

# *Sistemi Operativi*

## *Modulo 4: Gestione risorse e deadlock* *A.A. 2021-22*

Renzo Davoli  
Alberto Montresor

Copyright © 2002-2022 Renzo Davoli, Alberto Montresor

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found at:  
<http://www.gnu.org/licenses/fdl.html#TOC1>

**"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 scorso*

# 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 non condivisibili (seriali)**
  - ♦ una singola risorsa non può essere assegnata a più processi contemporaneamente
  - ♦ esempi:
    - ♦ i processori
    - ♦ le sezioni critiche
    - ♦ le stampanti
- ♦ **Risorse condivisibili**
  - ♦ 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 che necessarie affinché un deadlock si presenti
- ♦ le tecniche che possono essere utilizzate per gestire il problema dei deadlock

# Condizioni per avere un deadlock

---

- ♦ **Mutua esclusione / non condivisibili**
  - ♦ le risorse coinvolte devono essere non condivisibili (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  $P_0, P_1, \dots, P_n$ , tali per cui  $P_0$  attende una risorsa controllata da  $P_1$ ,  $P_1$  attende una risorsa controllata da  $P_2, \dots$ , 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

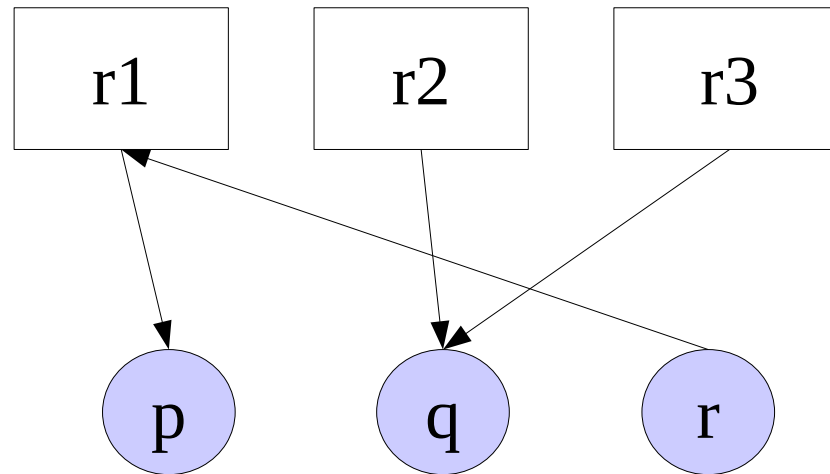
---

- ♦ **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

---



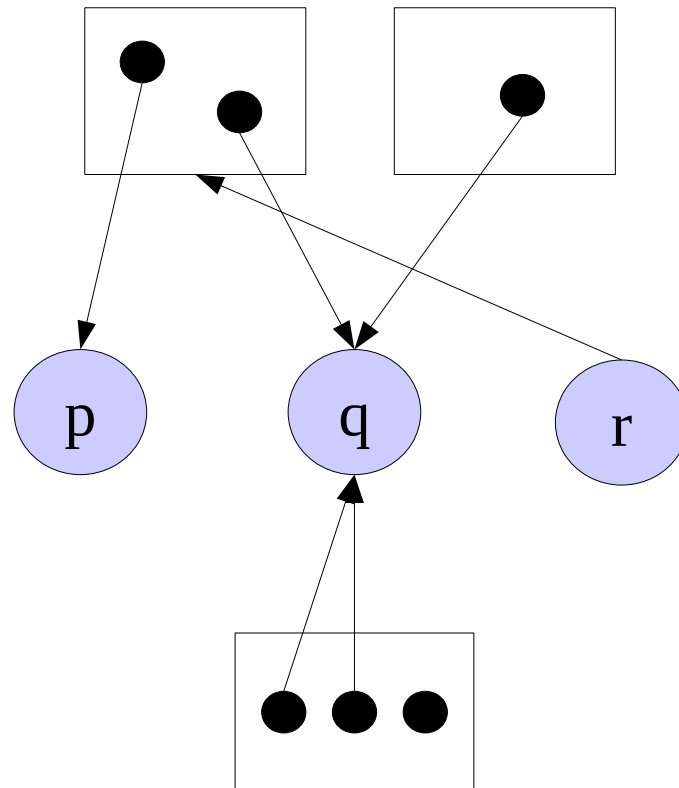


# Grafo di Holt generale



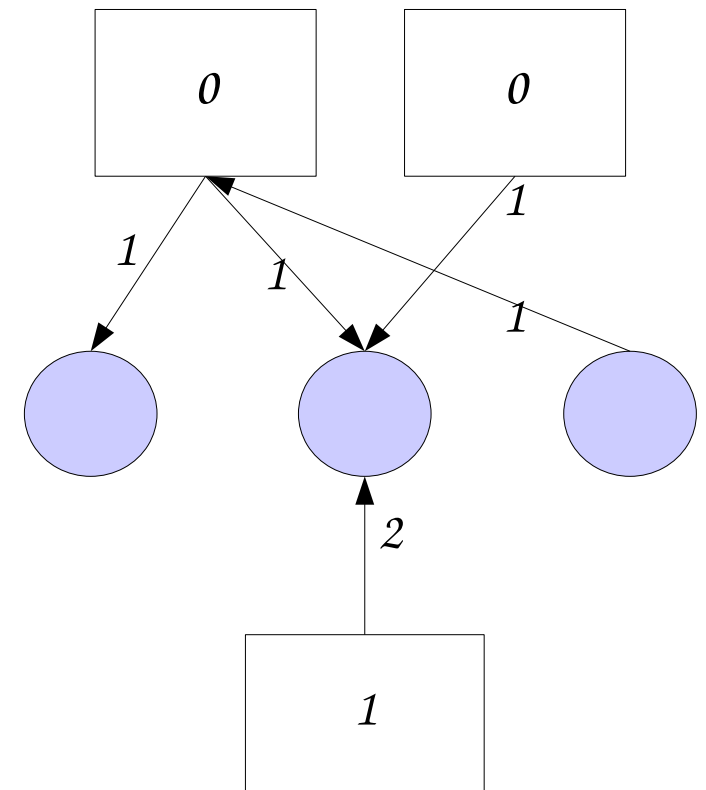
- ♦ **Nel caso di classi contenenti più istanze di una risorsa**
  - ♦ l'insieme delle risorse è partizionato in classi e gli archi di richiesta sono diretti alla classe e non alla singola risorsa
- ♦ **Rappresentazione**
  - ♦ i processi sono rappresentati da *cerchi*
  - ♦ le classi sono rappresentati 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 operativa

- **Nell'implementazione il grafo di Holt viene memorizzato come grafo pesato (con pesi ai nodi risorsa e pesi sugli archi)**
- **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 (Algoritmo dello struzzo)**
  - ♦ 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

