Sistemi Operativi per LT Informatica

Deadlock

Docente: Salvatore Sorce

Introduzione

"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 has gone"

Legge del Kansas
di inizio secolo

Risorse

- **Un sistema di elaborazione è composto da un insieme di risorse da assegnare ai processi presenti**
- **I processi competono nell'accesso alle risorse**
- **Esempi di risorse**
  - memoria
  - stampanti
  - processore
  - dischi
  - interfaccia di rete
  - descrittori di processo

Classi di risorse

- **Le risorse possono essere suddivise in classi**
  - le risorse appartenenti alla stessa classe sono equivalenti
  - esempi: byte della memoria, stampanti dello stesso tipo, etc.
- **Definizioni:**
  - le risorse di una classe vengono dette *istanze* della classe
  - 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

Assegnazione delle risorse

- **Risorse ad assegnazione statica**
  - avviene al momento della creazione del processo e rimane valida fino alla terminazione
  - esempi: *descrittori di processi, aree di memoria (in alcuni casi)*
- **Risorse ad assegnazione dinamica**
  - i processi
    - richiedono le risorse durante la loro esistenza
    - le utilizzano una volta ottenute
    - le rilasciano quando non più necessarie (eventualmente alla terminazione del processo
  - esempi: *periferiche di I/O, aree di memoria (in alcuni casi)*

Tipi di richieste

- **Richiesta singola:**
  - si riferisce a una singola risorsa di una classe definita
  - è il caso normale
- **Richiesta multipla**
  - si riferisce a una o più classi, e per ogni classe, ad una o più risorse
  - deve essere soddisfatta integralmente

Tipi di richieste

- **Richiesta bloccante**
  - il processo richiedente si sospende se non ottiene immediatamente l'assegnazione
  - la richiesta rimane pendente e viene riconsiderata dalla funzione di gestione ad ogni rilascio
- **Richiesta non bloccante**
  - la mancata assegnazione viene notificata al processo richiedente, senza provocare la sospensione

Tipi di risorse

- **Risorse seriali (o con accesso mutuamente esclusivo)**
  - una singola risorsa non può essere assegnata a più processi contemporaneamente
  - esempi:
    - i processori
    - le sezioni critiche
    - le stampanti
- **Risorse non seriali**
  - esempio:
    - file di sola lettura

Risorse prerilasciabili ("preemptable")

- **Definizione**
  - una risorsa si dice prerilasciabile se la funzione di gestione può sottrarla ad un processo prima che questo l'abbia effettivamente rilasciata
- **Meccanismo di gestione:**
  - il processo che subisce il prerilascio deve sospendersi
  - la risorsa prerilasciata sarà successivamente restituita al processo

Risorse prerilasciabili

- **Una risorsa è prerilasciabile:**
  - se il suo stato non si modifica durante l'utilizzo
  - oppure il suo stato può essere facilmente salvato e ripristinato
- **Esempi:**
  - processore
  - blocchi o partizioni di memoria
    (nel caso di assegnazione dinamica)

Risorse non prerilasciabili

- **Definizione**
  - la funzione di gestione non può sottrarle al processo al quale sono assegnate
  - sono non prerilasciabili le risorse il cui stato non può essere salvato e ripristinato
- **Esempi**
  - stampanti
  - classi di sezioni critiche
  - partizioni di memoria
    (nel caso di gestione statica)

Deadlock

- **Come abbiamo visto**
  - i deadlock impediscono ai processi di terminare correttamente
  - le risorse bloccate in deadlock non possono essere utilizzati da altri processi
- **Ora vediamo**
  - le condizioni necessarie (e sufficienti) affinché un deadlock si presenti
  - le tecniche che possono essere utilizzate per gestire il problema dei deadlock

Condizioni per avere un deadlock

- **Mutua esclusione**
  - le risorse coinvolte devono essere seriali
- **Assenza di prerilascio**
  - le risorse coinvolte non possono essere prerilasciate, ovvero devono essere rilasciate volontariamente dai processi che le controllano
- **Richieste bloccanti (detta anche "hold and wait")**
  - le richieste devono essere bloccanti, e un processo che ha già ottenuto risorse può chiederne ancora

Condizioni per avere un deadlock

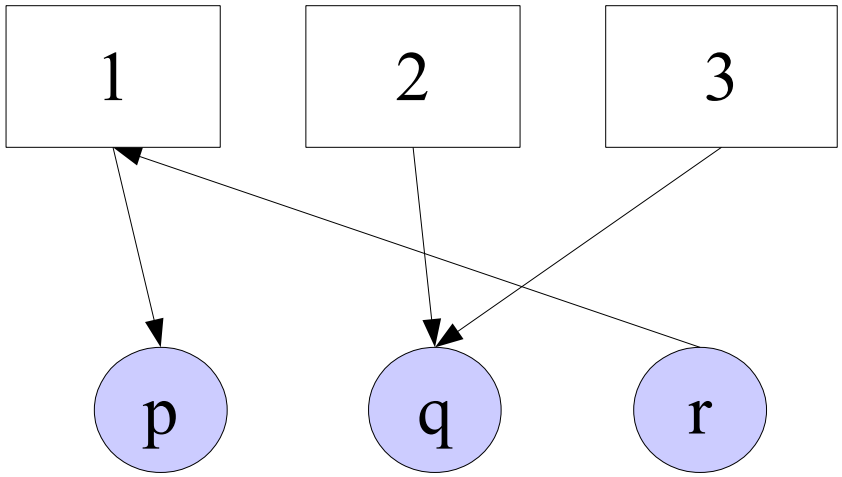
- **Attesa circolare**
  - esiste una sequenza di processi P_0,P_1, ..., P_n, tali per cui P_0 attende una risorsa controllata da P_1, P_1 attende una risorsa controllata da P_2, ..., e P_n attende una risorsa controllata da P_0

- **L'insieme di queste condizioni è necessario e sufficiente**
  - devono valere tutte contemporaneamente affinché un deadlock si presenti nel sistema

Grafo di Holt

- **Caratteristiche**
  - è un grafo *diretto*
    - gli archi hanno una direzione
  - è un grafo *bipartito*
    - i nodi sono suddivisi in due sottoinsiemi e non esistono archi che collegano nodi dello stesso sottoinsieme
    - i sottoinsiemi sono *risorse* e *processi*
  - gli archi *risorsa* → *processo* indicano che la risorsa è assegnata al processo
  - gli archi *processo* → *risorsa* indicano che il processo ha richiesto la risorsa

