - T_2 richiede R_1
 - T_1 richiede R_2
 - T_2 richiede R_2
 - T_3 richiede R_1
 - T_1 richiede R_1
- ❖ Verificare se il sistema si trova in stallo





Esempio 1 (cont.)

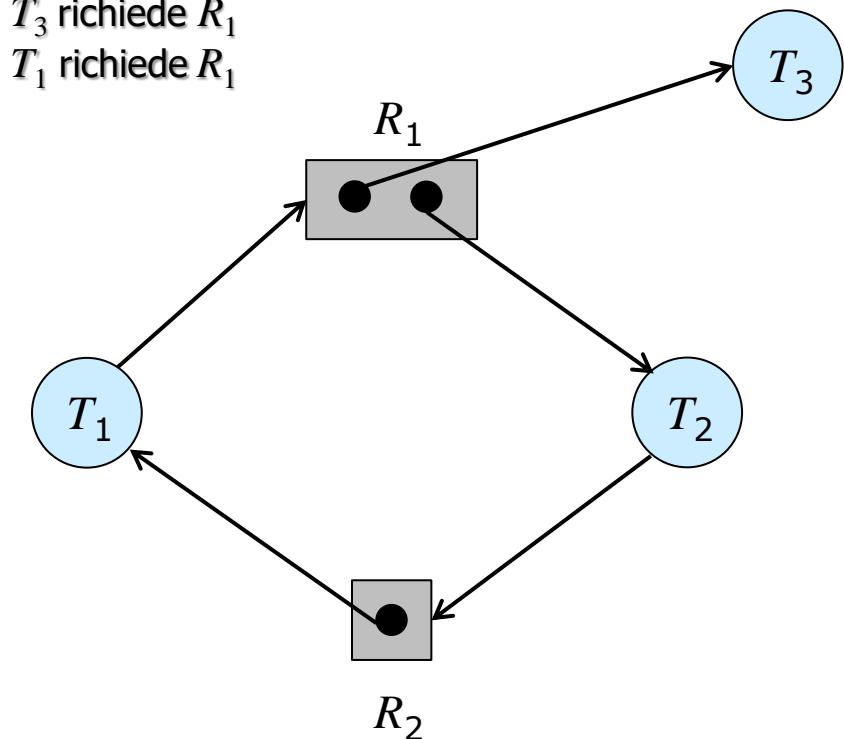
T_2 richiede R_1

T_1 richiede R_2

T_2 richiede R_2

T_3 richiede R_1

T_1 richiede R_1



- ❖ Si forma un ciclo ma il sistema **NON è in stallo**
- ❖ Infatti un elemento del ciclo (R_1) ha molteplicità due e una delle due istanze è “fuori dal ciclo”: T_3 può terminare e rilasciare la sua istanza di R_1 , che viene così assegnata a T_1





Esempio 2

- ❖ Si consideri un sistema con 4 thread (T_1, T_2, T_3, T_4) e 4 tipologie di risorse (R_1, R_2, R_3, R_4) con disponibilità:
 - 1 risorsa di tipo R_1
 - 1 risorsa di tipo R_2
 - 1 risorsa di tipo R_3
 - 2 risorse di tipo R_4
- ❖ Si assuma che:
 - ogni volta che un thread richiede una risorsa libera, questa venga assegnata al thread richiedente (l'arco di richiesta si trasforma *istantaneamente* in arco di assegnazione)
 - ogni volta che un thread richiede una risorsa già occupata, il richiedente deve attendere che la risorsa si liberi prima di impossessarsene (coda FIFO di attesa)

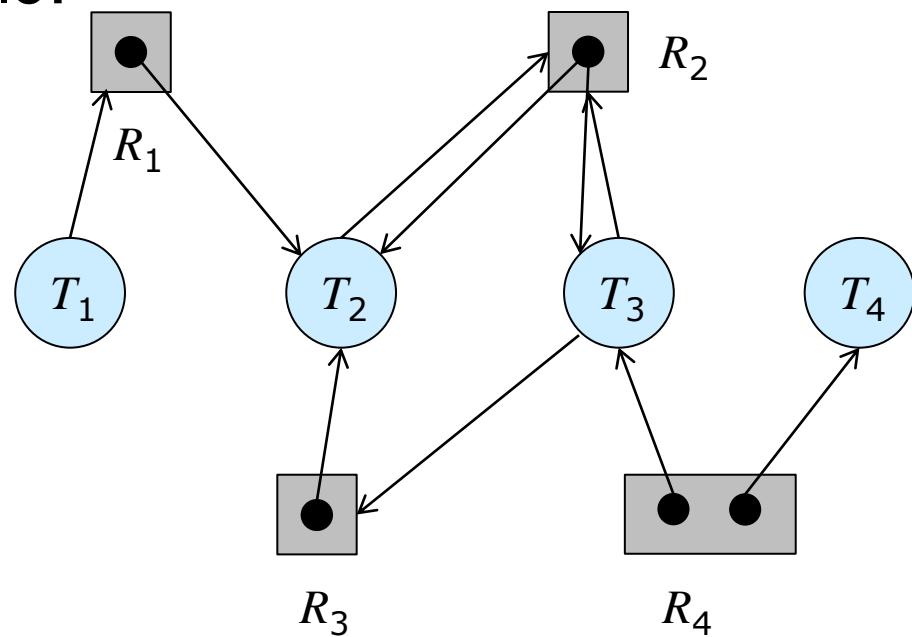




Esempio 2 (cont.)

