Stallo (o deadlock)

Obiettivi

- Illustrare come può verificarsi un deadlock
- Definire le quattro condizioni necessarie che caratterizzano il deadlock (sulle risorse)
- Identificare una situazione di deadlock
- Descrivere i quattro diversi approcci per gestire i deadlock
- Applicare l'algoritmo del banchiere per evitare deadlock
- Applicare l'algoritmo di rilevamento dei deadlock
- Valutare gli approcci per il ripristino da una situazione di deadlock

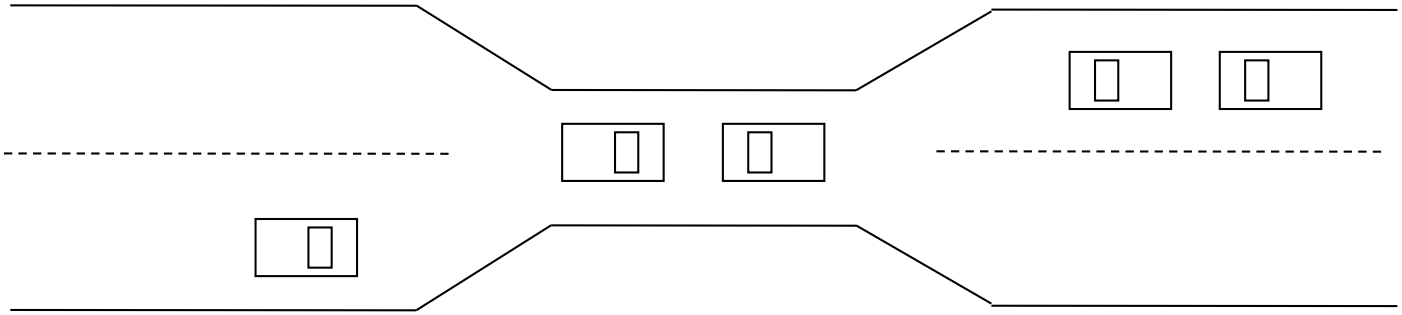
Stallo (o deadlock)

- Il problema del deadlock
- Modello delle risorse di un sistema
- Caratterizzazione del deadlock
- Metodi di gestione dei deadlock
  - Prevenire il deadlock
  - Evitare il deadlock
  - Rilevare il deadlock e ripristinare il sistema
  - Ignorare il deadlock

Stallo (o deadlock)

- Il problema del deadlock
- Modello delle risorse di un sistema
- Caratterizzazione del deadlock
- Metodi di gestione dei deadlock
  - Prevenire il deadlock
  - Evitare il deadlock
  - Rilevare il deadlock e ripristinare il sistema
  - Ignorare il deadlock

Deadlock nella vita reale: strettoia in senso unico alternato



- Ogni lato del ponte può essere visto come una risorsa
- Per attraversare il ponte, una macchina deve impossessarsi di entrambi i lati
- Si ha deadlock quando le macchine provenienti da un lato attendono indefinitamente che quelle provenienti dall'altro lato lascino libera la loro parte del ponte, e viceversa
- Se si verifica un deadlock, lo si può rimuovere facendo indietreggiare una macchina (prelazione della risorsa), ma può essere necessario fare indietreggiare molte macchine
- È possibile che si verifichi starvation (le macchine provenienti da una direzione non riescono mai ad attraversare)

Problema dei filosofi a cena (Dijkstra)

Classico problema di controllo della concorrenza

- Cinque filosofi passano la vita pensando e mangiando
- Essi condividono una tavola rotonda circondata da cinque sedie, una per filosofo
- Ogni filosofo ha un proprio piatto di spaghetti e la tavola è apparecchiata con cinque forchette
- Quando un filosofo ha fame, per poter mangiare deve prendere, una per volta, le forchette alla sua destra e alla sua sinistra … se non sono in mano del vicino!

Necessità di allocare varie risorse a diversi processi evitando deadlock e starvation

Problema dei filosofi a cena

Problema dei filosofi a cena

- L'attività di ciascun filosofo può essere descritta dal seguente processo

repeat
pensa
impossessati delle due forchette
mangia
rilascia le due forchette
until *false*

- Una soluzione semplice consiste nel rappresentare ogni forchetta con un semaforo di mutua esclusione

Problema dei filosofi a cena

- Dati condivisi

  var *fork*: **array** [0..4] **of** *semaphore*;
  (ogni elemento dell'array è inizializzato a 1)

- Filosofo *i*

  repeat
  [pensa]
  wait(*fork*[*i*])
  wait(*fork*[*i*+1 mod 5])
  [mangia]
  signal(*fork*[*i*]);
  signal(*fork*[*i*+1 mod 5]);
  until *false*;

Problema dei filosofi a cena

- Questa soluzione garantisce che due filosofi vicini non mangino contemporaneamente
- Non esclude la possibilità che si crei un deadlock
  - tutti i filosofi hanno fame contemporaneamente e tentano di afferrare la forchetta di sinistra
  - tutti gli elementi dell'array *fork* diventano uguali a zero
  - quando un filosofo tenta di afferrare la forchetta di destra viene messo in attesa (per sempre!)
- Sono state proposte diverse possibili soluzioni per evitare il deadlock in questo scenario, quali ad esempio
  - ordinare le forchette e richiederle rispettando l'ordinamento
  - chiedere entrambe le forchette con un'unica richiesta
  - ammettere al tavolo solo 4 filosofi

Deadlock: considerazioni generali

- Il deadlock è un comportamento emergente, che si può presentare, anche solo in alcune esecuzioni, a causa dello scheduling e dell'interleaving di processi concorrenti
- Se l'accesso in mutua esclusione a ciascuna risorsa è gestito tramite sincronizzazione su un corrispondente semaforo, allora l'interazione tra i processi può causare un deadlock, anche se ogni singolo processo è programmato correttamente
- Nella soluzione presentata al problema dei filosofi a cena, il deadlock è dovuto al fatto che tutti i filosofi prendono le forchette nello stesso ordine: prima quella di sinistra, poi quella di destra
  - Una soluzione soddisfacente consiste nel far sì che uno dei filosofi prenda le forchette nell'ordine opposto
  - Questo accorgimento, però, non funziona in generale
- Nell'esempio successivo, il deadlock è dovuto al fatto che i due processi richiedono le risorse non nello stesso ordine!

Deadlock in un sistema di elaborazione

- Un sistema ha 2 masterizzatori di cd
- I processi P_1 e P_2 vogliono entrambi effettuare una copia di un CD e per farlo hanno bisogno di ottenere entrambi i masterizzatori (o meglio, i semafori di mutua esclusione corrispondenti)
- Semafori A and B (per la mutua esclusione), inizializzati ad 1

P_1	P_2
wait(A);	**wait**(B);
wait(B);	**wait**(A);
<uso di A e B>;	<uso di A e B>;
signal(B);	**signal** (A);
signal(A);	**signal** (B);

- Situazione critica che si può presentare: P_1 e P_2 possiedono un masterizzatore ciascuno ed attendono di ottenere l'altro

Deadlock (o stallo)

- Un insieme di processi di un sistema è in stato di deadlock (o stallo) se ciascun processo dell'insieme è in attesa di un evento che solo un altro processo dell'insieme può provocare
  - Sincronizzazione tra processi: i processi attendono segnali uno dall'altro
  - Comunicazione tra processi: i processi attendono messaggi uno dall'altro
  - Condivisione di risorse: i processi attendono che altri processi rilascino le risorse di cui hanno bisogno
- Lo stallo sulle risorse riguarda il SO perché la gestione delle risorse è compito del SO, le altre due forme di deadlock non sono gestite dal SO (ma dalle applicazioni)
- Lo stallo è una situazione critica che va evitata:
  - oltre ad impedire l'avanzamento dei processi in stato di stallo (quindi pregiudica servizi per l'utente e produttività del sistema)
  - sottrae definitivamente al sistema e, quindi, alla possibilità di utilizzo da parte degli altri processi, le risorse già assegnate ai processi in stallo
  - degradando alla lunga le prestazioni del sistema (quindi pregiudica l'uso efficiente delle risorse)

Livelock (o stallo attivo)

- È una forma di mancanza di liveness (come il deadlock)
- È meno comune del deadlock, ma è comunque un problema complesso nella progettazione di applicazioni concorrenti e, come il deadlock, può verificarsi solo in determinate condizioni di scheduling
- Si verifica quando un processo tenta continuamente un'azione che non ha successo
- In generale, quindi, può essere evitato facendo sì che ogni processo riprovi l'operazione fallita in momenti casuali
- Questo è proprio l'approccio adottato dalle reti Ethernet quando si verifica una collisione di rete
  - Invece di provare a ritrasmettere un pacchetto immediatamente dopo che una collisione si è verificata, un dispositivo coinvolto nella collisione aspetterà un periodo di tempo casuale prima di tentare di trasmettere di nuovo

Stallo (o deadlock)

- Il problema del deadlock
- Modello delle risorse di un sistema
- Caratterizzazione del deadlock
- Metodi di gestione dei deadlock
  - Prevenire il deadlock
  - Evitare il deadlock
  - Rilevare il deadlock e ripristinare il sistema
  - Ignorare il deadlock

Risorse di un sistema

- Nel modello concorrente, un sistema è composto da un numero finito di risorse distribuite tra più processi, in competizione per il loro uso
- Per risorsa si intende qualsiasi entità HW o SW che appartenga al sistema e possa essere usata per l'esecuzione dei processi
  - risorse fisiche: CPU, stampanti, terminali, partizioni di memoria, ...
  - risorse logiche: file, semafori, PCB, ...
- Le risorse sono suddivise in tipi (o classi) differenti ciascuno formato da un certo numero di risorse omogenee, dette istanze
  - Un processo richiede una risorsa di un certo tipo, **non** una risorsa specifica

Proprietà delle risorse

- Condivisibili: possono essere usate simultaneamente da più processi (es. file aperto in sola lettura)
- Non-condivisibili o seriali: devono essere usate in maniera mutuamente esclusiva (es. ciclo di CPU)
- Statiche: sono assegnate ad un processo dalla sua creazione fino alla sua terminazione (es. PCB)
- Dinamiche: sono assegnate ad un processo durante la sua esecuzione (es. aree di memoria principale)
- Riusabili: possono essere date ripetutamente in uso ai processi (es. CPU, stampanti, file)
- Consumabili: possono essere usate una sola volta (es. messaggi, segnali, interruzioni)
- Prelazionabili: il SO può sottrarle al processo assegnatario prima che questo le rilasci spontaneamente (es. CPU)
- Non-prelazionabili: non possono essere riassegnate finché il processo assegnatario non le rilascia spontaneamente (es. file descriptor)