---

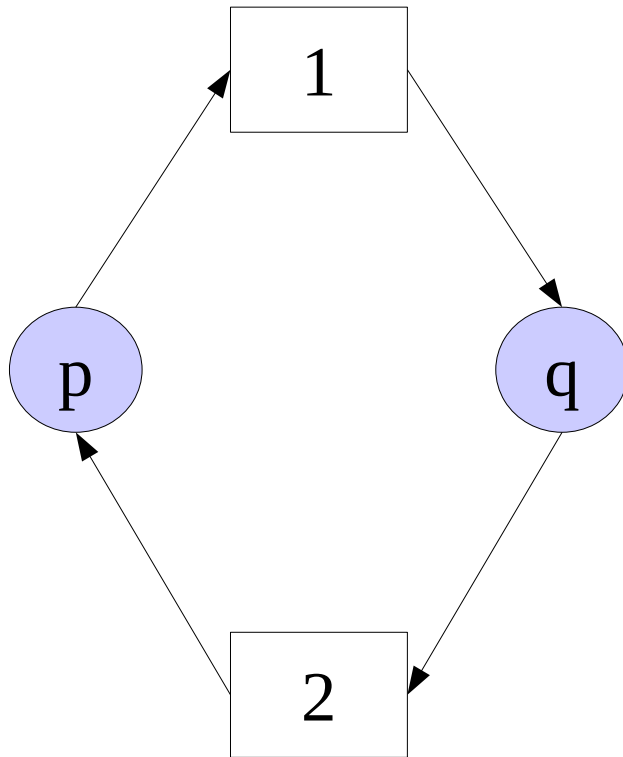
- ♦ **Teorema**

- ♦ se le risorse sono a richiesta bloccante, non condivisibili e non prerilasciabili
- ♦ *lo stato è di deadlock **se e solo se** il grafo di Holt contiene un ciclo*