❖ Si consideri la seguente successione cronologica di richieste e rilasci di risorse e si verifichi se, alla fine, il sistema si trova in stallo:

- T_2 richiede R_1, R_2, R_3
- T_3 richiede R_2, R_4
- T_2 rilascia R_2
- T_4 richiede R_4
- T_1 richiede R_1
- T_2 richiede R_2
- T_3 richiede R_3



Ciclo di richieste/assegnazioni che coinvolge $T_2, R_2, T_3, R_3 \Rightarrow$ il sistema è in stallo

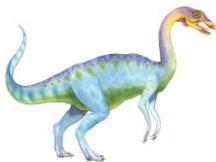




Metodi di gestione del deadlock

- ❖ Assicurare che il sistema non entri mai in uno stato di deadlock
 - **Prevenire i deadlock:** evitare che si verifichino contemporaneamente mutua esclusione, possesso e attesa, impossibilità di prelazione e attesa circolare
 - ⇒ Basso utilizzo delle risorse e throughput ridotto
 - **Evitare i deadlock:** evitare gli stati del sistema a partire dai quali si può evolvere verso il deadlock
 - ⇒ Informazioni aggiuntive sulle richieste dei thread
- ❖ Permettere al sistema di entrare in uno stato di deadlock, quindi **ripristinare** il sistema
 - Determinare la presenza di un deadlock
 - Ripristinare la corretta operatività del sistema
- ❖ Ignorare il problema e fingere che i deadlock non si verifichino mai; impiegato dalla maggior parte dei SO, inclusi Windows e UNIX/Linux





Prevenire i deadlock – 1

- ❖ Limitare le modalità di richiesta/accesso delle/alle risorse per invalidare una delle quattro condizioni necessarie
 - **Mutua esclusione** — non è richiesta per risorse condivisibili ⇒ non possono essere coinvolte in uno stallo; deve valere invece per risorse che non possono essere condivise, quindi non può essere prevenuta
 - **Possesso e attesa** — occorre garantire che, quando un thread richiede una risorsa, non ne possieda altre
 - ▶ Esigere dal thread di stabilire ed allocare tutte le risorse necessarie prima che inizi l'esecuzione, o consentire la richiesta di risorse solo quando il thread non ne possiede alcuna
 - ▶ Basso impiego delle risorse; possibile l'attesa indefinita per thread con forti richieste di risorse



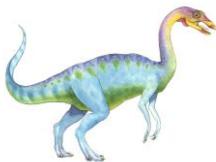


Prevenire i deadlock – 2

▪ Impossibilità di prelazione

- ▶ Se un thread, che possiede alcune risorse, richiede un'altra risorsa, che non gli può essere allocata immediatamente, allora rilascia tutte le risorse possedute
- ▶ Le risorse rilasciate (prelazionate al thread) vengono aggiunte alla lista delle risorse che il thread sta attendendo
- ▶ Il thread viene avviato nuovamente solo quando può ottenere sia le risorse precedentemente possedute sia quelle attualmente richieste
- ▶ Alternativamente, le risorse possono essere assegnate al thread che le richiede, qualora siano attualmente possedute da un altro thread in attesa
- ▶ Protocollo adatto per risorse il cui stato si può salvare e recuperare facilmente (registri, memoria), non per dispositivi di I/O o primitive di sincronizzazione





Prevenire i deadlock – 3

- **Attesa circolare** — si impone un ordinamento totale (possibilmente “logico”) su tutti i tipi di risorsa, $f: R \rightarrow N$, e si pretende che ciascun thread richieda le risorse in ordine crescente
- In alternativa, ogni thread, prima di richiedere una istanza di R_j , deve rilasciare tutte le istanze di R_i , tali che $f(R_i) \geq f(R_j)$
 - ▶ Supponiamo che, avendo imposto l’ordinamento suddetto, si verifichi comunque l’attesa circolare
 - ▶ Siano $\{T_0, T_1, \dots, T_n\}$ i thread coinvolti nell’attesa circolare, dove T_i attende la risorsa R_{i+1} posseduta dal thread T_{i+1} (in aritmetica modulare $\rightarrow T_n$ attende R_0)
 - ▶ Poiché il thread T_i possiede la risorsa R_i mentre richiede la risorsa R_{i+1} , $\forall i$, vale la condizione