Modello delle risorse (e dei processi)

- Tipi (o classi) di risorse $R_1, R_2, \ldots, R_m$
- Ogni tipo di risorsa R_i ha un certo numero di istanze omogenee
- Per utilizzare una risorsa, un processo deve effettuare il seguente ciclo di operazioni (nell'ordine specificato):
  - richiesta
  - uso
  - rilascio

Gestione delle risorse di un sistema

- La gestione delle risorse compete al SO
  - Il gestore di una risorsa è un modulo SW in grado di ricevere e soddisfare richieste d'uso e rilascio della risorsa da parte dei processi
  - Richiesta e rilascio avvengono tramite system call
    es. `open()` e `close()`, `allocate()` e `free()`
- Ogni volta che un processo usa una risorsa, il SO controlla che ne abbia fatto richiesta e che la risorsa gli sia stata assegnata
- Una tabella delle risorse di sistema registra lo stato di ogni risorsa e, se è assegnata, indica il processo a cui è assegnata
- Se un processo richiede una risorsa già assegnata, viene accodato agli altri processi eventualmente in attesa di quella risorsa

Gestione delle risorse di un sistema

- Genera tre tipi di eventi:
  - richiesta della risorsa
  - allocazione della risorsa
  - rilascio della risorsa
- Si verifica un evento di richiesta quando qualche processo P_i effettua una richiesta di una risorsa r_l
  - Un evento di richiesta da parte di P_i può causare l'evento di allocazione di r_l: nel qual caso, P_i diventerà **possessore** di r_l
  - P_i sarà bloccato sull'evento di allocazione di r_l se invece r_l è già allocata ad un processo P_k: P_i è **in attesa** che P_k rilasci r_l
- Un evento di rilascio di r_l da parte di P_k libera la risorsa ed il SO può decidere di allocare r_l ad un processo P_i in attesa
  - Un evento di rilascio da parte di P_k può quindi anch'esso causare l'evento di allocazione di cui P_i è in attesa
- Il processo richiedente P_i andrà incontro a starvation se il rilascio di r_l da parte di P_k viene ritardato indefinitamente

Gestione delle risorse di un sistema

- Il SO prende due tipi di decisioni riguardo la gestione delle risorse con l'obiettivo di usarle nel modo più efficiente possibile
  - allocazione: dato un insieme di richieste di risorse, a quali processi bisogna assegnare quali risorse?
  - scheduling: quando ci sono più richieste che risorse disponibili, in quale ordine devono essere servite le richieste?
- Scopi dello scheduling di una risorsa
  - garantire che prima o poi i processi richiedenti ottengano la risorsa (evitare starvation)
  - minimizzare il tempo medio di attesa della risorsa
  - minimizzare la variabilità dei tempi di attesa della risorsa
  - rispettare le priorità dei processi
- Esempi: scheduling della CPU (un processore, molti processi), scheduling delle richieste al disco, ...

Stallo (o deadlock)

- Il problema del deadlock
- Modello delle risorse di un sistema
- Caratterizzazione del deadlock
- Metodi di gestione dei deadlock
  - Prevenire il deadlock
  - Evitare il deadlock
  - Rilevare il deadlock e ripristinare il sistema
  - Ignorare il deadlock

Caratterizzazione dello stallo

Una situazione di deadlock (sulle risorse) si può verificare solo se si presentano simultaneamente le quattro condizioni seguenti:

- Mutua esclusione: le risorse possono essere usate soltanto da un processo alla volta (cioè le risorse sono seriali)
- Possesso e attesa: i processi trattengono le risorse che già posseggono e richiedono risorse addizionali
- Assenza di prelazione: le risorse già assegnate ai processi non possono essere a questi sottratte (cioè possono essere rilasciate soltanto volontariamente dal processo che le possiede)
- Attesa circolare: esiste un insieme di processi $\{P_0, \ldots, P_n\}$ in attesa di risorse, tali che
  - P_0 sta aspettando una risorsa che è posseduta da P_1,
  - …,
  - P_{n-1} sta aspettando una risorsa che è posseduta da P_n, e
  - P_n sta aspettando una risorsa che è posseduta da P_0

Le condizioni sono necessarie, non sufficenti, affinché lo stallo si verifichi

Modelli dello stato di allocazione delle risorse

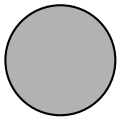
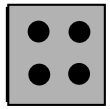
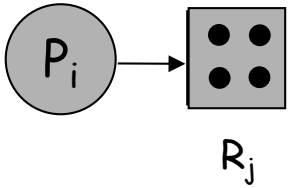
- Per stabilire se in un sistema è presente, o potrà verificarsi, una situazione di deadlock è necessario analizzare le informazioni relative
  - alle risorse allocate ai processi
  - alle richieste di risorse in attesa
- Tutte queste informazioni costituiscono lo stato di allocazione (delle risorse) di un sistema
- Per rappresentare tale stato si usano due tipi di modelli
  - basati su grafo
  - basati su matrice

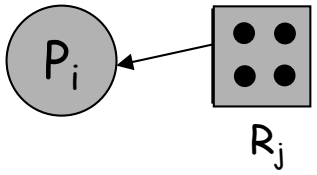
Grafo di allocazione delle risorse (di Holt)

- Grafo (V, E): rappresenta graficamente lo stato di allocazione del sistema
  - V, l'insieme dei nodi, è suddiviso in due tipi di nodi:
    - $P = \{P_1, P_2, \ldots, P_n\}$: insieme di tutti i processi del sistema
    - $R = \{R_1, R_2, \ldots, R_m\}$: insieme di tutti i tipi di risorsa del sistema
  - E, l'insieme degli archi orientati, è suddiviso in due tipi di archi:
    - arco di allocazione: arco orientato $R_j \rightarrow P_i$
    - arco di richiesta: arco orientato $P_i \rightarrow R_j$

- Archi nel grafo di Holt aggiunti e rimossi in base al verificarsi di eventi connessi alla gestione delle risorse
  - P_i richiede un'unità di risorsa del tipo R_j : inserito $P_i \rightarrow R_j$
  - Un'unità di risorsa del tipo R_j è allocata a P_i : $P_i \rightarrow R_j$ è rimpiazzato da $R_j \rightarrow P_i$
  - P_i rilascia un'unità di risorsa del tipo R_j : $R_j \rightarrow P_i$ è rimosso

Grafo di allocazione delle risorse (o di Holt)

- Processo 
- Tipo di risorsa con 4 istanze 
- P_i richiede una istanza di R_j 

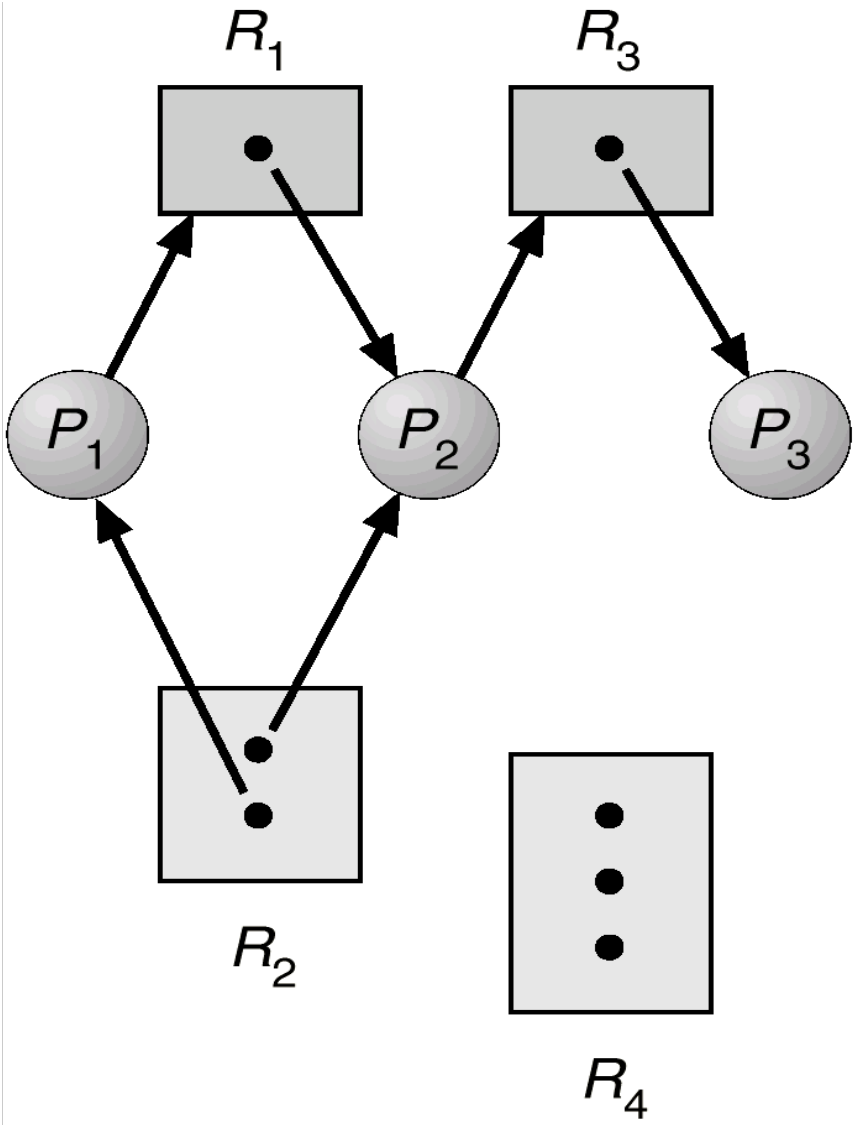
- P_i possiede una istanza di R_j 

Esempio di grafo di allocazione delle risorse

Insiemi P, R ed E:
$P = \{ P_1, P_2, P_3 \}$
$R = \{ R_1, R_2, R_3, R_4 \}$
$E = \{ P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2,$
$R_2 \rightarrow P_1, R_2 \rightarrow P_2, R_3 \rightarrow P_3 \}$

Istanze delle risorse:
1:R1, 2:R2, 1:R3, 3:R4

Stato dei processi:
P_1 possiede una istanza di tipo R_2
ed attende una istanza di tipo R_1;
P_2 possiede una istanza di tipo R_1 ed R_2
ed attende una istanza di tipo R_3;
P_3 possiede una istanza di tipo R_3.