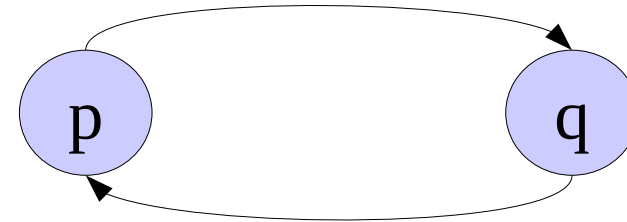
- ♦ **Dimostrazione**

- ♦ 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



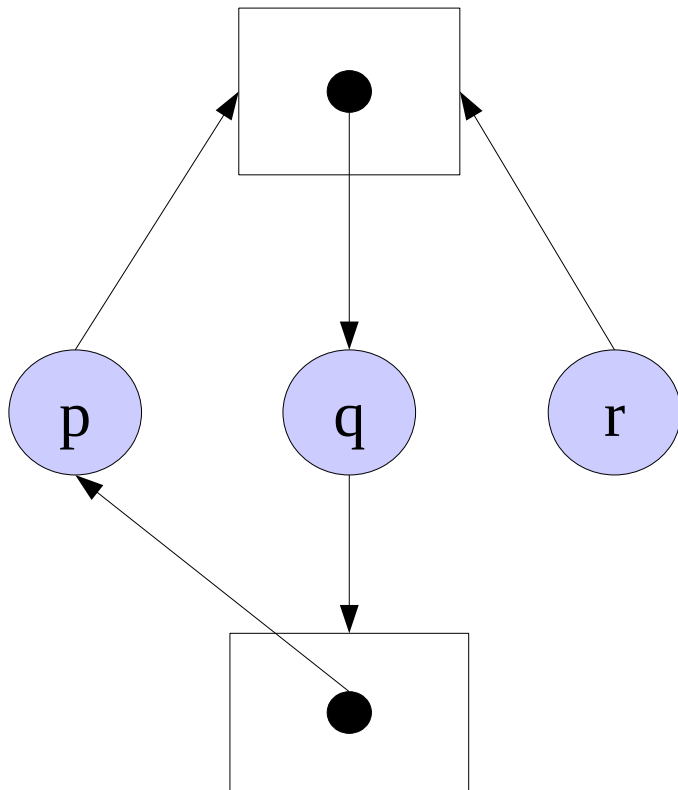
*Grafo di Holt*



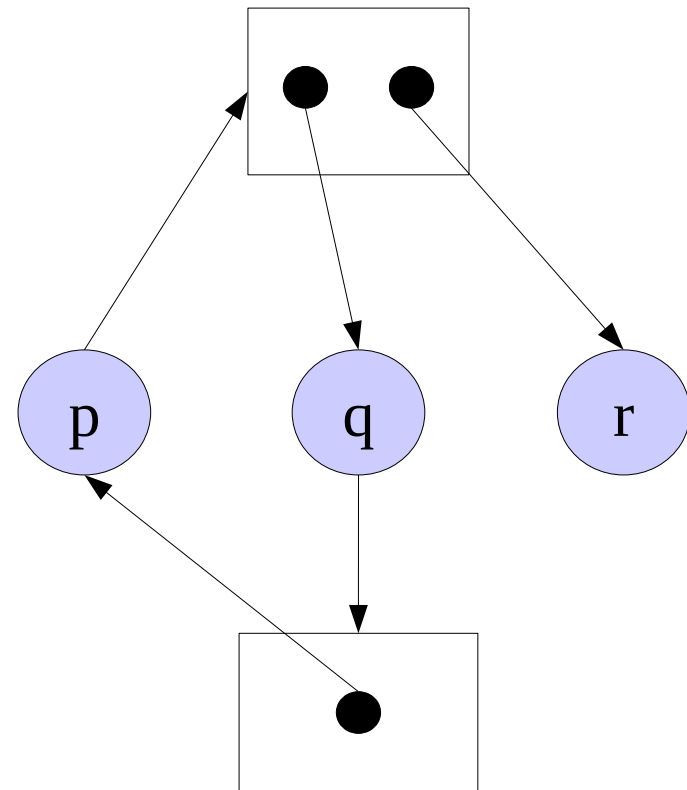