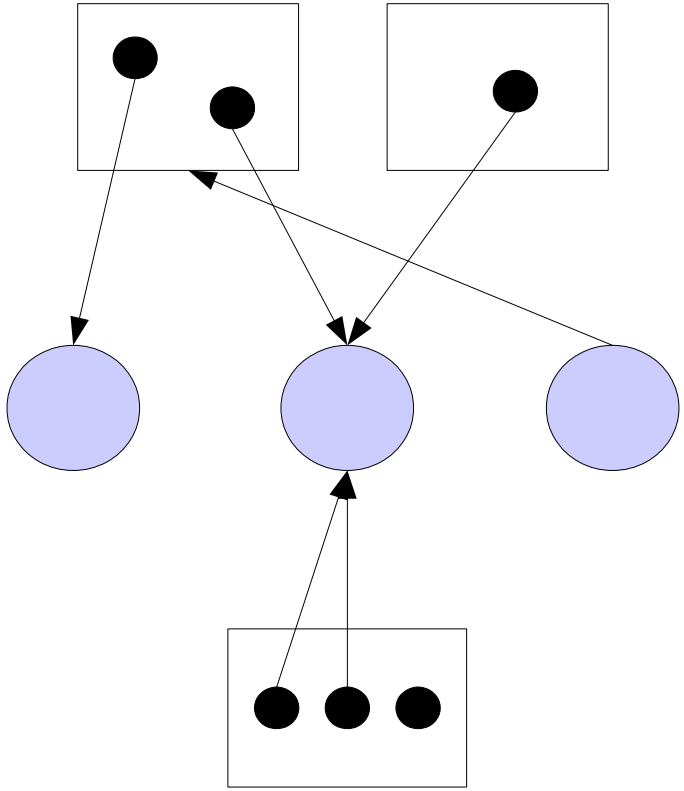
Grafo di Holt - Esempio



Grafo di Holt generale

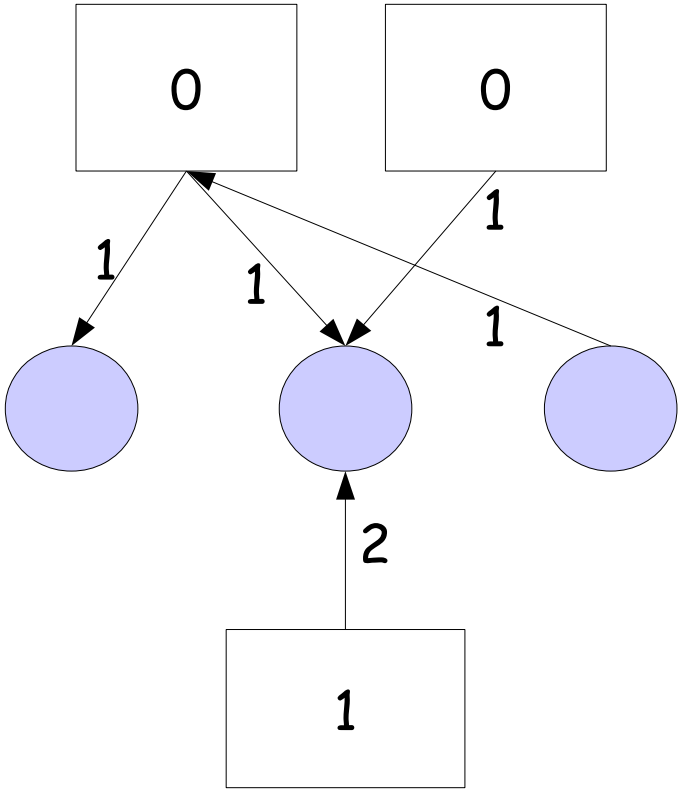
- **Nel caso di classi contenenti più istanze di una risorsa**
  - l'insieme delle risorse è partizionato in classi e gli archi di richiesta sono diretti alla classe e non alla singola risorsa
- **Rappresentazione**
  - i processi sono rappresentati da *cerchi*
  - le classi sono rappresentati come *contenitori rettangolari*
  - le risorse come *punti* all'interno delle classi
- **Nota:**
  - non si rappresentano grafi di Holt con archi relativi a richieste che possono essere soddisfatte
  - se esiste almeno un'istanza libera della risorsa richiesta, la risorsa viene assegnata

Grafo di Holt generale - Esempio



Grafo di Holt - Notazione alternativa

- **Alcuni autori preferiscono indicare numericamente:**
- **sugli archi:**
  - la molteplicità della richiesta (archi processo → classe)
  - la molteplicità dell'assegnazione (archi classe → processo)
- **all'interno delle classi**
  - il numero di risorse non ancora assegnate



Metodi di gestione dei deadlock

- **Deadlock detection and recovery**
  - permettere al sistema di entrare in stati di deadlock; utilizzare un algoritmo per rilevare questo stato ed eventualmente eseguire un'azione di recovery
- **Deadlock prevention / avoidance**
  - impedire al sistema di entrare in uno stato di deadlock
- **Ostrich algorithm**
  - ignorare il problema del tutto!

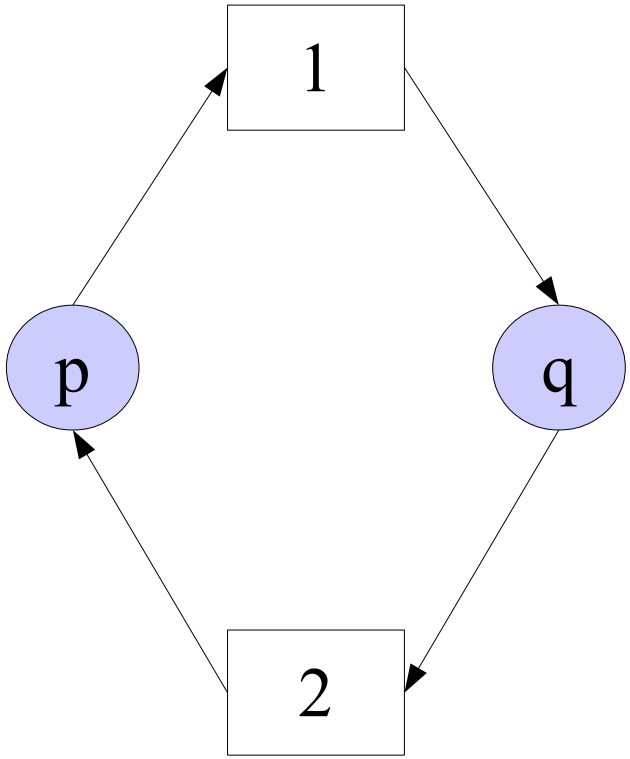
Deadlock detection

- **Descrizione**
  - mantenere aggiornato il grafo di Holt, registrando su di esso tutte le assegnazioni e le richieste di risorse
  - utilizzare il grafo di Holt al fine di riconoscere gli stati di deadlock
- **Problema:**
  - come riconoscere uno stato di deadlock?