Grafo di Holt e deadlock

Valgono le seguenti proprietà

- Se il grafo di allocazione delle risorse **non** contiene cicli
  ⇒
  non si verifica alcun deadlock

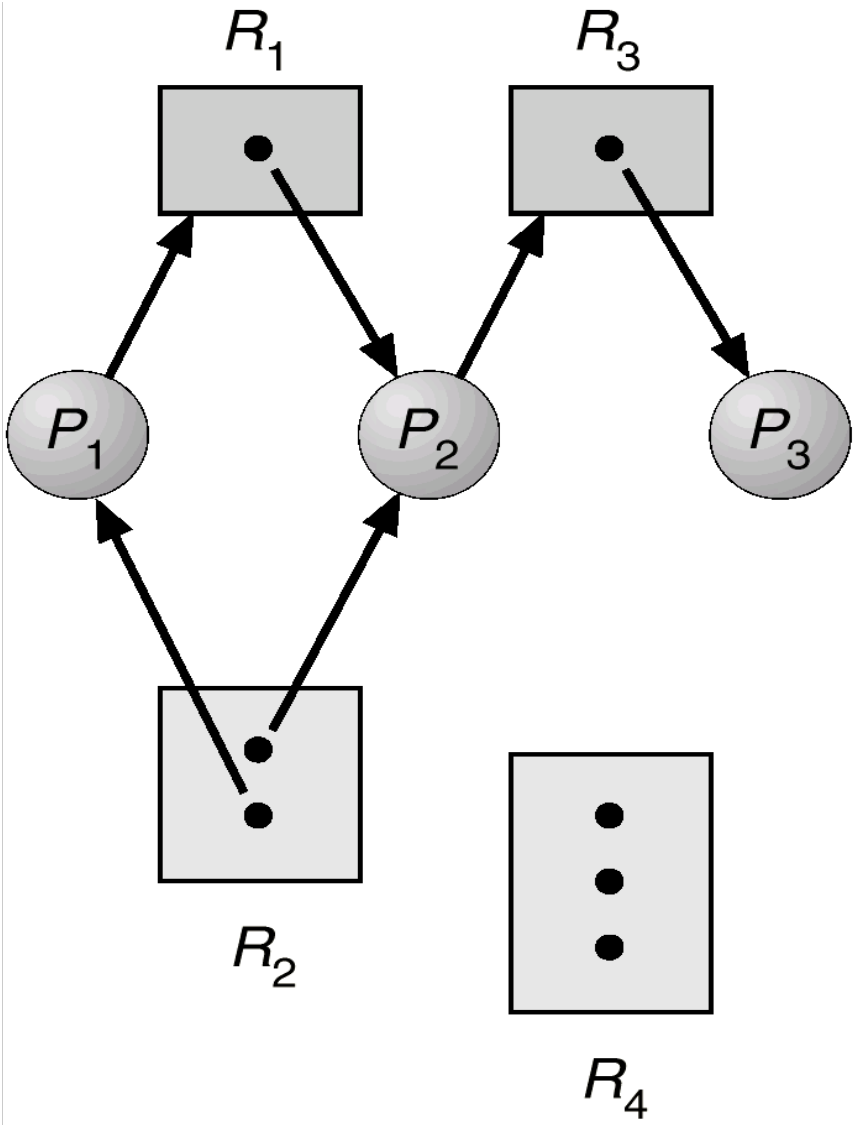
- Se il grafo di allocazione delle risorse contiene (**almeno**) un ciclo
  ⇒
  - se ogni tipo di risorsa coinvolto nel ciclo ha una sola istanza,
    allora si verifica una situazione di deadlock
  - altrimenti (esiste un tipo di risorsa che ha più istanze),
    c'è la possibilità che si verifichi un deadlock
    cioè, la condizione è necessaria ma non sufficiente

Esempio di grafo di allocazione delle risorse

Non ci sono cicli

⇓

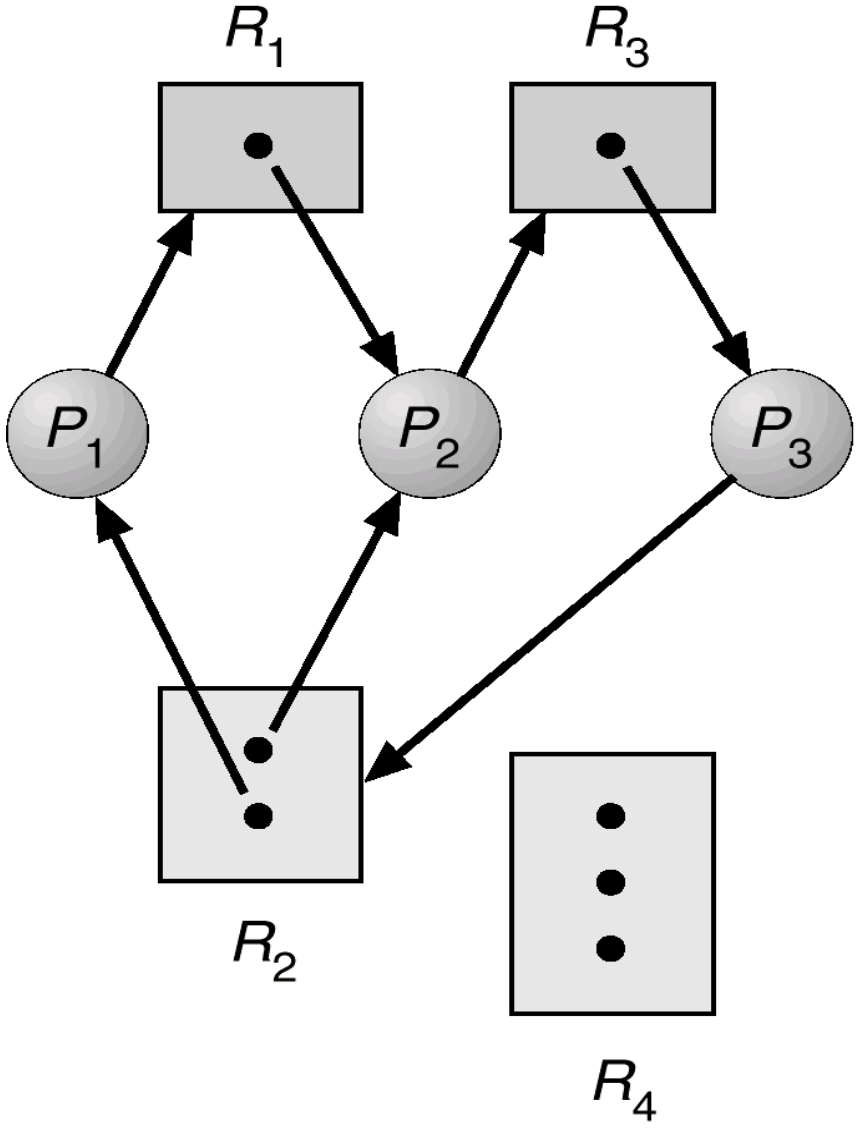
Non c'è deadlock



Esempio di grafo di allocazione delle risorse

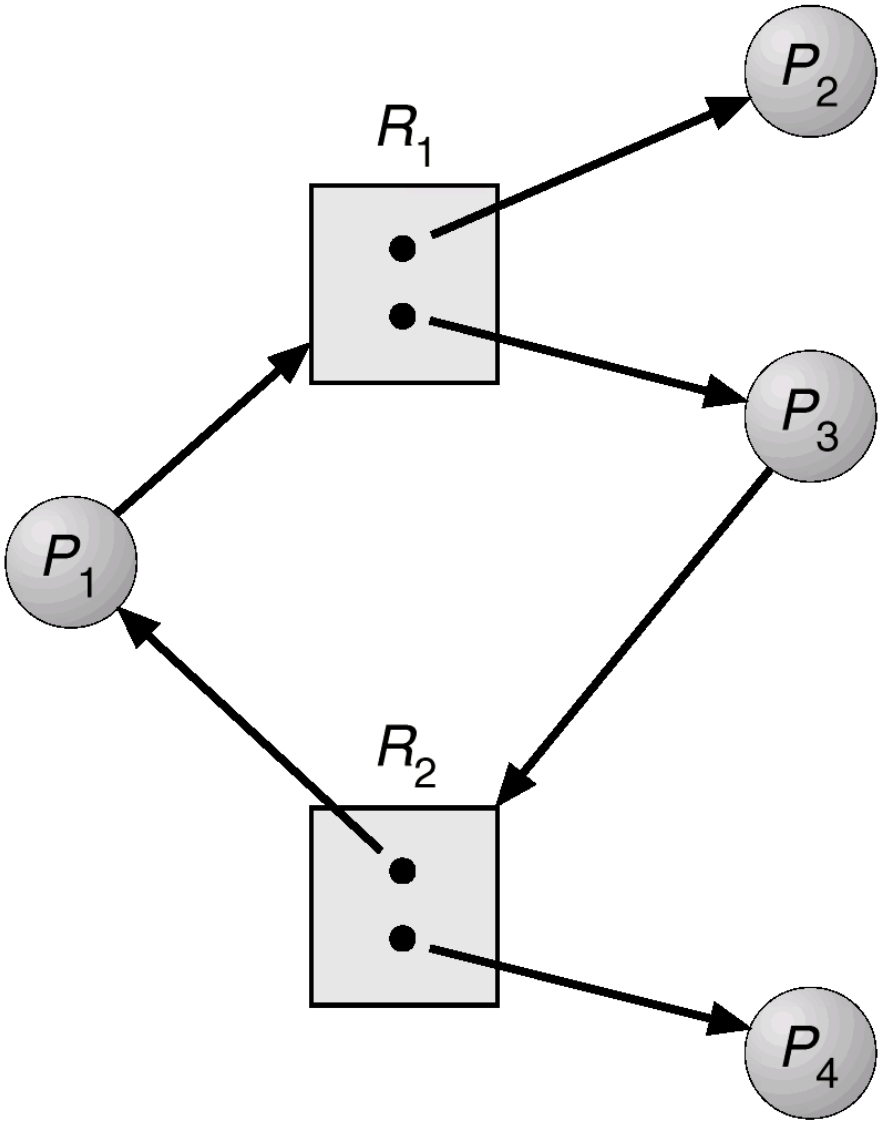
Aggiunto l'arco $P_3 \rightarrow R_2$

Ci sono cicli,
c'è deadlock



Esempio di grafo di allocazione delle risorse

C'è il ciclo
$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
però non si ha stallo:
il processo P_4 può rilasciare la propria istanza del tipo di risorsa R_2, che si può assegnare al processo P_3, rompendo il ciclo