*Grafo Wait-For*

## 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

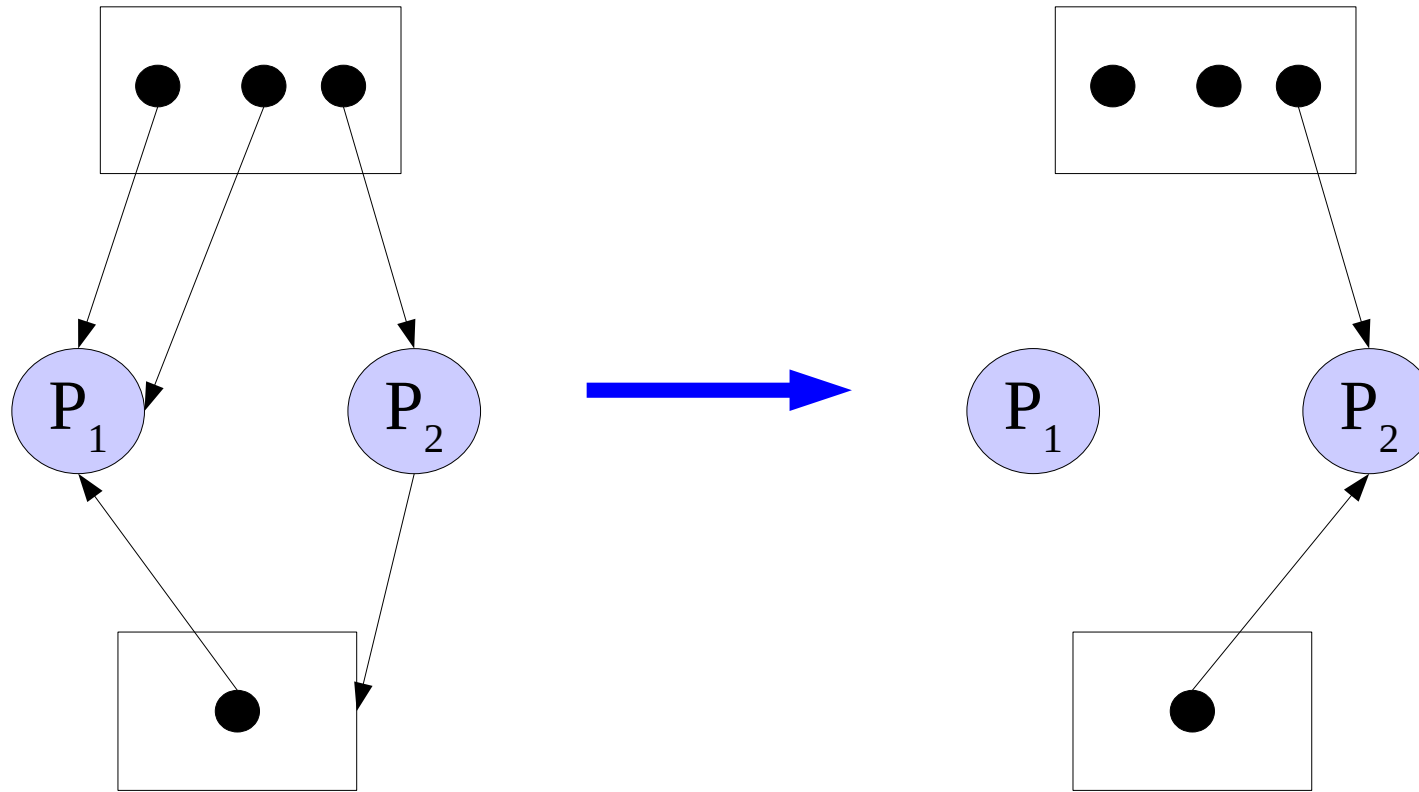
- ♦ **Riduzione**

- ♦ 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  $P_1$*