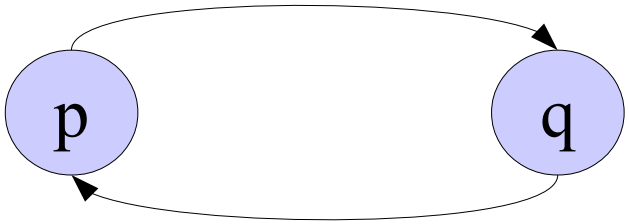
Caso 1 - Una sola risorsa per classe

- **Teorema**
  - se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili
  - *lo stato è di deadlock se e solo se il grafo di Holt contiene un ciclo*
- **Dimostrazione**
  - si utilizza una variante del grafo di Holt, detto *grafo Wait-For*
  - si ottiene un grafo wait-for eliminando i nodi di tipo risorsa e collassando gli archi appropriati
  - il grafo di Holt contiene un ciclo se e solo se il grafo Wait-for contiene un ciclo
  - se il grafo Wait-for contiene un ciclo, abbiamo attesa circolare

Grafo Wait-for



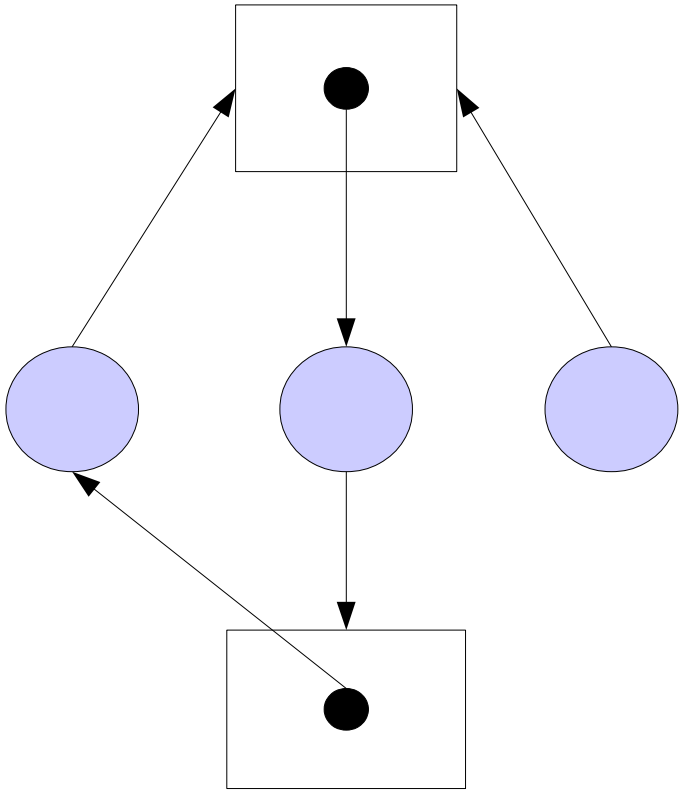
Grafo di Holt



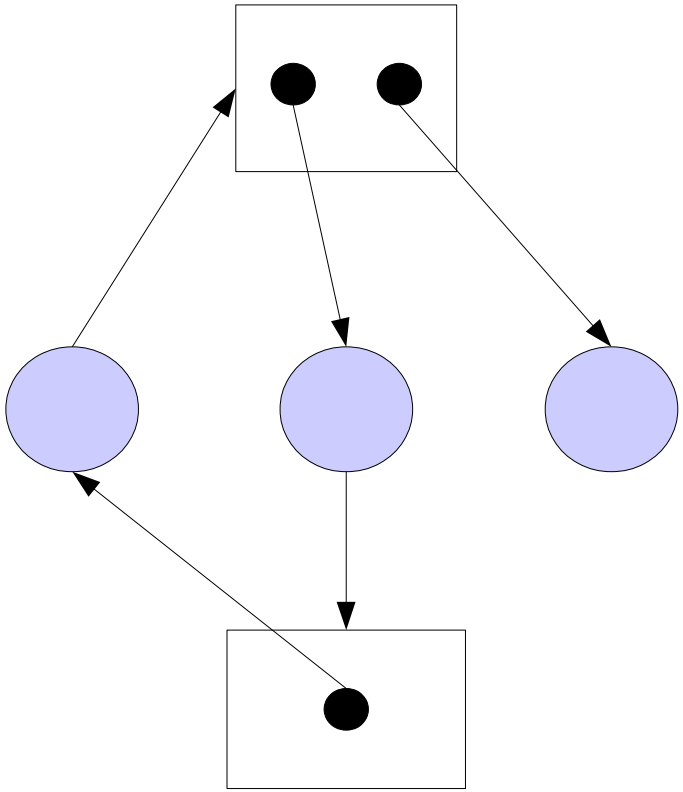
Grafo Wait-For

Caso 2 - Più risorse per classe

- La presenza di un ciclo nel caso di Holt non è condizione sufficiente per avere deadlock