Ci sono cicli, ma non c'è deadlock

Stallo (o deadlock)

- Il problema del deadlock
- Modello delle risorse di un sistema
- Caratterizzazione del deadlock
- Metodi di gestione dei deadlock
  1. Prevenire il deadlock
  2. Evitare il deadlock
  3. Rilevare il deadlock e ripristinare il sistema
  4. Ignorare il deadlock

Metodi di gestione dei deadlock

1. Situazioni di stallo non si possono verificare
   - Prevenire i deadlock: all'atto della scrittura dei programmi, vengono utilizzati dei metodi (es. assegnazione ordinata delle risorse) che vincolano il comportamento dei processi ed impediscono che si verifichino simultaneamente tutte le 4 condizioni necessarie per il deadlock
   - Evitare i deadlock: il SO riceve dai processi informazioni sulle risorse che essi intendono richiedere durante la loro attività ed utilizza una politica di gestione che evita lo stallo (es. algoritmo del banchiere)
2. Situazioni di stallo si possono verificare
   - Rilevare e ripristinare: quando il SO rileva uno stallo, interviene e recupera uno stato corretto precedente allo stallo
   - Ignorare del tutto il problema e assumere che i deadlock non si presentino mai

Stallo (o deadlock)

- Il problema del deadlock
- Modello delle risorse di un sistema
- Caratterizzazione del deadlock
- Metodi di gestione dei deadlock
  1. Prevenire il deadlock
  2. Evitare il deadlock
  3. Rilevare il deadlock e ripristinare il sistema
  4. Ignorare il deadlock

Prevenire i deadlock

Prevenzione statica

- Realizzata all'atto della scrittura dei programmi
- Imporre dei vincoli sul comportamento dei processi per far sì che almeno una delle condizioni necessarie per il deadlock non possa verificarsi

Mutua esclusione

Non è necessaria per le risorse condivisibili, mentre è inevitabile per le risorse seriali

- Se tutte le risorse potessero essere condivise, non ci sarebbero relazioni di attesa (niente cicli nel grafo di allocazione delle risorse)
  - File aperti in sola lettura
- La virtualizzazione delle risorse HW può supportare la condivisione
- Ma alcune risorse sono intrisecamente non condivisibili e in numero volutamente limitato
  - I lock mutex per la sincronizzazione
  - File condivisi, ed altre risorse SW, dovrebbero comunque essere modificati in mutua esclusione per evitare inconsistenze

Inconvenienti:

- Impedire la mutua esclusione è difficilmente realizzabile in pratica

Possesso ed attesa

Ogni volta che un processo richiede una risorsa, non deve già possederne altre; due possibili strategie:

1. Ogni processo deve chiedere l'assegnazione di tutte le risorse di cui ha bisogno con un'unica richiesta (allocazione globale)
2. Un processo può chiedere risorse solo se non ne possiede già altre (per fare una richiesta, deve eventualmente fare prima un rilascio delle risorse già assegnate)

Inconvenienti:

- Scarso utilizzo delle risorse
- Difficile prevedere in anticipo tutte le risorse necessarie
- Possibilità di attesa indefinita (se si richiedono risorse molto utilizzate)
- Pregiudica l'efficienza d'uso delle risorse e riduce il grado di multiprogrammazione, e quindi la produttività

Assenza di prelazione

Per ovviare all'assenza di prelazione, si potrebbe usare il seguente protocollo (sequenza di azioni e interazioni che deve essere implementata dagli stessi programmi):

- Se un processo richiede risorse che non possono essergli assegnate subito e ne possiede già altre, allora il processo rilascia implicitamente tutte le risorse attualmente possedute
  - Le risorse rilasciate anticipatamente vengono aggiunte alla lista delle risorse per cui il processo sta aspettando
  - Il processo potrà riprendere l'esecuzione soltanto quando potrà ottenere sia le risorse già possedute che quelle la cui richiesta ne ha causato la sospensione

Assenza di prelazione

La virtualizzazione delle risorse può venire in aiuto

- Ad esempio, se ad un processo è stata assegnata la stampante ed è in fase di stampa del suo output, rimuovere forzatamente la stampante per assegnarla ad un processo nel migliore dei casi è complicato, e nel peggiore è impossibile
- Per ovviare all'assenza di prelazione della stampante si potrebbe usare il disco per virtualizzare la stampante, effettuando così lo spooling dell'output della stampante sul disco e consentendo solo al demone della stampante l'accesso alla stampante reale
- Crea un potenziale deadlock sullo spazio su disco dedicato alla virtualizzazione, ma con dischi di grandi dimensioni è improbabile che lo spazio su disco si esaurisca

Non tutte le risorse possono essere virtualizzate

- Ad esempio, per modificare i record di un database condiviso o le tabelle all'interno del SO bisogna prima ottenere il relativo lock
- Qui risiede il rischio potenziale del deadlock perché i lock sono volutamente in numero limitato e non prelazionabili

Assenza di prelazione

Inconvenienti:

- Il prerilascio di risorse è un'operazione complicata
- Adatto solo a risorse il cui stato si può salvare e recuperare in un secondo tempo, es. CPU
- Non adatto a quei tipi di risorse che più comunemente generano situazioni di stallo, es. lock mutex e semafori

Attesa circolare

Definire una relazione di ordinamento tra tutti i tipi di risorsa e imporre che ogni processo debba rispettarla nel richiedere le risorse

- Ad ogni tipo di risorsa R_k si assegna un numero intero unico tramite una funzione F: R → $\mathbb{N}$
- Sia $F(R_k)$ l'intero assegnato al tipo R_k per ogni k: se $F(R_i) < F(R_j)$, allora le risorse di tipo R_i vanno richieste prima di richiedere risorse di tipo R_j
- Equivalentemente, un processo **non** può richiedere una risorsa di R_i se già possiede risorse di R_j con $F(R_i) < F(R_j)$
- Se il processo necessita di più istanze dello stesso tipo, deve presentare un'unica richiesta per tutte le istanze

Attesa circolare

In un sistema che utilizza l'ordinamento delle risorse, l'assenza di circolarità nelle relazioni di attesa può essere dimostrata ragionando per assurdo

- Supponiamo ci sia un'attesa circolare malgrado che i processi rispettino la politica basata sull'ordinamento nell'effettuare richieste di risorse
- Sia $\{P_0, P_1, \ldots, P_n\}$ l'insieme dei processi coinvolti nell'attesa circolare, dove il processo P_i attende una risorsa di tipo $R_{f(i)}$ posseduta dal processo P_{i+1} (sugli indici si usa l'aritmetica modulare)
- Poiché il processo P_{i+1} possiede una risorsa di $R_{f(i)}$ mentre richiede una risorsa di $R_{f(i+1)}$ (poiché è a sua volta in attesa del processo P_{i+2}) è necessario che sia verificata la condizione $F(R_{f(i)}) < F(R_{f(i+1)})$, per tutti gli indici i
- Ciò implica che $F(R_{f(0)}) < F(R_{f(1)}) < \ldots < F(R_{f(n)}) < F(R_{f(0)})$
- Per la proprietà transitiva dell'ordinamento risulta quindi che $F(R_{f(0)}) < F(R_{f(0)})$, il che è impossibile
- Quindi, non può esservi attesa circolare

Attesa circolare

Inconvenienti:

- Stabilire un ordinamento tra i tipi di risorse che possa andar bene a tutti i processi può essere difficile, specialmente in un sistema con centinaia o migliaia di risorse (e relativi lock mutex)

È comunque il metodo di prevenzione del deadlock più comunemente utilizzato

Prevenire i deadlock

Interpretazione grafica dei vincoli di prevenzione

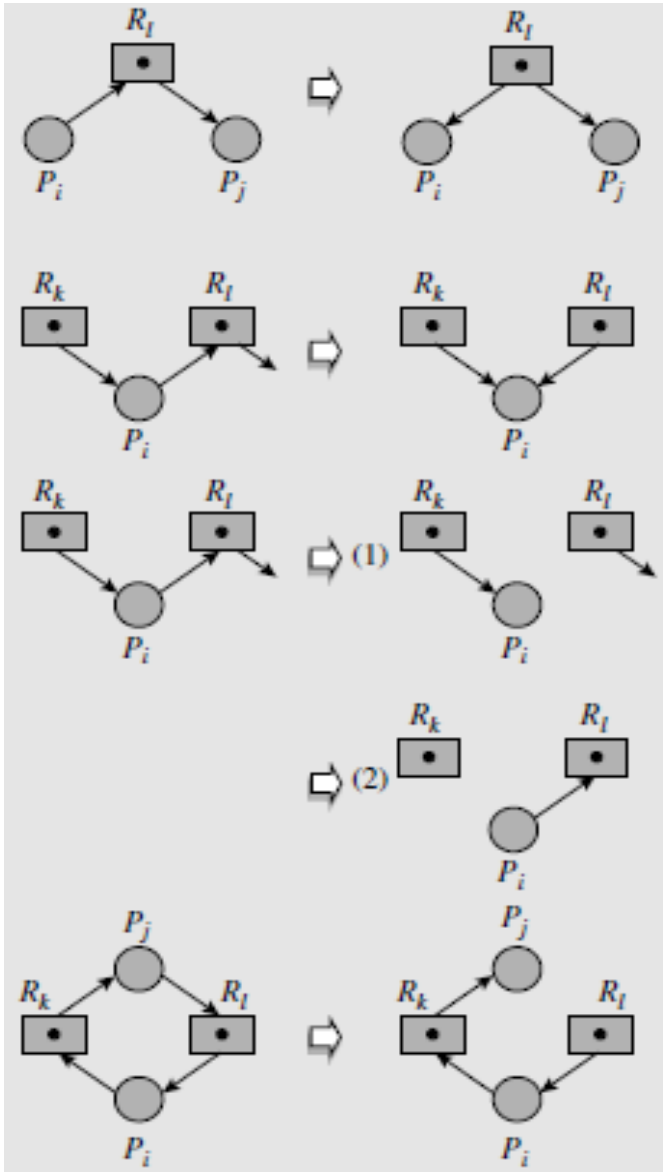
Rendere le risorse condivisibili (nessuna attesa): il processo P_i non si blocca sulla richiesta della risorsa R_l

Rendere le risorse prelazionabili (nessuna attesa): la risorsa R_l viene prelazionata ed allocata a P_i