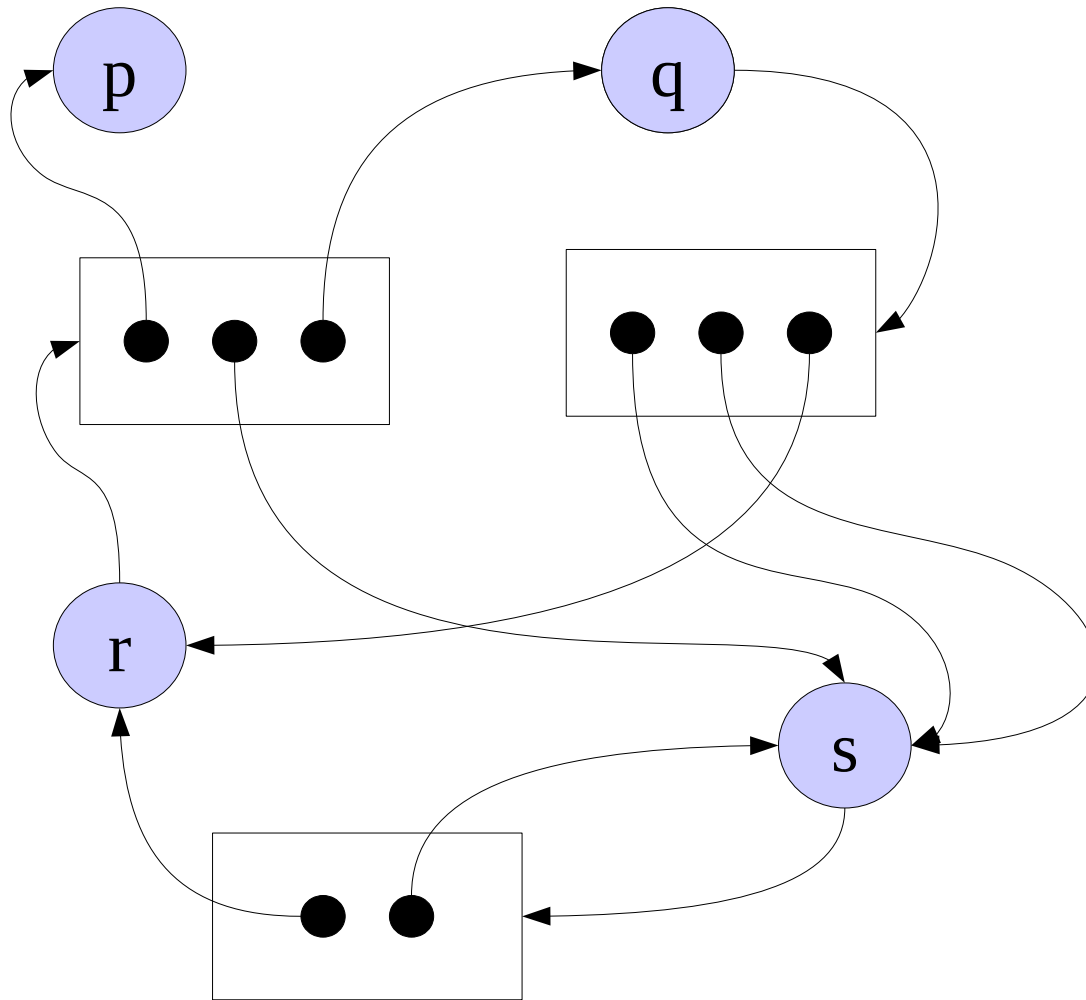
# Deadlock detection con grafo di Holt

---

- ♦ **Teorema**

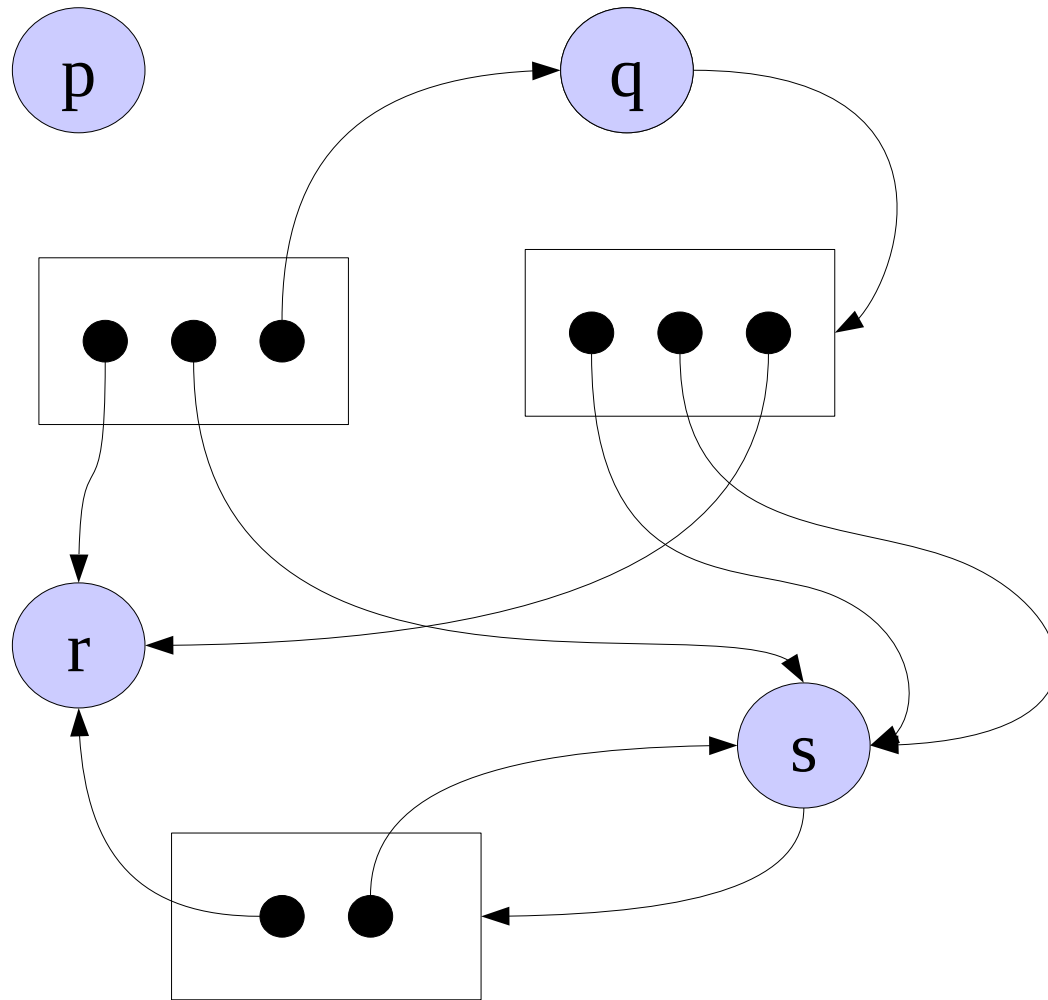
- ♦ se le risorse sono a richiesta bloccante, non condivisibili e non prerilasciabili
- ♦ lo stato non è di deadlock se e solo se il grafo di Holt è *completamente riducibile*, i.e. esiste una sequenza di passi di riduzione che elimina tutti gli archi del grafo