Deadlock

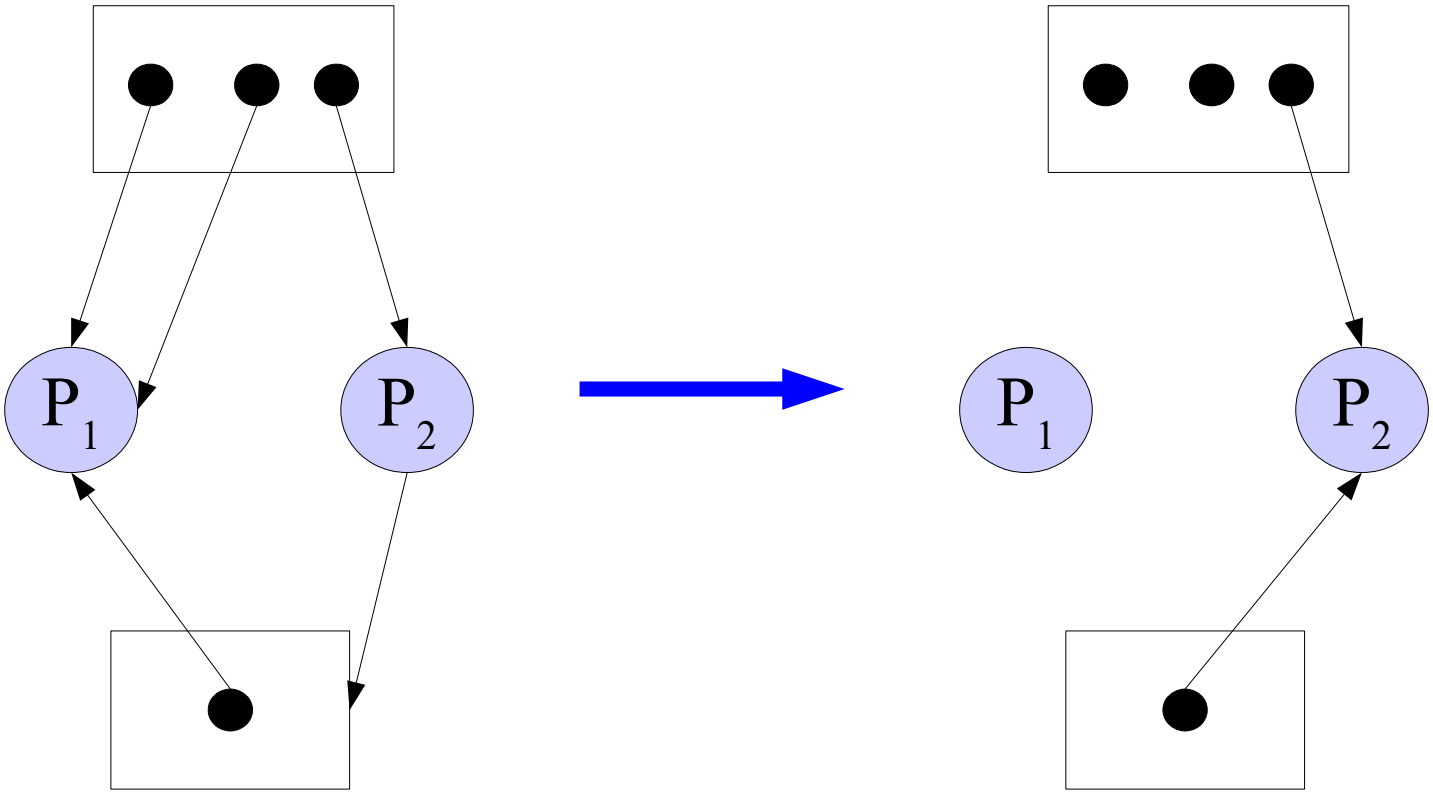


No Deadlock

Riducibilità di un grafo di Holt

- **Definizione**
  - un grafo di Holt si dice *riducibile* se esiste almeno un nodo processo con solo archi entranti
- **Riduzione**
  - consiste nell'eliminare tutti gli archi di tale nodo e riassegnare le risorse ad altri processi
- **Qual è la logica?**
  - eventualmente, un nodo che utilizza una risorsa prima o poi la rilascerà; a quel punto, la risorsa può essere riassegnata