$$f(R_0) < f(R_1) < \dots < f(R_n) < f(R_0) \quad \text{assurdo}$$





Esempio di attesa circolare

```
/* thread_one esegue in questa funzione */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /* Fa qualcosa... */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread_two esegue in questa funzione */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /* Fa qualcosa... */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

Si risolve imponendo, per es.,
 $f(\text{first_mutex})=1$
 $f(\text{second_mutex})=3$
e forzando l'invocazione dei lock mutex in ordine crescente

lockdep di Linux è uno strumento per rilevare possibili stalli nell'acquisizione di lock nel kernel; verifica che i lock vengano effettivamente richiesti secondo l'ordine stabilito dai loro identificativi

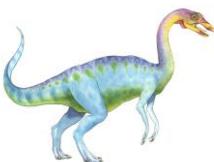




Evitare i deadlock

- ❖ Prevenire i deadlock può causare scarso utilizzo dei dispositivi e ridotta produttività del sistema
- ❖ Viceversa, evitare i deadlock presuppone che il sistema conosca a priori informazioni addizionali sulle richieste future dei thread
 - Il modello più semplice e utile richiede che ciascun thread dichiari il numero massimo di risorse di ciascun tipo di cui potrà usufruire nel corso della propria esecuzione
 - L'algoritmo di *deadlock-avoidance* esamina dinamicamente lo stato di allocazione delle risorse per assicurarsi che non si possa verificare una condizione di attesa circolare
 - Lo stato di allocazione delle risorse è definito dal numero di risorse disponibili ed allocate, e dal massimo numero di richieste dei thread

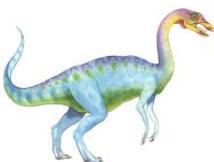




Stato sicuro – 1

- ❖ Quando un thread 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 e solo se esiste una **sequenza sicura** di esecuzione di tutti i thread
- ❖ La sequenza $\langle T_1, T_2, \dots, T_n \rangle$ è sicura se, per ogni T_i , le risorse che T_i può ancora richiedere possono essergli allocate sfruttando le risorse disponibili, più le risorse possedute da tutti i T_j , con $j < i$



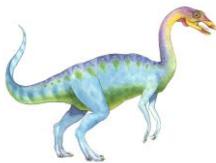


Stato sicuro – 2

❖ Cioè...

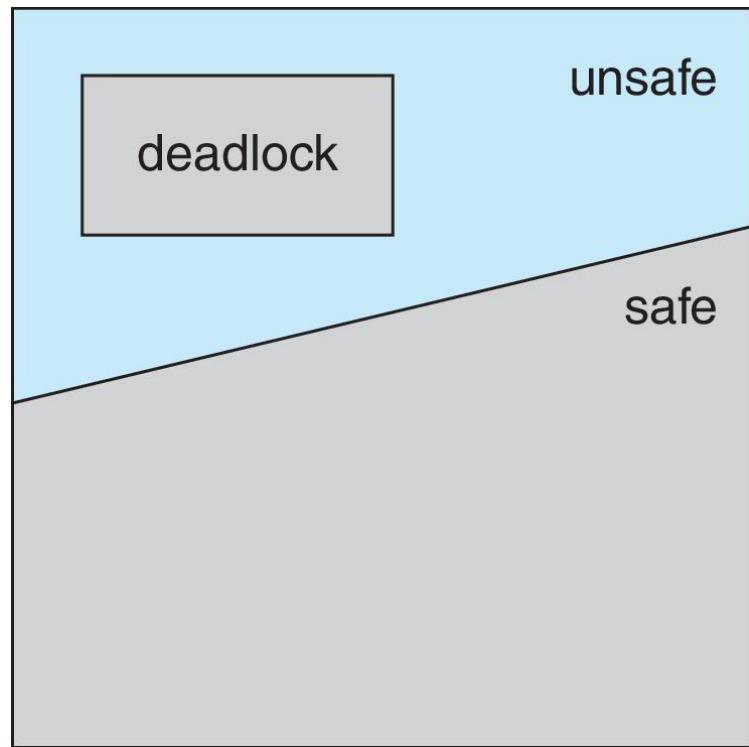
- Se le richieste di T_i non possono essere soddisfatte immediatamente, allora T_i può attendere finché tutti i T_j ($j < i$) abbiano terminato
- Quando i T_j hanno terminato, T_i può ottenere le risorse richieste, eseguire i suoi compiti, restituire le risorse allocate e terminare
- Quando T_i termina, T_{i+1} può ottenere le risorse richieste, etc.





Stato sicuro – 3

- ❖ 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 thread di richiedere risorse in modo da causare un deadlock
- ❖ *Deadlock-avoidance*
 - garantisce che un sistema non evolva mai verso uno stato non sicuro
 - se un thread richiede una risorsa disponibile, può dovere attendere
⇒ scarso utilizzo delle risorse del sistema





Stato sicuro – 4

❖ Esempio

- 12 istanze di una risorsa complessivamente disponibili
- Al tempo t_0

Thread	Richiesta max	Unità possedute
T_0	10	5
T_1	4	2
T_2	9	2

3 unità disponibili

- Stato sicuro: $\langle T_1, T_0, T_2 \rangle$ sequenza sicura
- Se all'istante t_1 il thread T_2 richiede un'ulteriore unità, e questa gli viene assegnata, lo stato diventa non sicuro (solo T_1 può terminare)





Algoritmi per evitare il deadlock

- ❖ Risorse con istanza singola: **utilizzo del grafo di assegnazione delle risorse** (versione modificata)
- ❖ Risorse con istanza multipla: **algoritmo del banchiere**



phillipmartin.com





Algoritmo con grafo di assegnazione risorse – 1

- ❖ Un **arco di reclamo** $T_i \rightarrow R_j$ (rappresentato da una linea tratteggiata) indica che il thread T_i può richiedere la risorsa R_j in futuro
- ❖ Un arco di reclamo viene convertito in un **arco di richiesta** quando il thread T_i richiede effettivamente la risorsa R_j