Prevenire possesso e attesa (nessun percorso nel grafo di allocazione con più di un processo):

al processo P_i (1) non è consentito chiedere la risorsa R_l, o (2) non è consentito mantenere R_k mentre chiede R_l

Prevenire attese circolari: se $F(R_l) < F(R_k)$, al processo P_j non è permesso richiedere la risorsa R_l se possiede già R_k



Prevenire i deadlock: valutazione

- Prevenzione statica
  - Imporre vincoli sul comportamento futuro dei processi al momento in cui vengono scritti i programmi
  - Tra le 4 condizioni, impedire che si verifichi l'attesa circolare è l'approccio più usato in pratica
- Inconvenienti
  - Può causare scarso utilizzo delle risorse e quindi ridotta produttività del sistema
  - La responsabilità di implementare i vincoli all'interno dei programmi è a carico dei programmatori di applicazioni
    - Il SO non ha infatti controllo sulle sequenze di istruzioni previste dai programmi che i processi eseguono
    - Anche perché tipicamente il SO non ha accesso ai sorgenti

Stallo (o deadlock)

- Il problema del deadlock
- Modello delle risorse di un sistema
- Caratterizzazione del deadlock
- Metodi di gestione dei deadlock
  1. Prevenire il deadlock
  2. Evitare il deadlock
  3. Rilevare il deadlock e ripristinare il sistema
  4. Ignorare il deadlock

Evitare il deadlock

- Prevenzione dinamica
  - Non si impongono vincoli sul comportamento dei processi
- Il SO usa algoritmi che esaminano dinamicamente lo stato di allocazione delle risorse per accertarsi che la condizione di attesa circolare non possa mai verificarsi
- Gli algoritmi necessitano di informazioni a priori su come vengono richieste le risorse dai processi
  - L'algoritmo più semplice si basa sul numero massimo di risorse di ciascun tipo di cui un processo può avere bisogno durante tutta la sua esecuzione
- Assunzione: i processi, una volta ottenute le risorse richieste, terminano rilasciando tutte le risorse loro assegnate

Stato sicuro

- Per garantire che l'allocazione delle risorse conseguente ad una richiesta non possa portare a deadlock, né immediatamente né successivamente, si usa la nozione di stato (di allocazione) sicuro
- Uno stato è sicuro se c'è una sequenza ordinata con cui il sistema può allocare le risorse ad ogni processo (fino alle sue massime richieste) ed evitare che si presentino deadlock
- Un sistema è in uno stato sicuro soltanto se esiste una sequenza sicura di allocazione delle risorse ai processi
- La sequenza $<P_1, P_2, \ldots, P_n>$ è sicura se, per ogni P_i, le richieste di risorse che P_i può effettuare possono essere soddifatte usando le risorse attualmente disponibili e le risorse possedute da tutti i processi P_j, con $j < i$
  - Così facendo, se le risorse di cui P_i ha bisogno non fossero disponibili immediatamente, allora P_i dovrà aspettare tutt'al più finché tutti i processo P_j, con $j < i$, abbiano finito
  - A quel punto, P_i potrà ottenere tutte le risorse necessarie, completare la propria esecuzione, restituire le risorse assegnate e terminare
  - Quando P_i sarà terminato, P_{i+1} potrà ottenere le risorse necessarie e così via …

Stato sicuro, non sicuro e di stallo

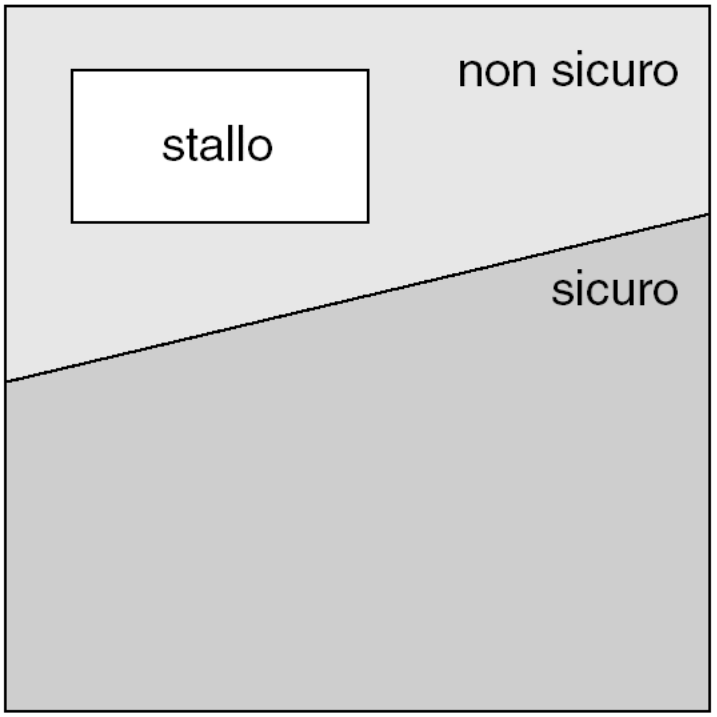
Proprietà

- Se il sistema è in uno stato sicuro ⇒ nessun deadlock si potrà presentare
- Se il sistema è in uno stato non sicuro ⇒ c'è la possibilità che si prensenti un deadlock

Uno stato sicuro non è di stallo

Viceversa, uno stato di stallo è uno stato non sicuro

Tuttavia, **non** tutti gli stati non sicuri sono stati di stallo



Strategia

Per evitare il deadlock ⇒

nell'assegnare risorse ai processi sulla base delle loro richieste, è sufficente che il SO si assicuri che così facendo il sistema non entri mai in uno stato non sicuro

- Inizialmente, il sistema è in uno stato sicuro
- Ogni volta che un processo richiede una risorsa correntemente disponibile, il SO deve decidere se la risorsa può essere allocata immediatamente o se il processo deve attendere
  - Il SO accorda la richiesta solo se l'allocazione lascia il sistema in uno stato sicuro

La proprietà stato sicuro è una invariante del sistema

Evitare il deadlock: valutazione

- Quando un processo richiede una risorsa, anche se attualmente la risorsa è disponibile, il processo potrebbe comunque essere costretto ad attendere
- L'utilizzo delle risorse potrebbe essere inferiore a quello che si avrebbe nel caso in cui non si usasse un algoritmo per evitare i deadlock

Algoritmo basato sul grafo di allocazione delle risorse

- Applicabile solo nel caso che ogni tipo di risorsa abbia una sola istanza
- Al grafo di allocazione si aggiunge una terza tipologia di archi (rappresentati da una frecce tratteggiate):
  - arco di prenotazione $P_i \dashrightarrow R_j$
  - significa che il processo P_i durante la sua esecuzione potrà chiedere la risorsa R_j
  - tali archi rappresentano le informazioni a priori che bisogna conoscere riguardo il comportamento futuro dei processi
- Le risorse devono essere prenotate a priori nel sistema:

  cioè prima che un processo cominci la sua esecuzione, tutti i suoi archi di prenotazione devono essere inseriti nel grafo di allocazione delle risorse
  - In alternativa, un arco di prenotazione $P_i \dashrightarrow R_j$ può essere aggiunto dinamicamente ma solo fino a che P_i ha solo archi di prenotazione

Algoritmo basato sul grafo di allocazione delle risorse

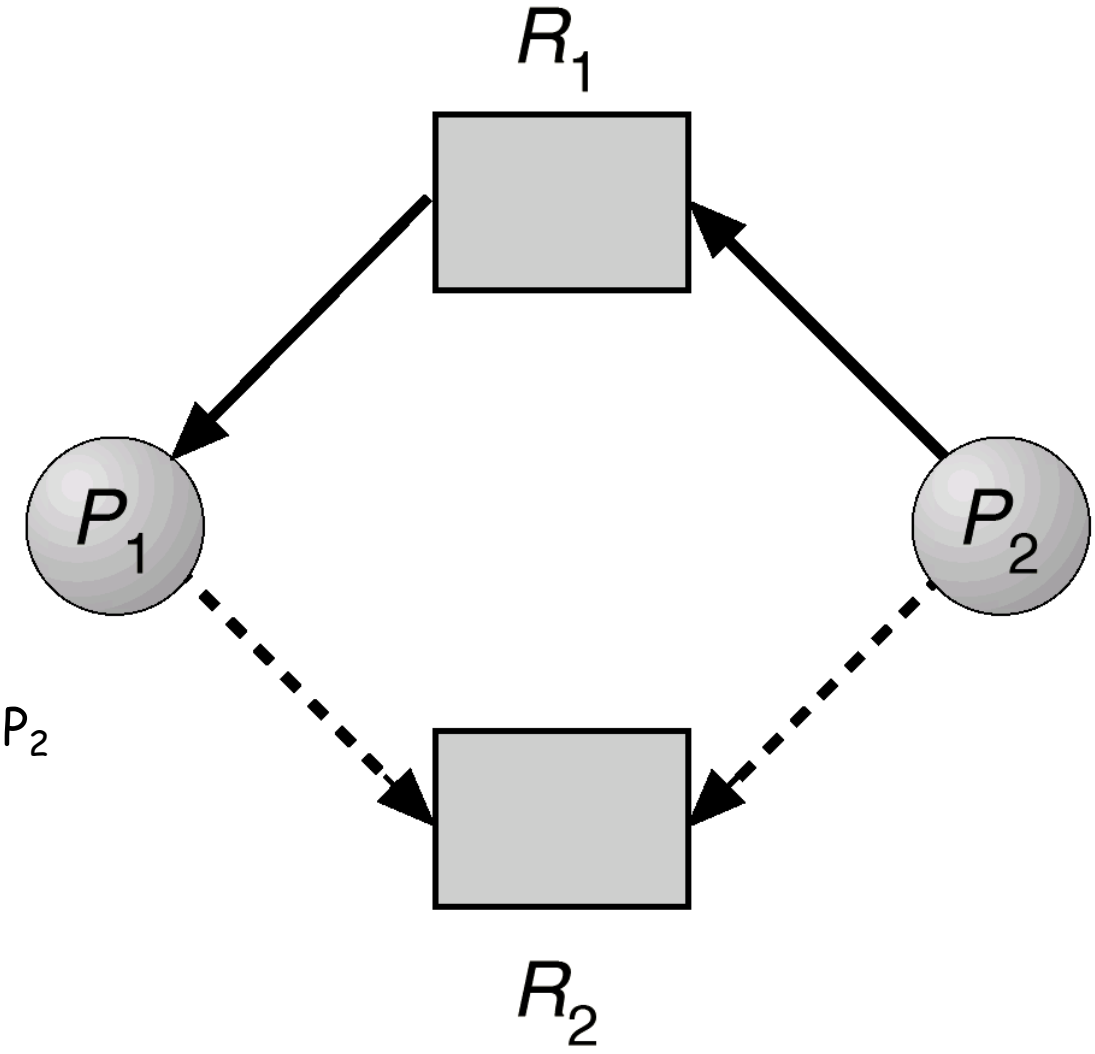
- Quando un processo richiede effettivamente una risorsa:
  - se la risorsa è disponibile, il relativo arco di prenotazione è convertito in un arco di assegnazione:
    - se ciò non crea un ciclo, l'assegnazione della risorsa ha successo poiché lascia il sistema in uno stato sicuro;
    - altrimenti, il processo deve attendere e l'arco di assegnazione è convertito in un arco di richiesta
  - altrimenti (la risorsa non è disponibile), il processo deve attendere e il relativo arco di prenotazione è convertito in un arco di richiesta
- Quando un processo rilascia una risorsa, l'arco di assegnazione è convertito in un arco di prenotazione (per tener conto che il processo potrà richiedere nuovamente la risorsa in un momento successivo)
- Il costo dell'algoritmo di ricerca di un ciclo è $O(n^2)$ (dove n è il numero dei nodi)