Esempio di riduzione

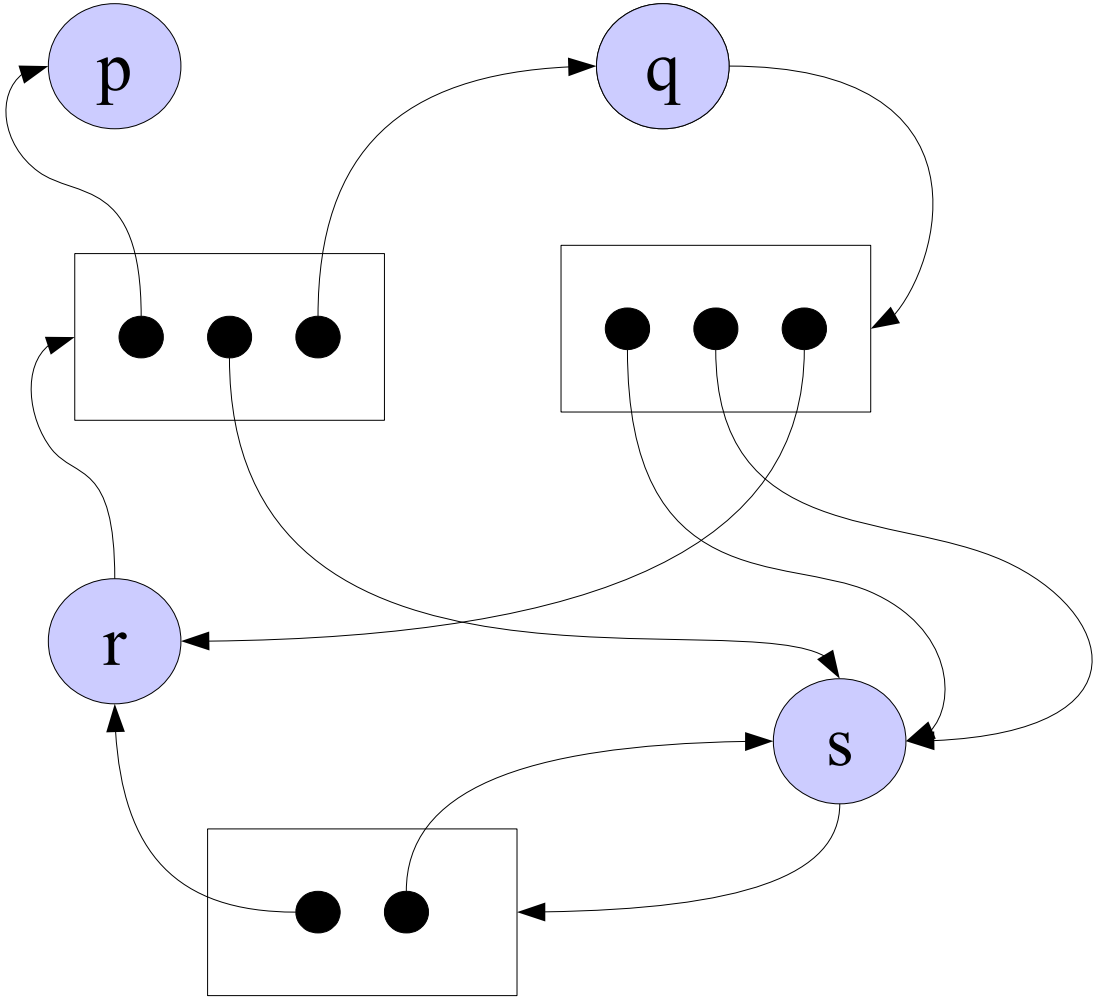


Riduzione per P_1

Deadlock detection con grafo di Holt

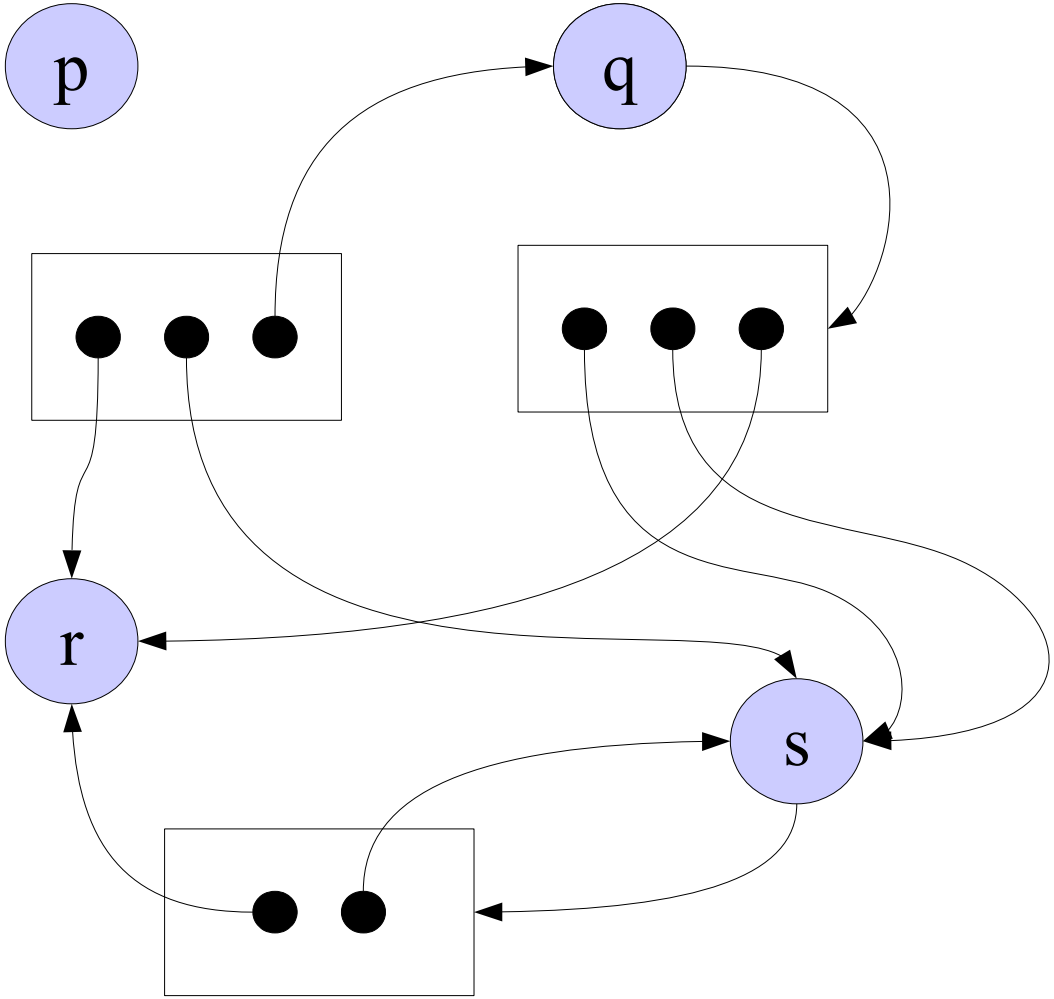
- **Teorema**
  - se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili
  - lo stato non è di deadlock se e solo se il grafo di Holt è *completamente riducibile*, i.e. esiste una sequenza di passi di riduzione che elimina tutti gli archi del grafo

Esempio



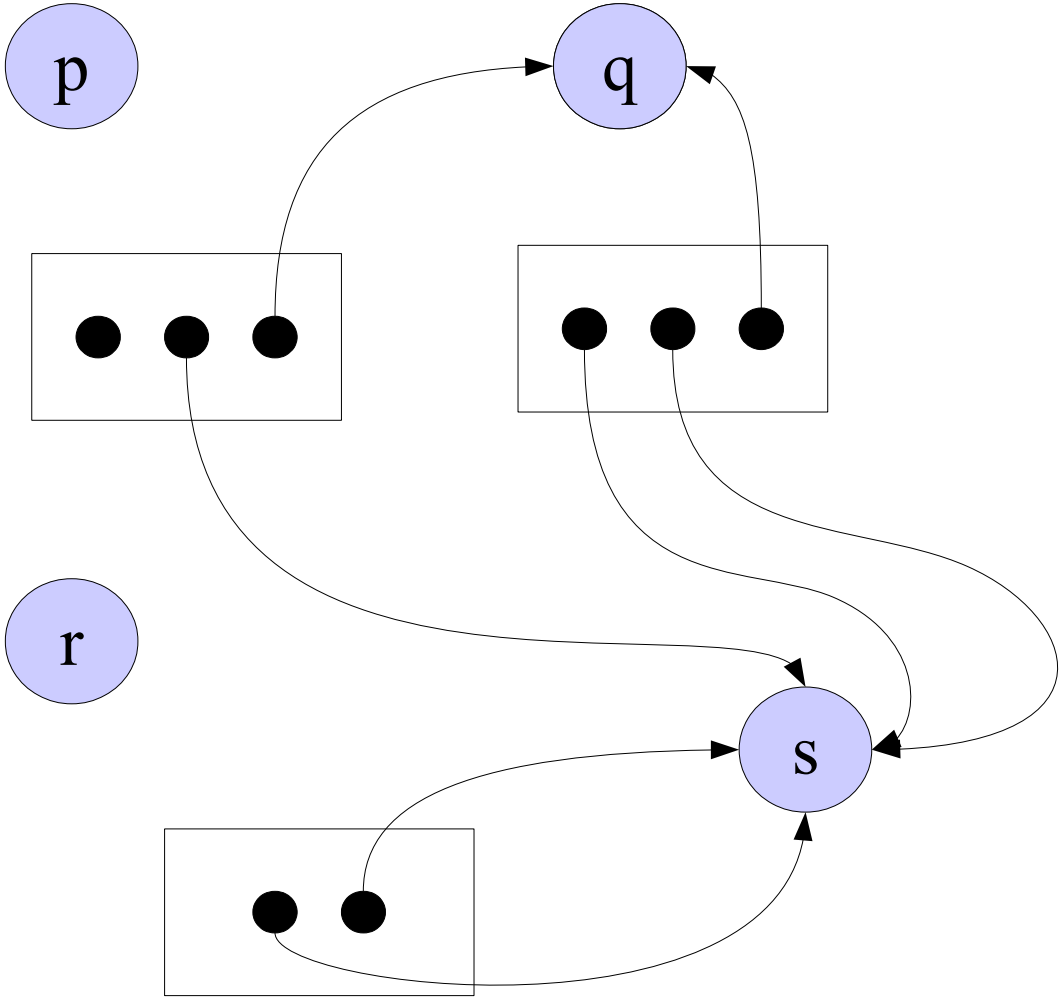
C'è deadlock?