- ❖ Un arco di richiesta viene convertito in un **arco di assegnazione** quando la risorsa viene allocata al thread
- ❖ Quando una risorsa viene rilasciata da un thread, l'arco di assegnazione viene riconvertito in un arco di reclamo





Algoritmo con grafo di assegnazione risorse – 2

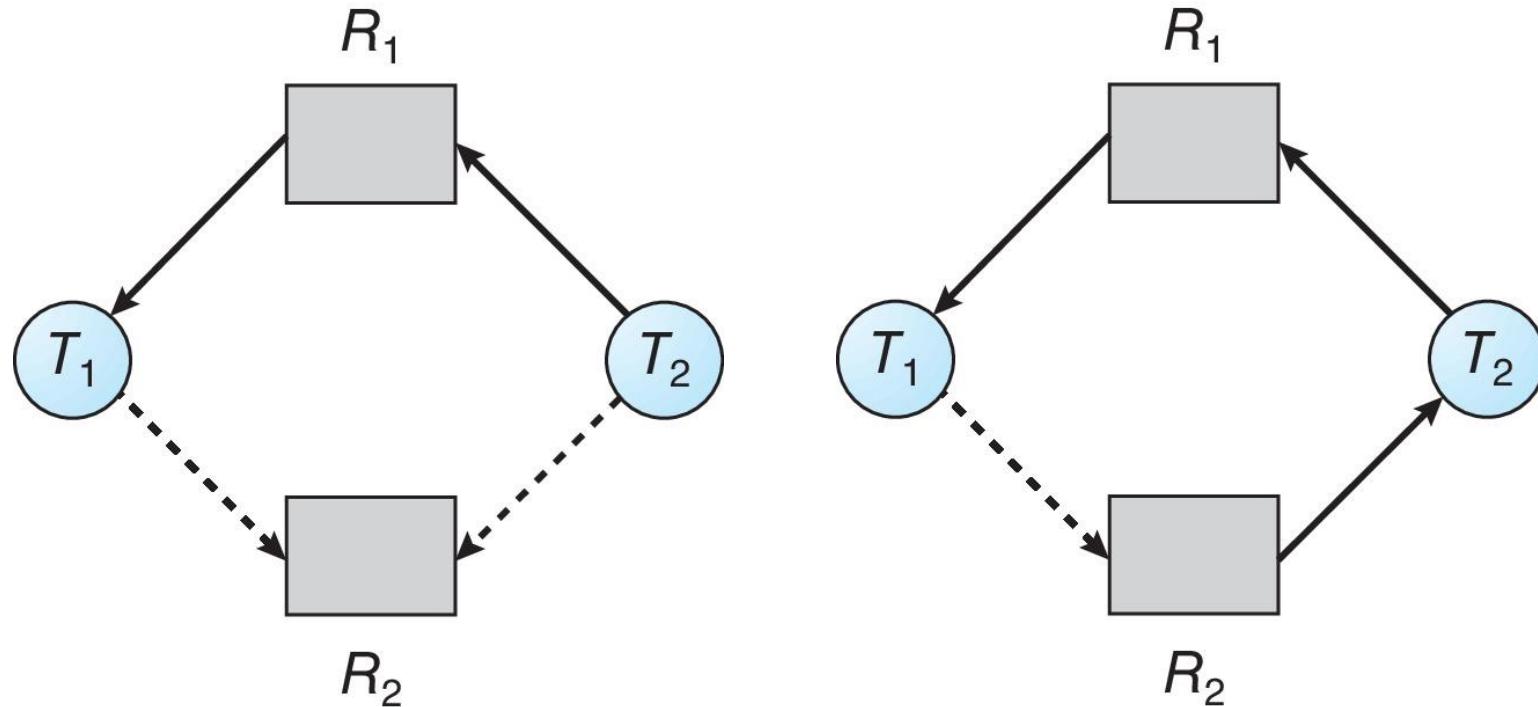
- ❖ Le risorse devono venir reclamate *a priori* nel sistema
 - Prima che il thread T_i inizi l'esecuzione, tutti i suoi archi di reclamo devono essere già inseriti nel grafo di assegnazione delle risorse
 - Alternativamente, un arco di reclamo può essere aggiunto successivamente al thread T_i se esso è connesso al grafo solo mediante archi di reclamo
- ❖ Supponiamo che il thread T_i richieda una risorsa R_j
- ❖ La richiesta può essere accordata solo se la conversione dell'arco di reclamo in arco di assegnazione non provoca il formarsi di un ciclo nel grafo di assegnazione delle risorse

Costo della ricerca di cicli in un grafo pari a $O(n^2)$





Algoritmo con grafo di assegnazione risorse – 3



Grafo di assegnazione risorse

Se T_2 richiede R_2 e gli viene
assegnata...

Stato non sicuro

⇒ Stallo se T_1 richiede R_2





Algoritmo del banchiere

- ❖ Deve il suo nome al fatto che le banche non assegnano mai tutto il denaro di cui disponono, per non restare sprovviste di fronte a nuove richieste dei clienti
- ❖ Permette di gestire istanze multiple di una risorsa (a differenza dell'algoritmo con grafo di assegnazione risorse)
- ❖ Ciascun thread deve dichiarare a priori il massimo impiego di risorse
- ❖ Quando un thread richiede una risorsa può non venir servito istantaneamente, per mantenere lo stato sicuro
- ❖ Quando ad un thread vengono allocate tutte le risorse deve restituirle in tempo finito





Strutture dati dell'algoritmo del banchiere

- ❖ Sia n il numero dei thread presenti nel sistema ed m il numero dei tipi di risorse
 - **Available:** vettore di lunghezza m ; se $\text{Available}[j]=k$, vi sono k istanze disponibili del tipo di risorsa R_j
 - **Max:** matrice $n \times m$; se $\text{Max}[i,j]=k$, il thread T_i può richiedere al più k istanze del tipo di risorsa R_j
 - **Allocation:** matrice $n \times m$; se $\text{Allocation}[i,j]=k$, a T_i sono attualmente allocate k istanze di R_j
 - **Need:** matrice $n \times m$; se $\text{Need}[i,j]=k$, T_i può richiedere k ulteriori istanze di R_j per completare il proprio task

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

- ❖ Al trascorrere del tempo, le strutture variano sia nelle dimensioni che nei valori





Algoritmo di verifica della sicurezza

1. Siano **Work** e **Finish** vettori di lunghezza m ed n , rispettivamente; si inizializzi
 - a) $\text{Work} \leftarrow \text{Available}$
 - b) $\text{Finish}[i] \leftarrow \text{false}$, per $i=1,2,\dots,n$
2. Si cerchi i tale che valgano contemporaneamente:
 - a) $\text{Finish}[i] = \text{false}$
 - b) $\text{Need}_i \leq \text{Work}$

Se tale i non esiste, si esegua il passo 4.
3. Si assegni: $\text{Work} \leftarrow \text{Work} + \text{Allocation}_i$
 $\text{Finish}[i] \leftarrow \text{true}$
si torni al passo 2.
4. Se $\text{Finish}[i] = \text{true}$ per ogni i , il sistema è in stato sicuro; altrimenti lo stato è non sicuro

Costo dell'algoritmo pari a $O(m \times n^2)$





Algoritmo di richiesta delle risorse per il thread T_i

- ❖ Sia Request_i il vettore delle richieste per il thread T_i
- ❖ Se $\text{Request}_i[j]=k$, il thread T_i richiede ulteriori k istanze del tipo di risorsa R_j
 1. Se $\text{Request}_i \leq \text{Need}_i$, si vada al passo 2.; altrimenti, si riporti una condizione di errore, poiché il thread ha ecceduto il massimo numero di richieste
 2. Se $\text{Request}_i \leq \text{Available}$, si vada al passo 3.; altrimenti T_i deve attendere, poiché le risorse non sono disponibili