Grafo di allocazione delle risorse per evitare i deadlock

Supponiamo che P_2 richiede R_2

Se si assegna R_2 al processo P_2

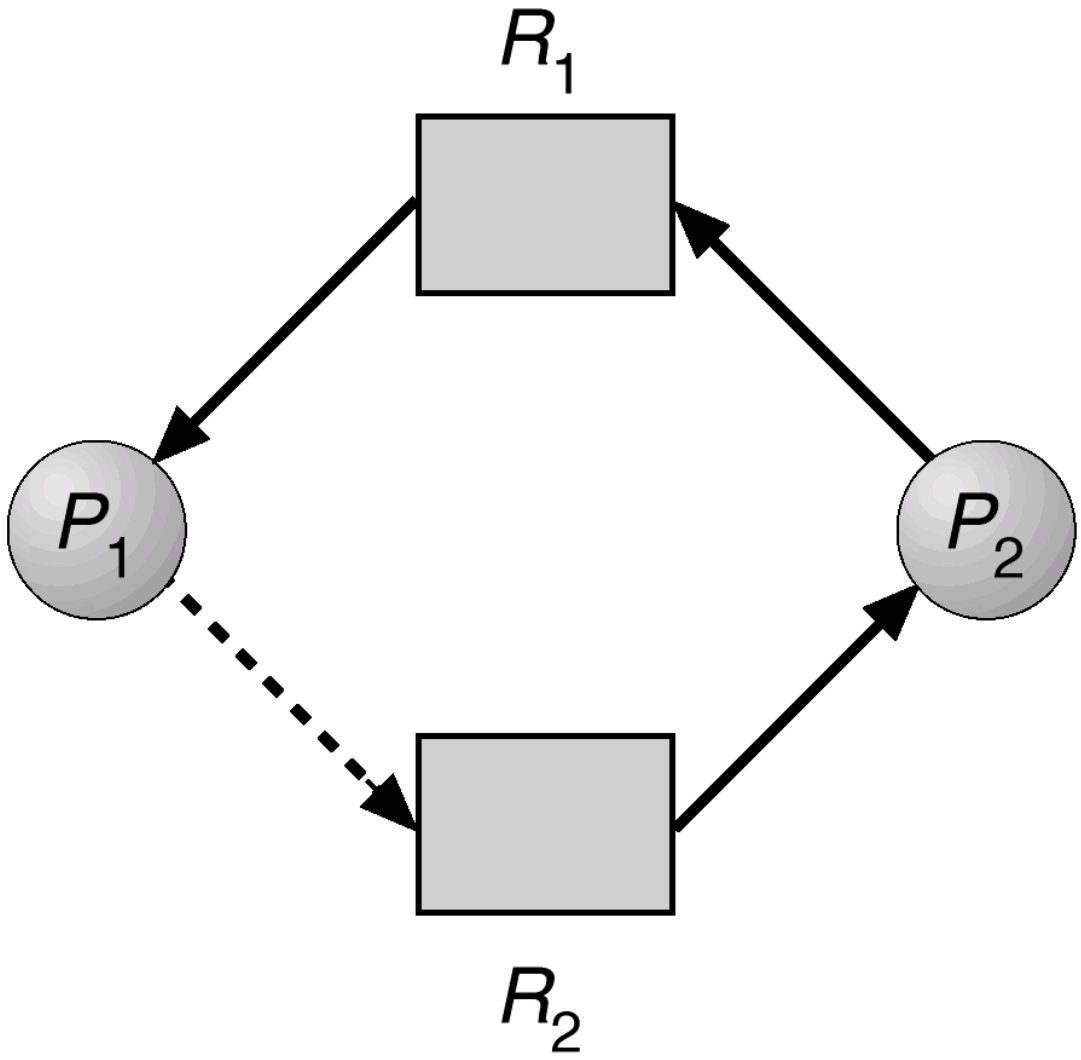
cioè se si trasforma
l'arco di prenotazione $P_2 \dashrightarrow R_2$
nell'arco di assegnazione $R_2 \longrightarrow P_2$



Stato non sicuro in un grafo di allocazione delle risorse

si ottiene il grafo

che rappresenta uno stato non sicuro (infatti, contiene un ciclo)



L'algoritmo del banchiere

- L'algoritmo del banchiere fu pubblicato da Dijkstra (1965)
- Questo nome è stato scelto perché l'algoritmo si potrebbe impiegare in un sistema bancario per assicurare che la banca non allochi mai il denaro disponibile in modo da non poter soddisfare le richieste di prestito di tutti i suoi clienti
- Applicabile in generale (anche nel caso di tipi di risorse con istanze multiple)
- Ogni processo deve dichiarare il numero massimo di istanze per ogni tipo di risorsa di cui può avere bisogno durante l'esecuzione (ovviamente, non può superare il numero delle risorse del sistema)
- L'algoritmo è attivato ad ogni richiesta e ad ogni rilascio
  - Quando un processo richiede delle risorse, si deve stabilire se la loro assegnazione lascia il sistema in uno stato sicuro; altrimenti, il processo deve attendere
  - Un processo, dopo aver ottenuto tutte le risorse richieste, dovrà rilasciarle in un periodo di tempo finito

Strutture dati dell'algoritmo

Sia **n** = num. dei processi e **m** = num. di tipi di risorse

Lo stato del sistema è rappresentato dalle seguenti strutture dati

- Available: vettore di lunghezza m
  se Available[j] = k, ci sono k istanze di risorse di tipo R_j disponibili
- Max: matrice n x m
  se Max [i,j] = k, allora il processo P_i, durante la sua esecuzione, può richiedere al più k istanze di risorse di tipo R_j
- Allocation: matrice n x m
  se Allocation[i,j] = k allora al processo P_i sono attualmente assegnate k istanze di risorse del tipo R_j
- Need: matrice n x m
  se Need[i,j] = k, allora il processo P_i può avere bisogno di altre k istanze di risorse del tipo R_j per completare la sua esecuzione

Proprietà invariante:

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

Algoritmo per verificare se uno stato è sicuro

1. Siano Work e Finish vettori rispettivamente di lunghezza m e n. Inizializzazione:
   a) Work = Available
   b) Finish[i] = false per i = 1,2, …, n
2. Cercare un indice i tale che:
   a) Finish[i] == false
   b) $Need_i \leq Work$

   Se un tale i non esiste, passare al punto 4
3. Work = Work + $Allocation_i$
   Finish[i] = true
   Tornare al punto 2
4. Se per ogni i, Finish[i] == true, allora lo stato è sicuro; altrimenti non lo è.

Costo: $O(m \times n^2)$ operazioni per trovare una sequenza sicura

Un'implementazione in Java

```
for(int j=0; j<n; j++) {
  // first find a thread that can finish
  for(int i=0; i<n; i++) {
    if (!finish[i]) {
      boolean temp = true;
      for(int k=0; k<m; k++) {
        if (need[i][k] > work[k])
          temp = false;
      }
      if (temp) { // if this thread can finish
        finish[i] = true;
        for (int x = 0; x < m; x++)
          work[x] += allocation[i][x];
      }
    }
  }
}
```

Algoritmo del banchiere

Sia $Request_i$ il vettore (di lunghezza m) delle richieste di P_i

1. Se $Request_i \leq Need_i$, passare al punto 2.

   Altrimenti sollevare una condizione di errore, poichè la richiesta di P_i ha ecceduto il numero massimo di risorse di cui P_i ha dichiarato di aver bisogno per portare a termine la sua esecuzione

2. Se $Request_i \leq Available$, passare al punto 3.

   Altrimenti P_i deve attendere poichè le risorse disponibili non sono sufficienti

3. Il sistema assegna al processo P_i le risorse richieste modificando momentaneamente lo stato nel modo seguente:

   $Available = Available - Request_i;$
   $Allocation_i = Allocation_i + Request_i;$
   $Need_i = Need_i - Request_i;$

   Viene quindi invocato l'*algoritmo per verificare se uno stato è sicuro*

   - Se lo stato ottenuto è sicuro ⇒ le risorse vengono realmente assegnate a P_i e lo stato diventa effettivo
   - Se lo stato ottenuto è non sicuro ⇒ P_i deve aspettare e viene ristabilito il vecchio stato di allocazione delle risorse

     (cioè i precedenti valori di Available, Allocation e Need)

Attivazione dell'algoritmo

L'algoritmo è attivato ad ogni richiesta e ad ogni rilascio

- Quando un processo effettua una nuova richiesta, viene invocato l'algoritmo del banchiere con l'identificativo del richiedente
- Quando un processo effettua un rilascio di alcune risorse ad esso allocate oppure termina, viene aggiornato lo stato di allocazione

  Es. se il processo P_i rilascia k (istanze di) risorse di tipo R_j, aggiornare lo stato di allocazione significa effettuare i seguenti assegnamenti

  ```
  Available[j] = Available[j] + k;
  Allocation[i,j] = Allocation[i,j] - k;
  Need[i,j] = Need[i,j] + k;
  ```

  e quindi viene invocato l'algoritmo del banchiere per ogni processo la cui richiesta è in attesa

Esempio dell'algoritmo del banchiere

- Si consideri un sistema con 5 processi da P_0 a P_4 e 3 tipi di risorse:
  - A: 10 istanze
  - B: 5 istanze
  - C: 7 istanze
- Supponiamo che lo stato corrente sia il seguente:

	Allocation	Max
	A B C	A B C
P_0	0 1 0	7 5 3
P_1	2 0 0	3 2 2
P_2	3 0 2	9 0 2
P_3	2 1 1	2 2 2
P_4	0 0 2	4 3 3

- Lo stato corrente è sicuro?