Esempio



Riduzione di *p*
Assegnamento
risorsa a *r*

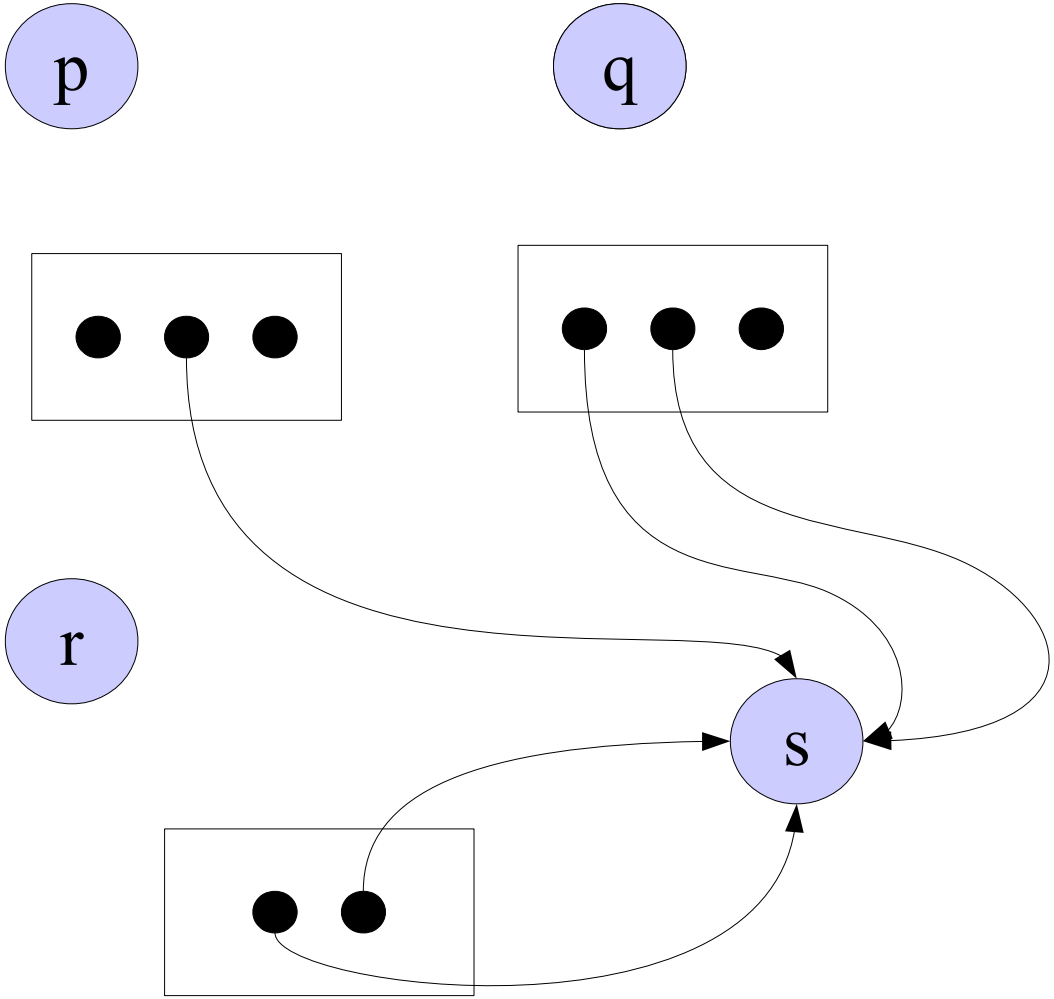
Esempio



Riduzione di r

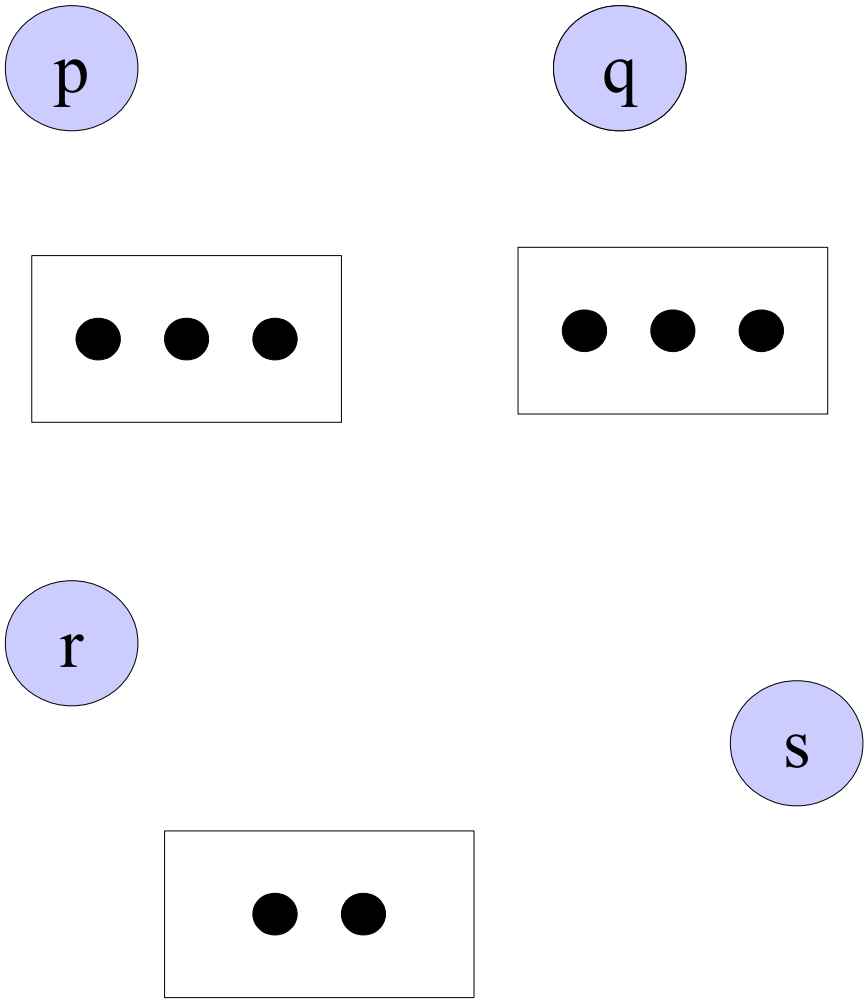
Assegnamento risorse a q,s

Esempio



Riduzione di *q*

Esempio

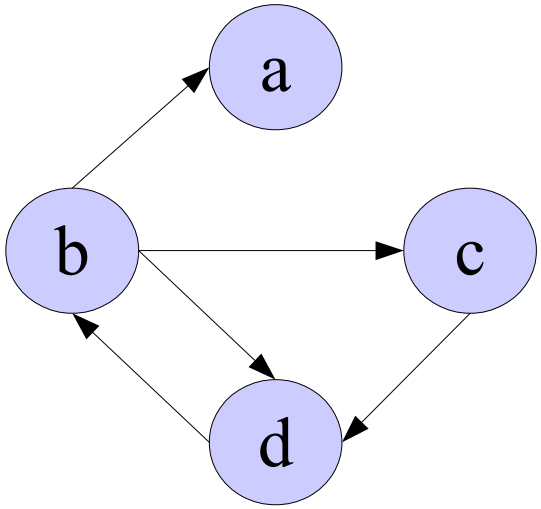


Riduzione di s

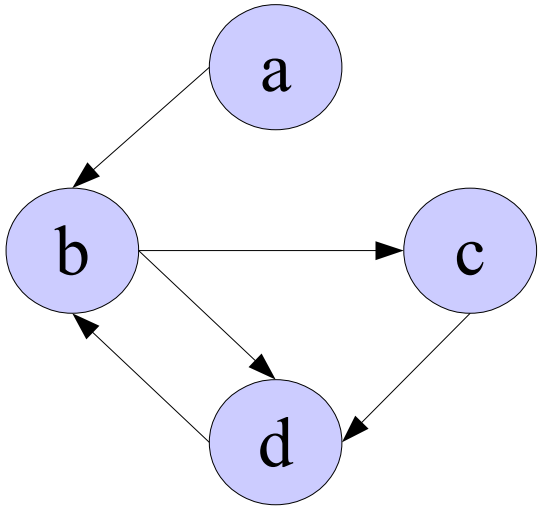
Non c'è deadlock

Deadlock detection - Knot

- **Definizione**
  - dato un nodo *n*, l'insieme dei nodi raggiungibili da *n* viene detto *insieme di raggiungibilità* di *n* (scritto R*(n)*)
  - un knot del grafo G è il massimo sottoinsieme (non banale) di nodi M tale che per ogni *n* in *M*, R*(n)=M*
  - in altre parole: partendo da un qualunque nodo di M, si possono raggiungere tutti i nodi di M e nessun nodo all'infuori di esso.