 3. Il sistema simula l'allocazione a T_i delle risorse richieste, modificando lo stato di allocazione come segue:

$$\text{Available} = \text{Available} - \text{Request}_i$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i$$

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





Esempio 1

- ❖ 5 thread, da T_0 a T_4
- ❖ 3 tipi di risorse:
A (10 istanze), B (5 istanze) e C (7 istanze)
- ❖ Istantanea al tempo t_0 :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	7	5	3	3	3	2
T_1	2	0	0	3	2	2			
T_2	3	0	2	9	0	2			
T_3	2	1	1	2	2	2			
T_4	0	0	2	4	3	3			





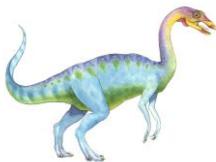
Esempio 1 (cont.)

- ❖ La matrice Need è definita come $\text{Need} = \text{Max} - \text{Allocation}$

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>		<u>Need</u>
	A B C	A B C	A B C		A B C
T_0	0 1 0	7 5 3	3 3 2	T_0	7 4 3
T_1	2 0 0	3 2 2		T_1	1 2 2
T_2	3 0 2	9 0 2		T_2	6 0 0
T_3	2 1 1	2 2 2		T_3	0 1 1
T_4	0 0 2	4 3 3		T_4	4 3 1

- ❖ Il sistema è in stato sicuro, dato che la sequenza $\langle T_1, T_3, T_4, T_2, T_0 \rangle$ soddisfa i criteri di sicurezza
- (5,3,2) (7,4,3)... si può eseguire un T_i qualsiasi





Esempio 1 (cont.)

- ❖ T_1 richiede $(1,0,2)$ $\Rightarrow \text{Request}_1 \leq \text{Need}_1$ $(1,0,2) \leq (1,2,2)$
- ❖ Verificare che $\text{Request}_1 \leq \text{Available}$ (cioè, $(1,0,2) \leq (3,3,2)$)

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	7	4	3	2	3	0
T_1	3	0	2	0	2	0			
T_2	3	0	2	6	0	0			
T_3	2	1	1	0	1	1			
T_4	0	0	2	4	3	1			

- ❖ L'esecuzione dell'algoritmo di sicurezza mostra che la sequenza $\langle T_1, T_3, T_4, T_0, T_2 \rangle$ soddisfa i requisiti di sicurezza
 - ⇒ La richiesta di T_1 viene soddisfatta immediatamente
 - ⇒ Ma dopo...
 - Può essere soddisfatta la richiesta di T_4 per $(3,3,0)$?
 - Può essere soddisfatta la richiesta di T_0 per $(0,2,0)$?





Esempio 2

- ❖ Sia dato il sistema descritto da P insieme dei processi, R insieme delle risorse R_{ij} di indice i e molteplicità j , $E(P)$ insieme delle richieste di accesso a risorse in R emesse da processi in P e attualmente pendenti, e $E(R)$ insieme degli accessi attualmente soddisfatti di processi in P a risorse in R :

$$P = \{P_1, P_2, P_3, P_4\}$$

$$R = \{R_{11}, R_{22}, R_{31}, R_{42}\}$$

$$E(P) = \{P_1 \rightarrow R_1, P_2 \rightarrow R_4, P_3 \rightarrow R_3\}$$

$$E(R) = \{R_1 \rightarrow P_2, R_2 \rightarrow (P_3, P_4), R_3 \rightarrow P_4, R_4 \rightarrow (P_1, P_3)\}$$

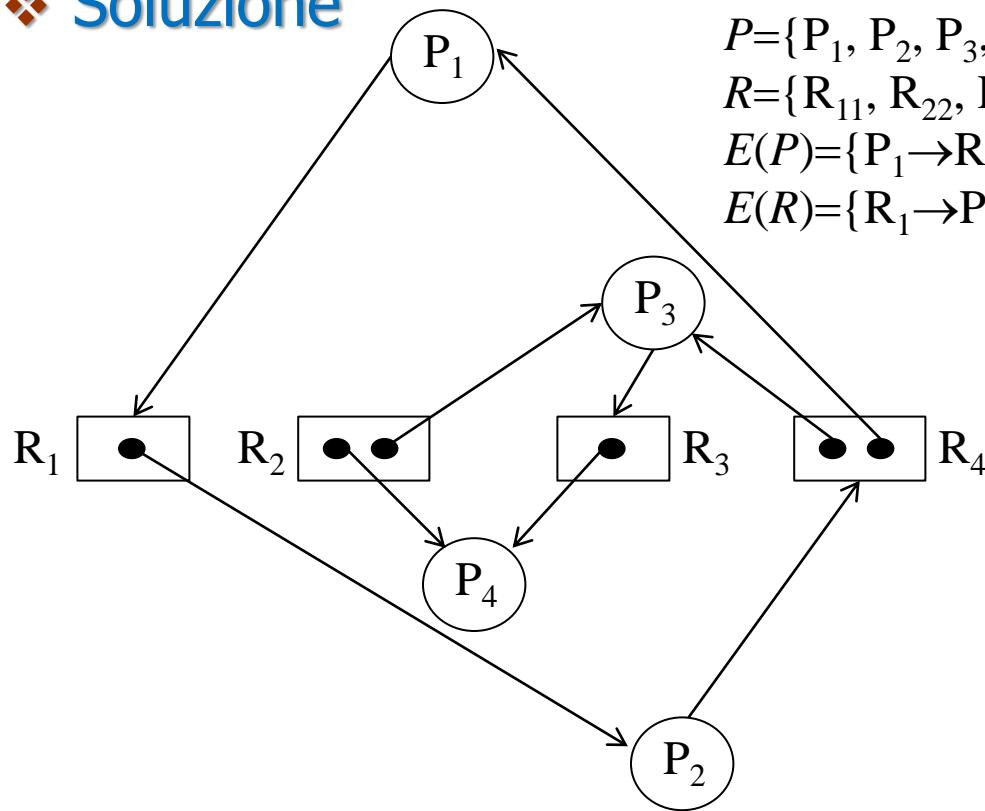
- ❖ Si analizzi il grafo di allocazione delle risorse, determinando se il sistema si trova attualmente in stallo.





Esempio 2 (cont.)

❖ Soluzione



$$P = \{P_1, P_2, P_3, P_4\}$$

$$R = \{R_{11}, R_{22}, R_{31}, R_{42}\}$$

$$E(P) = \{P_1 \rightarrow R_1, P_2 \rightarrow R_4, P_3 \rightarrow R_3\}$$

$$E(R) = \{R_1 \rightarrow P_2, R_2 \rightarrow (P_3, P_4), R_3 \rightarrow P_4, R_4 \rightarrow (P_1, P_3)\}$$

- ❖ Se P_4 termina, poiché ha tutte le risorse a disposizione, R_3 può essere concessa a P_3 , che pure terminerà rilasciando un'istanza di R_4 utile a P_2 e che interrompe il ciclo.





Esempio 3

- ❖ Si consideri la situazione in cui vi siano in esecuzione quattro thread, T_A , T_B , T_C e T_D , con le matrici relative alle risorse allocate ed al numero massimo di risorse richieste dai thread definite come segue:

$$\text{Allocation} = \begin{pmatrix} 1 & 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix} \quad \text{Max} = \begin{pmatrix} 1 & 1 & 2 & 1 & 3 \\ 2 & 2 & 2 & 1 & 1 \\ 2 & 1 & 3 & 1 & 0 \\ 1 & 1 & 2 & 2 & 1 \end{pmatrix}$$

- ❖ Se il vettore delle risorse disponibili è

$$\text{Available} = (0,0,x,1,1),$$

qual è il minimo valore di x che rende lo stato attuale sicuro?





Esempio 3 (cont.)

❖ Soluzione

- Calcolo dell matrice Need

$$\text{Need} = \begin{pmatrix} 0 & 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \quad \text{Available} = (0,0,x,1,1)$$