Esempio dell'algoritmo del banchiere

- La matrice Need, definita come Max – Allocation, ed il vettore Available delle risorse attualmente disponibili, ricavabile sottraendo alle risorse complessive del sistema quelle già allocate, quindi $Available = [10,5,7] - \sum_i Allocation_i$, sono

	Allocation	Max		Need	Available
	A B C	A B C		A B C	A B C
P_0	0 1 0	7 5 3	P_0	7 4 3	3 3 2
P_1	2 0 0	3 2 2	P_1	1 2 2	
P_2	3 0 2	9 0 2	P_2	6 0 0	
P_3	2 1 1	2 2 2	P_3	0 1 1	
P_4	0 0 2	4 3 3	P_4	4 3 1	

- Il sistema è in uno stato sicuro poichè la sequenza < P_1, P_3, P_4, P_2, P_0 > è sicura, infatti …

Esempio dell'algoritmo del banchiere

	Allocation	Max		Need	Available
	A B C	A B C		A B C	A B C
P_0	0 1 0	7 5 3	P_0	7 4 3	3 3 2
P_1	2 0 0	3 2 2	P_1	1 2 2	
P_2	3 0 2	9 0 2	P_2	6 0 0	
P_3	2 1 1	2 2 2	P_3	0 1 1	
P_4	0 0 2	4 3 3	P_4	4 3 1	

$Need_1 = (1\ 2\ 2) \leq (3\ 3\ 2) = Work\ (= Available)$

$\Rightarrow P_1$ ottiene le risorse e termina e si ha Work = (5 3 2)

$Need_3 = (0\ 1\ 1) \leq (5\ 3\ 2) \Rightarrow P_3$ ottiene le risorse e termina e Work = (7 4 3)

$Need_4 = (4\ 3\ 1) \leq (7\ 4\ 3) \Rightarrow P_4$ ottiene le risorse e termina e Work = (7 4 5)

$Need_2 = (6\ 0\ 0) \leq (7\ 4\ 5) \Rightarrow P_2$ ottiene le risorse e termina e Work = (10 4 7)

$Need_0 = (7\ 4\ 3) \leq (10\ 4\ 7) \Rightarrow P_0$ ottiene le risorse e termina e Work = (10 5 7)

Esempio dell'algoritmo del banchiere

- La richiesta (1,0,2) da P_1 può essere soddisfatta? Applichiamo l'algoritmo del banchiere
- Controlliamo che

  $Request_1 \leq Need_1$ cioè (1 0 2) ≤ (1 2 2)

  $Request_1 \leq Available$ cioè (1 0 2) ≤ (3 3 2)
- Aggiorniamo lo stato

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

- Invochiamo l'algoritmo per la verifica dello stato sicuro che restituisce la sequenza sicura $< P_1, P_3, P_4, P_2, P_0 >$

Esempio dell'algoritmo del banchiere

- La richiesta (3,3,0) di P_4 può essere soddisfatta?

  Non immediatamente, bisogna aspettare la disponibilità di ulteriore risorse poiché quelle attualmente disponibili sono (2 3 0)

- La richiesta (0,2,0) di P_0 può essere soddisfatta?

  No, perché lo stato raggiunto non sarebbe sicuro: è semplice verificarlo in questo caso perché non esiste un indice i tale che $Need_i \leq Work$

Evitare i deadlock: valutazione

- Caratteristiche
  - Prevenzione dinamica
  - Considerano il caso pessimo: in qualsiasi momento, tutti i processi possono richiedere le risorse residue di cui hanno bisogno contemporaneamente e il SO deve essere in grado di poter soddisfare ordinatamente tutte le richieste
- Inconvenienti
  - Sono costosi in termini di overhead per il sistema
  - Non permettono di utilizzare al massimo le risorse (proprio perché si basano sul caso pessimo)
  - Riducono la produttività
  - Non sono sempre applicabili perché
    - le esigenze dei processi in termini di risorse necessarie per il completamento dell'esecuzione raramente sono note a priori
    - i processi in un sistema non sono fissi, ma variano dinamicamente
    - le risorse ritenute disponibili in un sistema possono improvvisamente svanire (es. i dischi possono rompersi, i dispositivi sconnettersi)

Algoritmo del banchiere: valutazione

- In teoria l'algoritmo risolve il problema di evitare i deadlock
  - È meno efficiente dell'algoritmo che usa il grafo delle risorse
  - Ma è applicabile anche nei casi di tipi di risorsa con istanze multiple
- In pratica, pochi o nessun sistema esistente lo utilizza, per via degli inconveniente menzionati in precedenza
- Alcuni sistemi, tuttavia, per prevenire deadlock usano un'euristica simile a quella dell'algoritmo del banchiere
  - Ad esempio, le reti possono limitare il traffico quando l'utilizzo del buffer raggiunge un valore superiore, per esempio, al 70%, stimando che il restante 30% sarà sufficiente affinché gli utenti attuali completino i servizi richiesti e restituiscano le risorse impiegate

Stallo (o deadlock)

- Il problema del deadlock
- Modello delle risorse di un sistema
- Caratterizzazione del deadlock
- Metodi di gestione dei deadlock
  1. Prevenire il deadlock
  2. Evitare il deadlock
  3. Rilevare il deadlock e ripristinare il sistema
  4. Ignorare il deadlock

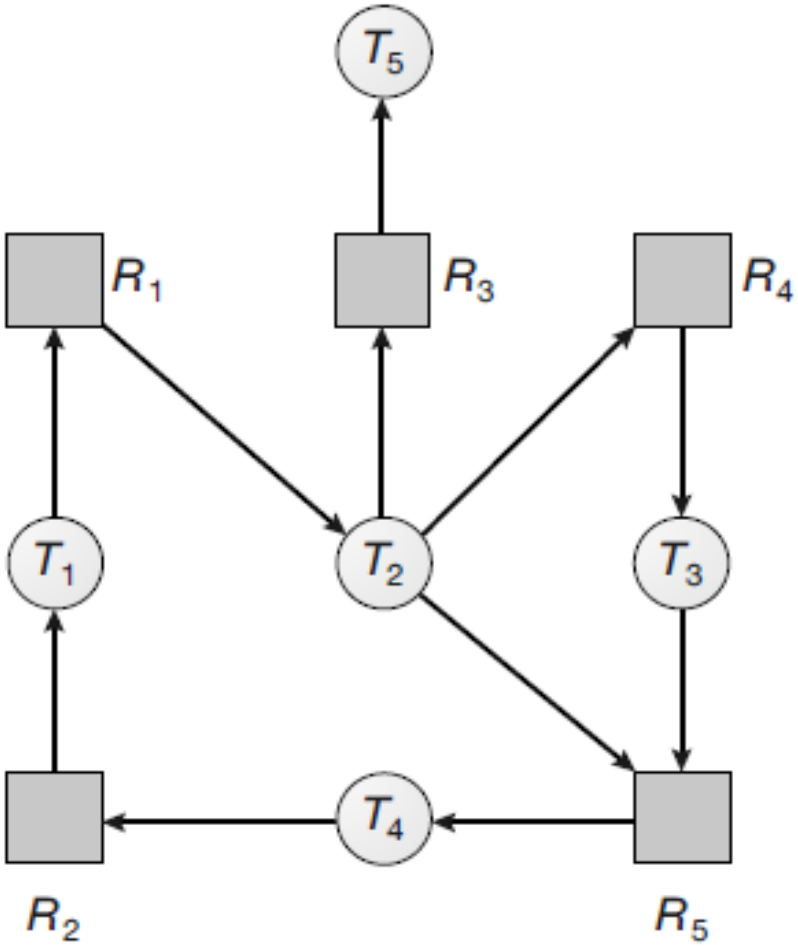
Rilevare i deadlock

- Nei sistemi che non utilizzano algoritmi per prevenire o per evitare i deadlock, tali situazioni si possono presentare
- In un ambiente di questo genere, il sistema può fornire
  - un algoritmo per il rilevamento della presenza nel sistema di una situazione di deadlock
  - un algoritmo per il ripristino del sistema da una situazione di deadlock

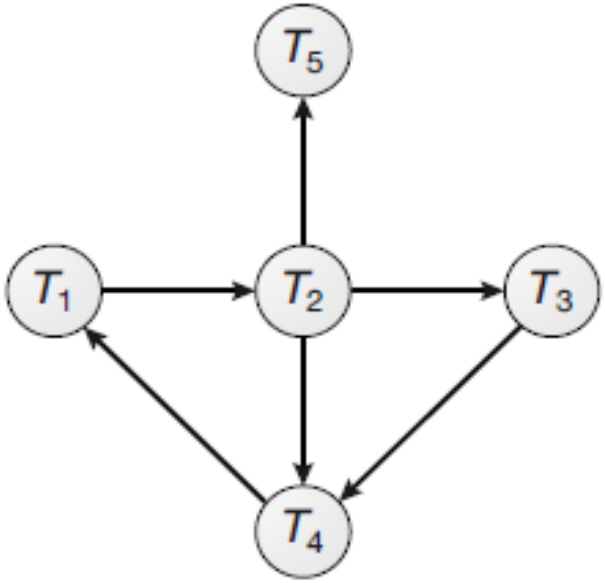
Singola istanza per ogni tipo di risorsa

- Se ogni tipo di risorsa del sistema contiene una sola istanza, si può usare un grafo di allocazione delle risorse, oppure si può usare un grafo di attesa per rappresentare graficamente lo stato di allocazione del sistema in modo più conciso
  - I nodi rappresentano solo processi
  - Esiste un arco $P_i \rightarrow P_j$ se P_i è in attesa che P_j rilasci una risorsa
- Corrispondenza: nel grafo di attesa esiste l'arco $P_i \rightarrow P_j$ se, e solo se, nel grafo di allocazione delle risorse esistono gli archi $P_i \rightarrow R_k$ e $R_k \rightarrow P_j$ per qualche risorsa R_k
- Proprietà: Nel sistema esiste un deadlock se, e solo se, nel grafo di attesa esiste (almeno) un ciclo
- Idea: per individuare le situazioni di stallo, il sistema mantiene un grafo di attesa e periodicamente invoca un algoritmo di ricerca di cicli
- Costo dell'algoritmo: $O(n^2)$, dove n è il numero di vertici del grafo
  - Conviene ragionare sul grafo di attesa anziché su quello di allocazione delle risorse perché ha meno nodi