{ b,c,d } non è un knot



{ b,c,d } è un knot

Deadlock detection - Knot

- **Teorema**
  - dato un grafo di Holt con una sola richiesta sospesa per processo
  - se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili,
  - allora il grafo rappresenta uno stato di deadlock se e solo se esiste un knot

Deadlock recovery

- **Dopo aver rilevato un deadlock...**
- **... bisogna risolvere la situazione**
- **La soluzione può essere**
  - *manuale*
    - l'operatore viene informato e eseguirà alcune azioni che permettano al sistema di proseguire
  - *automatica*
    - il sistema operativo è dotato di meccanismi che permettono di risolvere in modo automatico la situazione, in base ad alcune politiche

Meccanismi per il deadlock recovery

- **Terminazione totale**
  - tutti i processi coinvolti vengono terminati
- **Terminazione parziale**
  - viene eliminato un processo alla volta, fino a quando il deadlock non scompare
- **Preemption**
  - una risorsa (o più di una, se necessario) viene sottratta ad uno dei processi coinvolti nel deadlock

Meccanismi per il deadlock recovery

- **Checkpoint/rollback**
  - lo stato dei processi viene periodicamente salvato su disco (*checkpoint*)
  - in caso di deadlock, si ripristina (*rollback*) uno o più processi ad uno stato precedente, fino a quando il deadlock non scompare

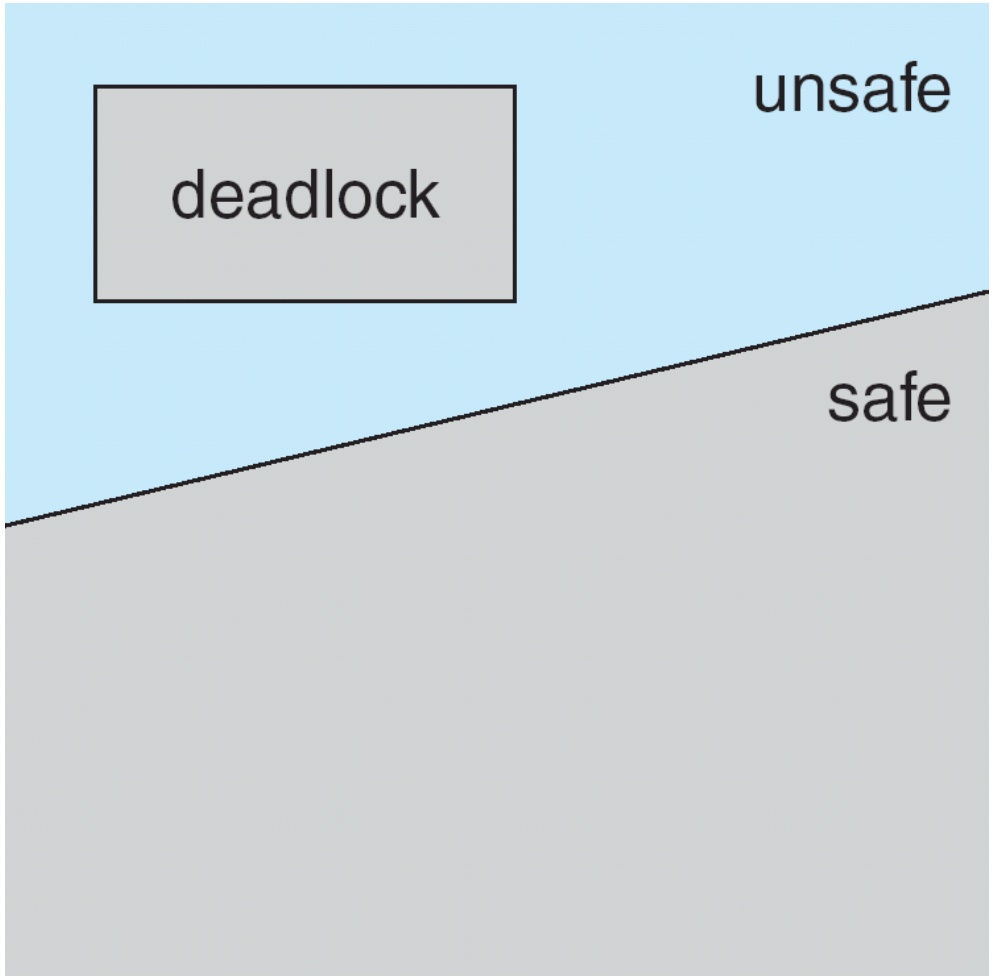
Considerazioni

- **Terminare processi può essere costoso**
  - questi processi possono essere stati eseguiti per molto tempo;
  - se terminati, dovranno ripartire da capo
- **Terminare processi può lasciare le risorse in uno stato inconsistente**
  - se un processo viene terminato nel mezzo di una sezione critica
- **Fare preemption**
  - non sempre è possibile
  - può richiedere interventi manuali
  - esempio: preemption di una stampante

Deadlock prevention / avoidance

- **Prevention**
  - per evitare il deadlock si elimina una delle quattro condizioni del deadlock
  - il deadlock viene eliminato *strutturalmente*
- **Avoidance**
  - prima di assegnare una risorsa ad un processo, si controlla se l'operazione può portare al pericolo di deadlock
  - in quest'ultimo caso, l'operazione viene ritardata

Deadlock prevention / avoidance



Deadlock prevention

- **Attaccare la condizione di "*Mutua esclusione*"**
  - permettere la condivisione di risorse
  - e.g. spool di stampa, tutti i processi "pensano" di usare contemporaneamente la stampante
- **Problemi dello spooling**
  - in generale, lo spooling non sempre è applicabile
    - ad esempio, descrittori di processi
  - si sposta il problema verso altre risorse
    - il disco nel caso di spooling di stampa
    - cosa succede se due processi che vogliono stampare due documenti esauriscono lo spazio su disco?