- Nello stato corrente, può essere eseguito solo il thread T_D , ponendo $x=1$. T_D restituisce quindi le risorse che aveva allocate e Available diviene $(1,1,x+1,2,1)$.
- Ora, solo il thread T_C può ricevere tutte le risorse necessarie, nell'ipotesi però che $x=2$. Quindi Available si aggiorna a $(2,2,x+1,3,1)$.
- Si può servire T_B , senza vincoli ulteriori sul valore di x . Il vettore Available diviene $(4,2,x+2,4,2)$, sufficiente per T_A .
- Quindi $\langle T_D, T_C, T_B, T_A \rangle$, con $x=2$.





Esempio 4

- ❖ In un giardino sono disponibili quattro vanghe e tre carriole e ci sono quattro giardinieri al lavoro. Al giardiniere₁ occorrono complessivamente due vanghe e due carriole; al giardiniere₂ occorrono due vanghe ed una carriola; al giardiniere₃ occorrono una vanga e due carriole; al giardiniere₄ occorrono tre vanghe e tre carriole. Ad un certo istante t , giardiniere₁ sta utilizzando una carriola, giardiniere₂ una vanga, giardiniere₃ una carriola e giardiniere₄ due vanghe.
- ❖ Il sistema è sicuro? Verificarlo tramite l'algoritmo del Banchiere.
- ❖ Se giardiniere₁ avesse necessità di un'ulteriore carriola, gli potrà effettivamente essere concessa? Perché?





Esempio 4 (cont.)

❖ Soluzione

- Il vettore delle risorse globali è (4,3), mentre

$$\text{Max} = \begin{pmatrix} 2 & 2 \\ 2 & 1 \\ 1 & 2 \\ 3 & 3 \end{pmatrix} \xrightarrow{\text{Al tempo } t} \text{Allocation} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 2 & 0 \end{pmatrix} \quad \text{Available} = (1,1)$$

- La matrice Need risulta

$$\text{Need} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 3 \end{pmatrix} \xrightarrow{} \begin{matrix} (2,1) \\ (2,2) \\ (2,3) \end{matrix} \quad \langle G_2, G_1, G_3, G_4 \rangle \text{ successione sicura}$$





Esempio 4 (cont.)

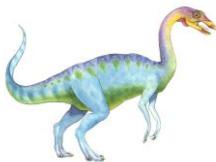
❖ Soluzione

- Se G_1 richiede un'ulteriore carriola si avrebbe

$$\text{Allocation} = \begin{pmatrix} 0 & 2 \\ 1 & 0 \\ 0 & 1 \\ 2 & 0 \end{pmatrix} \quad \text{Available} = (1,0)$$

$$\text{Need} = \begin{pmatrix} 2 & 0 \\ 1 & 1 \\ 1 & 1 \\ 1 & 3 \end{pmatrix} \rightarrow \text{Nessun giardiniere può ottenere tutte le risorse richieste}$$





Rilevamento del deadlock

- ❖ Se il SO non si avvale di tecniche per prevenire o evitare lo stallo, indirettamente si permette al sistema di entrare in uno stato di deadlock
- ❖ Per ripristinarne le funzionalità, sono necessari...
 - ...un algoritmo di rilevamento del deadlock
 - e, successivamente, un algoritmo di ripristino dal deadlock
- ❖ Costi aggiuntivi per la memorizzazione delle informazioni necessarie al ripristino, per l'esecuzione dell'algoritmo di rilevamento, e dovuti alla possibile perdita di informazioni (e calcoli già svolti) connessa al ripristino stesso





Istanza singola di ciascun tipo di risorsa

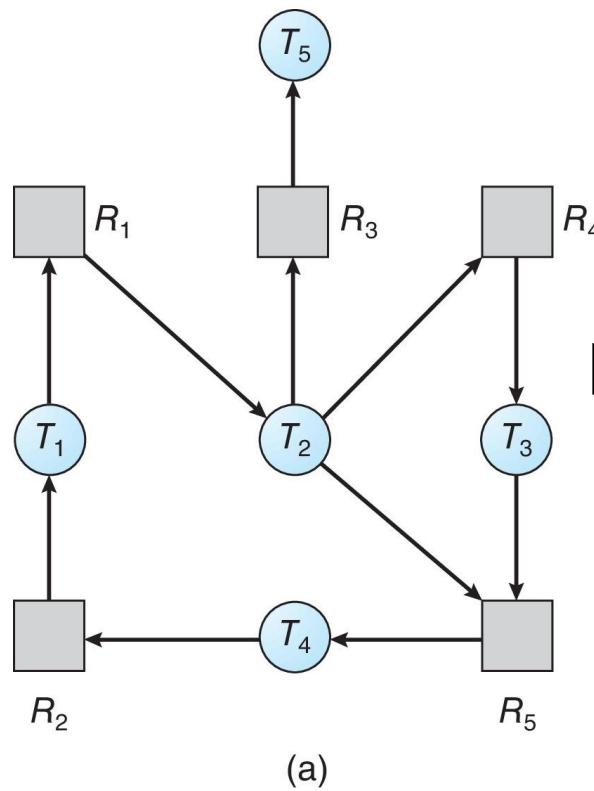
- ❖ Si impiega un **grafo di attesa**
 - I nodi rappresentano i thread
 - L'arco $T_i \rightarrow T_j$ esiste se T_i è in attesa che T_j rilasci una risorsa che gli occorre
 - Periodicamente viene richiamato un algoritmo che verifica la presenza di cicli nel grafo (che deve essere conservato e mantenuto aggiornato)
 - ▶ La presenza di un ciclo attesta una situazione di deadlock
- ❖ 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



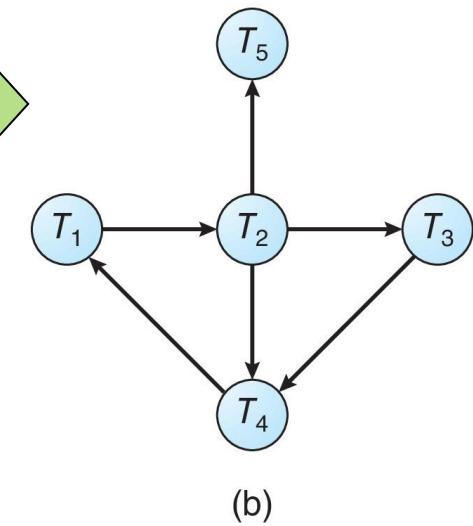


Grafo di assegnazione risorse e grafo di attesa

Un arco $T_i \rightarrow T_j$ esiste nel grafo di attesa se e solo se il corrispondente grafo di assegnazione risorse contiene i due archi $T_i \rightarrow R_q, R_q \rightarrow T_j$ per una qualche risorsa R_q



Grafo di assegnazione risorse



Grafo di attesa corrispondente





Toolkit BCC di Linux

- ❖ Fornisce lo strumento `deadlock_detector` per il tracciamento delle chiamate alle funzioni Pthreads `pthread_mutex_lock()` e `pthread_mutex_unlock()` nei processi utente