Grafo di allocazione risorse & grafo di attesa corrispondente



Grafo di allocazione delle risorse



Corrispondente grafo di attesa

Tipi di risorsa con istanze multiple

- Il SO controlla la presenza di un deadlock nel sistema cercando di costruire sequenze fattibili di eventi che permettano a tutti i processi di ottenere le risorse che hanno richiesto (e, quindi, di terminare)
  - Anche se si tratta di sequenze con cui soddisfare le richieste dei processi, **non** si tratta più di "sequenze sicure"!
- Se **n** = num. dei processi e **m** = num. di tipi di risorse, lo stato del sistema è rappresentato dalle seguenti strutture dati
  - Available: vettore di lunghezza m che indica il numero di istanze disponibili per ciascun tipo di risorsa
  - Allocation: matrice n x m che indica il numero di risorse di ogni tipo correntemente allocate a ciascun processo
  - Request: matrice n x m che indica le richieste correnti di ogni processo
    - Se Request [i,j] == k, allora il processo P_i sta chiedendo k ulteriori istanze della risorsa di tipo R_j
- Costo: $O(m \times n^2)$ operazioni per rilevare se c'è uno stallo

Algoritmo di rilevamento

1. Siano Work e Finish vettori di lunghezza m and n rispettivamente. Inizializzazione
   a) Work = Available
   b) Per i = 1,2, …, n, se $Allocation_i \neq 0$, allora Finish[i] = false;
      altrimenti, Finish[i] = true.
2. Trovare un indice i tale che valgano entrambe le seguenti condizioni:
   a) Finish[i] == false
   b) $Request_i \leq Work$

   Se un tale i non esiste, passare al punto 4.
3. Work = Work + $Allocation_i$
   Finish[i] = true
   Tornare al punto 2.
4. Se Finish[i] == false, per qualche i, $1 \leq i \leq n$, allora il sistema è in stallo; inoltre, se Finish[i] == false, allora P_i è in stallo.
   Altrimenti il sistema non è in stallo.

Algoritmo di rilevamento: osservazioni

- Simile all'algoritmo di "verifica dello stato sicuro" utilizzato dall'algoritmo del banchiere, da cui differisce perché lavora con le esigenze effettive, anziché con le quelle massime
- Se $Allocation_i == 0$ allora P_i non possiede risorse, quindi non può far parte di un gruppo di processi in attesa circolare (anche se potrebbe essere che $Request_i \neq 0$); perciò in tal caso si pone Finish[i] = true
- Alla fine dell'algoritmo, se Finish[i] == false allora P_i è in stallo, ma se Finish[i] == true allora non è detto che P_i non sia in stallo: potrebbe non possedere risorse ed essere bloccato in attesa di risorse allocate ad un processo (indirettamente) coinvolto in un'attesa circolare
- Ipotesi ottimistica: per completare la propria esecuzione, P_i non richiede risorse aggiuntive a parte quelle specificate in $Request_i$
  - Quindi se $Request_i \leq Work$, allora P_i può terminare rilasciando le risorse attualmente possedute
  - Se l'ipotesi non fosse valida (in questo approccio non si conoscono le necessità massime), si potrebbe verificare un deadlock che sarebbe comunque rilevato da invocazioni successive dell'algoritmo

Esempio di algoritmo di rilevamento

- Si consideri un sistema con 5 processi P_0-P_4 e 3 tipi di risorse A, B, C
- Supponiamo che lo stato attuale sia il seguente:

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

- La sequenza <P_0, P_2, P_3, P_1, P_4> fa sì che Finish[i] == true per ogni i, infatti

Req_0 = (0 0 0) ≤ (0 0 0) = Work (= Available)

⇒ P_0 ottiene le risorse e termina e si ha Work = (0 1 0)

Req_2 = (0 0 0) ≤ (0 1 0) ⇒ P_2 ottiene le risorse e termina e Work = (3 1 3)

Req_3 = (1 0 0) ≤ (3 1 3) ⇒ P_3 ottiene le risorse e termina e Work = (5 2 4)

Req_1 = (2 0 2) ≤ (5 2 4) ⇒ P_1 ottiene le risorse e termina e Work = (7 2 4)

Req_4 = (0 0 2) ≤ (7 2 4) ⇒ P_4 ottiene le risorse e termina e Work = (7 2 6)

Esempio di algoritmo di rilevamento

- Supponiamo ora che in aggiunta alle esigenze precedenti, P_2 necessiti di una risorsa di tipo C; in pratica, abbiamo lo stato seguente

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

- In questo stato c'è deadlock?

Esempio di algoritmo di rilevamento

- Supponiamo ora che in aggiunta alle esigenze precedenti, P_2 necessiti di una risorsa di tipo C; in pratica, abbiamo lo stato seguente

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

- In questo stato c'è deadlock?
  - Si possono recuperare le risorse possedute da P_0 (che non ha esigenze residue), infatti

    $Req_0 = (0\ 0\ 0) \leq (0\ 0\ 0) = Work\ (= Available)$

    $\Rightarrow P_0$ ottiene le risorse e termina e si ha Work = (0 1 0)

    ma non ci sono comunque risorse sufficienti per soddisfare gli altri processi poiché non esiste un i t.c. $Req_i \leq Work = (0\ 1\ 0)$
  - C'è quindi uno stallo che coinvolge i processi P_1, P_2, P_3 e P_4

Uso dell'algoritmo di rilevamento

- Quando il SO dovrebbe eseguire l'algoritmo di rilevamento? Dipende da:
  - con quale frequenza può presentarsi un deadlock
  - quanti processi saranno coinvolti dal deadlock
- La frequenza di esecuzione dovrebbe aumentare all'aumentare di ciascuno dei 2 fattori
  - Caso estremo: ad ogni richiesta non immediatamente soddisfacibile (molto costoso)
  - Con cadenza periodica, allo scadere di un certo intervallo di tempo
  - Quando le prestazioni degradano (es. uso della CPU sotto il 40%)
- L'algoritmo di rilevamento, se fosse eseguito in modo arbitrario, potrebbe trovare molti cicli nel grafo di attesa
  - Il SO potrebbe quindi non essere in grado di stabilire quale dei molti processi bloccati ha causato il deadlock

Modalità di ripristino dal deadlock

- Informare l'operatore del sistema
  (il rispristino non è quindi a carico del SO)

- Effettuare un ripristino automatico a carico del SO
  - Terminazione forzata di processi
  - Rilascio anticipato di risorse

Terminazione forzata di processi

- Far terminare forzatamente tutti i processi in deadlock: molto costoso perché le loro computazioni vanno ripetute
- Far terminare un processo alla volta fino ad eliminare il ciclo
  - però l'algoritmo di rilevamento va eseguito più volte
  - in quale ordine devono essere fatti terminare i processi? Cominciare da quello la cui riesecuzione ha `costo' minore; molti fattori influenzano la scelta, quali
    - Priorità e tipo del processo (es. interattivo o batch)
    - Per quanto tempo il processo ha già elaborato e per quanto tempo ancora l'esecuzione del processo dovrà proseguire
    - Quanti e quali tipi di risorse sono state usate dal processo
    - Quante altre risorse il processo necessita per completare la propria elaborazione
    - Quanti processi devono essere fatti terminare
- Forzare la terminazione di un processo non è comunque un'operazione semplice (es. si rischia inconsistenza sui dati condivisi su cui eventualmente sta lavorando)

Rilascio anticipato di risorse

- Vengono pre-rilasciate (e poi riassegnate) alcune risorse fino ad interrompere il ciclo, cercando di minimizzare il costo
  - Complicazione: stabilire un ordine di prelazione che minimizzi i costi
- Rollback: riportare in uno stato corretto il processo a cui è stata tolta la risorsa (stato iniziale o intermedio, più costoso perché richiede il mantenimento di più informazioni)
- Starvation: in un sistema in cui la selezione della vittima della prelazione è basata soprattutto su fattori di costo, si potrebbe finire per scegliere come vittime sempre gli stessi processi
  - Nella valutazione del costo, bisognerebbe tener conto del numero di rollback subiti dai processi

Rilevare il deadlock e ripristinare il sistema: valutazione

Inconvenienti: ci sono costi per

- la memorizzazione delle informazioni necessarie
- l'esecuzione dell'algoritmo di rilevamento
- il ripristino dello stato del sistema
  - riesecuzione dei processi abortiti
  - possibili perdite di informazioni

Stallo (o deadlock)

- Il problema del deadlock
- Modello delle risorse di un sistema
- Caratterizzazione del deadlock
- Metodi di gestione dei deadlock
  1. Prevenire il deadlock
  2. Evitare il deadlock
  3. Rilevare il deadlock e ripristinare il sistema
  4. Ignorare il deadlock

Ignorare i deadlock

- L'overhead per evitare i deadlock o per il loro rilevamento e il ripristino di uno stato corretto rende tali metodi di gestione costosi e poco utilizzati in pratica
- Quindi un SO può
  - basarsi sull'approccio di prevenzione (statica) dei deadlock in cui non sono necessarie azioni di gestione eplicite, ma non sempre ciò è fattibile
  - oppure non preoccuparsi della possibilità che il deadlock si verifichi
- Ignorare del tutto il problema e assumere che i deadlock non si presentino mai
  - Comportamento frequente in molti sistemi operativi, compresi Linux e Windows (la gestione dello stallo è a carico dei programmatori del kernel e delle applicazioni)
  - Adottato sia perché è il metodo meno costoso in termini di overhead di gestione sia a causa della scarsa frequenza dell'occorrenza dello stallo in certi sistemi (per cui è inutile e costoso implementare metodi di gestione dello stallo)
- Però, sia l'aumento del numero delle risorse e dei processi/thread nei sistemi moderni, che la programmazione multithread e i sistemi multicore, rendono questa soluzione sempre meno conveniente

Gestione dei deadlock in pratica

- Alcuni sviluppatori sostengono che nessuno degli approcci di base che abbiamo visto si adatta da solo a risolvere l'intero spettro di problemi di allocazione delle risorse nei SO
- Tali approcci possono comunque essere combinati in modo da permettere la selezione dell'approccio migliore per ciascun tipo di risorsa del sistema