Deadlock prevention

- **Attaccare la condizione di "*Richiesta bloccante*"**
  - *Allocazione totale*
    - ogni processo deve richiedere tutte le risorse all'inizio della computazione
  - Problemi
    - non sempre l'insieme di richieste è noto fin dall'inizio
    - si riduce il parallelismo
- **Attaccare la condizione di "*Assenza di prerilascio*"**
  - come detto prima:
    - non sempre è possibile
    - può richiedere interventi manuali

Deadlock prevention

- **Attaccare la condizione di "*Attesa Circolare*"**
  - *Allocazione gerarchica*
    - alle classi di risorse vengono associati valori di priorità
    - ogni processo in ogni istante può allocare solamente risorse di priorità superiore a quelle che già possiede
    - se un processo vuole allocare una risorsa a priorità inferiore, deve prima rilasciare tutte le risorse con priorità uguale o superiore a quella desiderata

Deadlock prevention

- **Allocazione gerarchica e allocazione totale: problemi**
  - prevengono il verificarsi di deadlock, ma sono altamente inefficienti
- **Nell'allocazione gerarchica:**
  - l'indisponibilità di una risorsa ad alta priorità ritarda processi che già detengono risorse ad alta priorità
- **Nell'allocazione totale:**
  - anche se un processo ha necessità di risorse per poco tempo, deve allocarle per tutta la propria esistenza

Deadlock Avoidance

- **L'algoritmo del banchiere**
  - un algoritmo per evitare lo stallo sviluppato da Dijkstra (1965)
  - il nome deriva dal metodo utilizzato da un ipotetico banchiere di provincia che gestisce un gruppo di clienti a cui ha concesso del credito; non tutti i clienti avranno bisogno dello stesso credito simultaneamente

Algoritmo del banchiere

- **Descrizione**
  - un banchiere desidera condividere un capitale (fisso) con un numero (prefissato) di clienti
    - per Dijkstra l'"unità di misura" erano fiorini olandesi
  - ogni cliente specifica in anticipo la sua necessità massima di denaro
    - che ovviamente non deve superare il capitale del banchiere
  - i clienti fanno due tipi di transazioni
    - *richieste di prestito*
    - *restituzioni*

Algoritmo del banchiere

- **Descrizione**
  - il denaro prestato ad ogni cliente non può mai eccedere la necessità massima specificata a priori
  - ogni cliente può fare richieste multiple, fino al massimo importo specificato
  - una volta che le richieste sono state accolte e il denaro è stato ottenuto deve garantire la restituzione in un tempo finito

Algoritmo del banchiere

- **Metodo di funzionamento**
  - il banchiere deve essere in ogni istante in grado di soddisfare tutte le richieste dei clienti, o concedendo immediatamente il prestito oppure comunque facendo loro aspettare la disponibilità del denaro in un tempo finito

Algoritmo del banchiere

- n: numero di processi
- m: tipi di risorse disponibili
- Disponibili[m]: memorizza la quantità di ogni tipo di risorsa disponibile nel sistema. Se *Disponibili[j]=k*, ci sono k istanze della risorsa j-sima disponibili
- Max[m x n]: indica la quantità risorse che un processo allocherà al massimo. Se *Max*[*i,j*] = *k*, il processo i-simo può richiedere al massimo k istanze della risorsa j-sima
- Assegnate [m x n]: memorizza quante risorse per ogni tipo sono state allocate ad ognuno dei processi. Se *Assegnate[i,j] = k,* il processo i-simo ha k istanze allocate della risorsa j-sima.
- Necessità [m x n]: necessità residua di risorse per ogni processo. Se *Necessità*[*i,j*] = *k*, il processo i-simo potrà richiedere ancora k istanze della risorsa j-sima per completare il suo compito.

Necessità [i,j] = Max [i,j] – Assegnate [i,j]

Algoritmo del banchiere – verifica della safety

1) Siano *Work[m]* e *Finish[n]* due vettori di lunghezza *m* e *n*, rispettivamente.
Inizializziamo:

Work = *Disponibili*

Finish [*i*] = *false per i* = 0, 1, …, *n*- 1

2) Trovare un indice *i* tale che siano contemporaneamente valide (AND)*:*

(a) *Finish* [*i*] = *false*

(b) $Necessità_i \leq Work$ // $Necessità_i$ = riga i-sima della matrice Necessità

Se tale *i* non esiste, vai al passo 4;

3. *Work* = *Work* + $Assegnate_i$
Finish[*i*] = *true*
torna al passo 2

4. Se *Finish* [*i*] == *true* per ogni *i*, allora il sistema è in uno stato sicuro

Algoritmo del banchiere – richiesta di risorse

Request$_i$ = vettore delle richieste per il P_i.

Se ***Request***$_i$ **[*j*] = *k*** allora il processo P_i vuole ***k*** istanze della j-sima risorsa R_j

1. Se ***Request***$_i \leq$ ***Need***$_i$ vai al passo 2, ***altrimenti***, riporta un errore, perché il processo ha chiesto più risorse del massimo previsto
2. Se ***Request***$_i \leq$ ***Disponibili***, vai al passo 3, ***altrimenti*** P_i deve attendere perché le risorse non sono disponibili
3. Simula l'allocazione delle risorse a P_i modificando lo stato delle allocazioni come segue::

   Disponibili = Disponibili – Request$_i$***;***

   Assegnate$_i$ **= *Assegnate***$_i$ **+ *Request***$_i$**;**

   Necessità$_i$ **= *Necessità***$_i$ **– *Request***$_i$***;***

   - Se lo stato risultante è sicuro, le risorse vengono allocate a P_i
   - Se lo stato risultante è insicuro, P_i deve attendere

Algoritmo del banchiere – esempio

- 5 processi, da P_0 a P_4;

  3 tipi di risorse: *A* (10 istanze), *B* (5 istanze), *C* (7 istanze)

- Al tempo T_0 sia:

	Assegnate	*Max*	*Disponibilità*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
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	

Algoritmo del banchiere – esempio

- ***Necessità = Max – Assegnate***

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

- Il sistema è in uno stato sicuro, perché la sequenza $<P_1, P_3, P_4, P_2, P_0>$ soddisfa il criterio di sicurezza

Algoritmo del banchiere – esempio

- P_1 richiede (1,0,2)
  Bisogna verificare $Request_1 \leq Available$ (cioè, (1,0,2) ≤ (3,3,2) → true)

	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	

- L'algoritmo di verifica della safety mostra che la sequenza < P_1, P_3, P_4, P_0, P_2> porta il sistema in uno stato sicuro → la richiesta di P_1 può essere soddisfatta
- Se P_4 richiede (3,3,0)? → risorse non disponibili
- Se P_0 richiede (0,3,0)? → stato insicuro

Algoritmo dello struzzo

- **Algoritmo**
  - nascondere la testa sotto la sabbia, ovvero fare finta che i deadlock non si possano mai verificare
- **Motivazioni**
  - dal punto di vista ingegneristico, il costo di evitare i deadlock può essere troppo elevato
- **Esempi**
  - è la soluzione più adottata nei sistemi Unix
  - è usata anche nelle JVM