- ❖ Quando il processo specificato effettua una chiamata ad una delle due funzioni, `deadlock_detector` costruisce un grafo d'attesa dei lock mutex del processo e segnala il possibile verificarsi di stalli, se rileva un ciclo nel grafo





Più istanze per ciascun tipo di risorsa

- ❖ Occorrono strutture dati variabili, simili a quelle utilizzate per l'algoritmo del Banchiere
- ❖ **Available:** vettore di lunghezza m , indica il numero di risorse disponibili di ciascun tipo
- ❖ **Allocation:** matrice $n \times m$, definisce il numero di risorse di ciascun tipo attualmente allocate a ciascun thread
- ❖ **Request:** matrice $n \times m$, indica la richiesta corrente di ciascun thread; se $\text{Request}[i,j]=k$, il thread T_i richiede k istanze supplementari della risorsa R_j
- ❖ L'algoritmo di rilevamento indaga su ogni possibile sequenza di assegnazione per i thread da completare



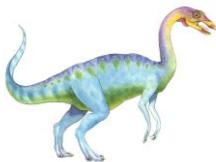


Algoritmo di rilevamento – 1

1. Siano **Work** e **Finish** due vettori di lunghezza m e n , rispettivamente; inizialmente sia:
 - a) $\text{Work} \leftarrow \text{Available}$
 - b) Per $i=1,2,\dots,n$, se $\text{Allocation}_i \neq 0$, $\text{Finish}[i] \leftarrow \text{false}$; altrimenti, $\text{Finish}[i] \leftarrow \text{true}$
2. Si trovi un indice i tale che valgano entrambe le condizioni:
 - a) $\text{Finish}[i] = \text{false}$
 - b) $\text{Request}_i \leq \text{Work}$

Se tale i non esiste, si vada al passo 4.
3. Si assegna: $\text{Work} \leftarrow \text{Work} + \text{Allocation}_i$
 $\text{Finish}[i] \leftarrow \text{true}$
Si vada al passo 2.
4. Se $\text{Finish}[i] = \text{false}$, per qualche i , $1 \leq i \leq n$, il sistema è in uno stato di deadlock; inoltre, se $\text{Finish}[i] = \text{false}$, T_i è in deadlock





Algoritmo di rilevamento – 2

❖ Note

- L'algoritmo richiede un numero di operazioni dell'ordine di $O(m \times n^2)$ per determinare se il sistema è in uno stato di deadlock
- Si ipotizza (ottimisticamente) che T_i non intenda richiedere altre risorse per completare il proprio compito, ma piuttosto che restituisca in tempo finito quelle che già possiede
 - ▶ Se ciò non accade, può verificarsi uno stallo, che verrà rilevato alla prossima verifica sul sistema





Esempio di applicazione dell'algoritmo di rilevamento

- ❖ 5 thread, da T_0 a T_4
- ❖ 3 tipi di risorse:
A (7 istanze), B (2 istanze) e C (6 istanze)
- ❖ Istantanea al tempo t_0 :

Allocation Request Available

	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

- ❖ La sequenza $\langle T_0, T_2, T_1, T_3, T_4 \rangle$ conduce a $\text{Finish}[i] = \text{true}$ per ogni i





Esempio (cont.)

- ❖ T_2 richiede un'istanza supplementare della risorsa C

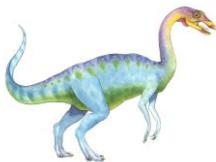
Request

	A	B	C
T_0	0	0	0
T_1	2	0	2
T_2	0	0	1
T_3	1	0	0
T_4	0	0	2

- ❖ Qual è lo stato del sistema?

- Può reclamare le risorse possedute dal thread T_0 , ma il numero di risorse disponibili non è sufficiente a soddisfare gli altri thread
- ⇒ **deadlock:** coinvolge i thread T_1 , T_2 , T_3 e T_4

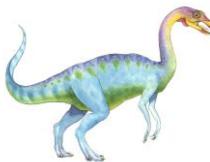




Impiego dell'algoritmo di rilevamento

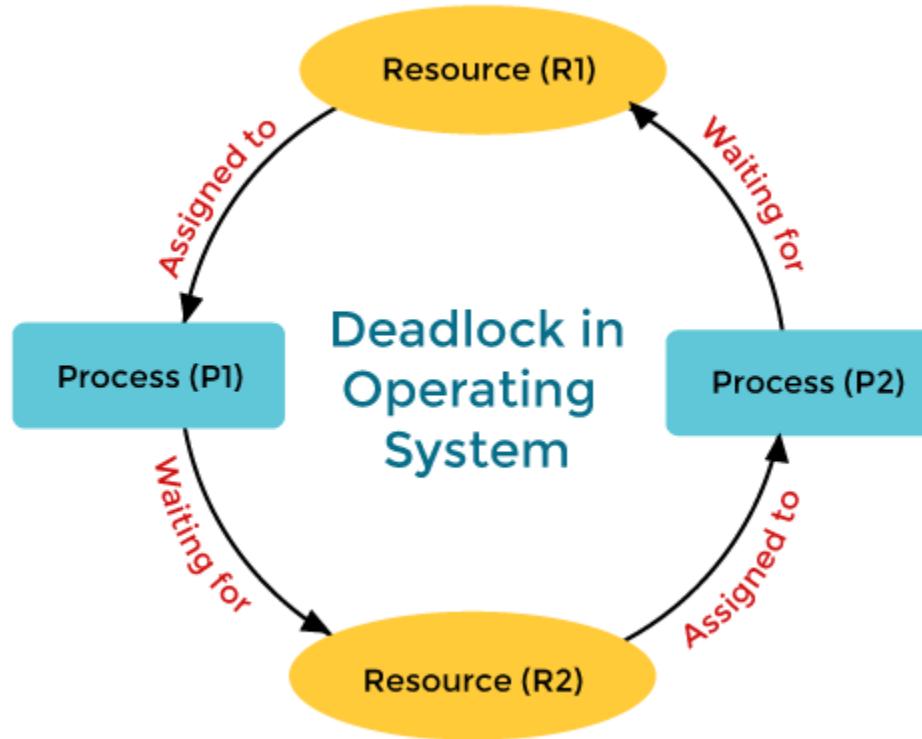
- ❖ Quando e quanto spesso richiamare l'algoritmo di rilevamento dipende da:
 - Frequenza (presunta) con la quale si verificano i deadlock
 - Numero di thread che vengono eventualmente influenzati dal deadlock (e sui quali occorre effettuare un roll back)
 - ▶ Almeno un thread per ciascun ciclo
- ❖ Se l'algoritmo viene richiamato con frequenza casuale, possono essere presenti molti cicli nel grafo delle risorse \Rightarrow non si può stabilire quale dei thread coinvolti nel ciclo abbia "causato" il deadlock
- ❖ Alternativamente, l'algoritmo può essere richiamato ogni volta che una richiesta non può essere soddisfatta immediatamente, a intervalli regolari, o quando l'utilizzo della CPU scende al di sotto del 40%

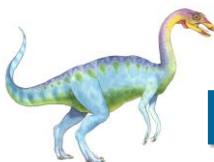




Ripristino dal deadlock

- ❖ Occorre interrompere l'attesa circolare
 - Terminare uno o più processi
 - Prelazionare una o più risorse

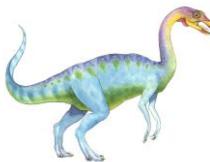




Ripristino dal deadlock: terminazione dei thread

- ❖ Terminazione in abort di tutti i thread in deadlock
- ❖ Terminazione in abort di un thread alla volta fino all'eliminazione del ciclo di deadlock
- ❖ In quale ordine si decide di terminare i thread?
- ❖ I fattori significativi sono:
 - La priorità del thread
 - Il tempo di computazione trascorso e il tempo ancora necessario al completamento del thread
 - Quantità e tipo di risorse impiegate (non/facilmente prelazionabili)
 - Risorse ulteriori necessarie al thread per terminare il proprio compito
 - Numero di thread che devono essere terminati

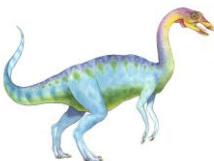




Ripristino dal deadlock: prelazione di risorse

- ❖ **Selezione di una vittima** — È necessario stabilire l'ordine di prelazione per minimizzare i costi (numero delle risorse possedute e quantità di tempo di esecuzione già utilizzato)
- ❖ **Roll back** — Un thread a cui sia stata prelazionata una risorsa deve ritornare ad uno stato sicuro, da cui ripartire
 - Per non mantenere troppe informazioni aggiuntive, la soluzione più semplice consiste nel terminare il thread
- ❖ **Starvation** — Alcuni thread possono essere sempre selezionati come vittime della prelazione: includere il numero di roll back nel fattore di costo





Esercizio 1

- ❖ Si consideri la situazione in cui vi siano in esecuzione cinque thread, T_0, T_1, T_2, T_3 e T_4 , con le matrici relative alle risorse allocate ed al numero massimo di risorse richieste dai thread definite come segue:

$$\text{Allocation} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \quad \text{Max} = \begin{pmatrix} 2 & 0 & 2 & 1 & 3 \\ 1 & 2 & 1 & 2 & 2 \\ 2 & 1 & 3 & 1 & 2 \\ 0 & 1 & 2 & 1 & 2 \\ 1 & 1 & 2 & 2 & 2 \end{pmatrix}$$

Se il vettore delle risorse disponibili è

$$\text{Available} = (1, 0, 1, 0, 1),$$

lo stato appena descritto è sicuro? Cosa accadrebbe se il thread T_0 richiedesse e ottenesse un'ulteriore istanza della risorsa del primo tipo?





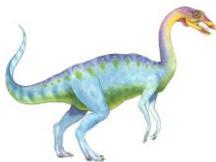
Esercizio 2

- ❖ Si supponga che cinque processi, P_0, P_1, P_2, P_3 e P_4 , condividano un insieme di risorse di quattro tipi diversi, A, B, C, D. Si supponga di trovarsi nella configurazione seguente:

	<i>Allocation</i>	<i>Max</i>	<i>Available</i>
P_0	2 4 0 6	3 X 2 7	5 12 6 8
P_1	3 0 3 5	5 7 10 6	
P_2	2 10 2 1	6 12 10 12	
P_3	6 3 3 0	X Y 5 4	
P_4	1 3 2 8	2 6 10 Z	

- ❖ Determinare quali sono le risorse massime che ciascun processo può richiedere tali che sia comunque garantita l'assenza di stallo a fronte dell'ulteriore richiesta di P_2 per (3, 2, 2, 7).





Esercizio 3

- ❖ Sia dato il sistema descritto da T insieme dei thread, R insieme delle risorse R_{ij} di indice i e molteplicità j , $E(T)$ insieme delle richieste di accesso a risorse in R emesse da thread in T e **attualmente pendenti**, e $E(R)$ insieme degli accessi **attualmente soddisfatti** di thread in T a risorse in R :

$$T = \{T_1, T_2, T_3, T_4, T_5, T_6\}$$

$$R = \{R_{12}, R_{21}, R_{32}, R_{41}\}$$

$$E(T) = \{T_2 \rightarrow R_2, T_3 \rightarrow R_4, T_4 \rightarrow R_1, T_5 \rightarrow R_4, T_6 \rightarrow R_3\}$$

$$E(R) = \{R_1 \rightarrow (T_1, T_2), R_2 \rightarrow T_3, R_3 \rightarrow (T_4, T_5), R_4 \rightarrow T_6\}$$

- ❖ Si analizzi il grafo di allocazione delle risorse, determinando se il sistema si trovi attualmente in situazione di stallo o meno
- ❖ Si studi l'evoluzione successiva dello stato del sistema se il thread T_1 richiedesse la risorsa R_2



Fine del Capitolo 8