## Esercizio 2 - 4/6/2002



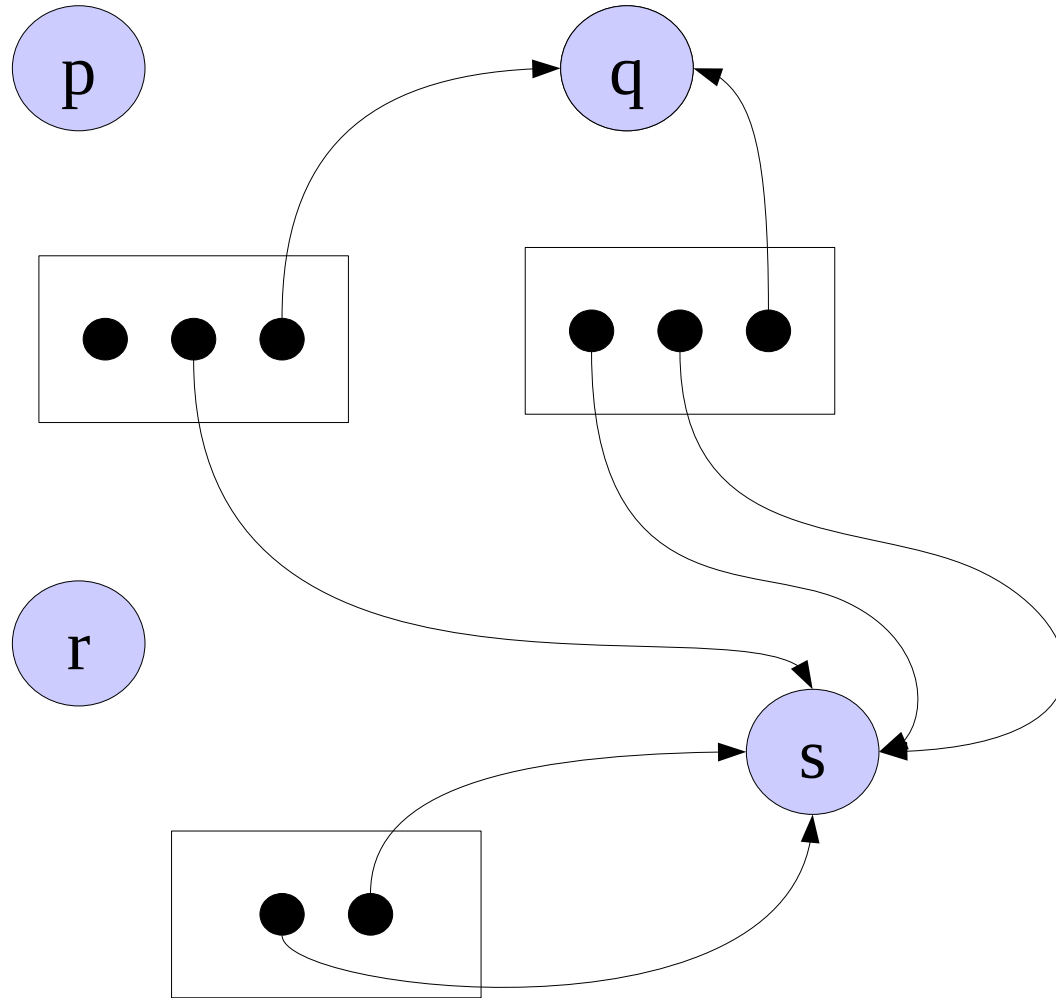
*C'è deadlock?*

## Esercizio 2 - 4/6/2002

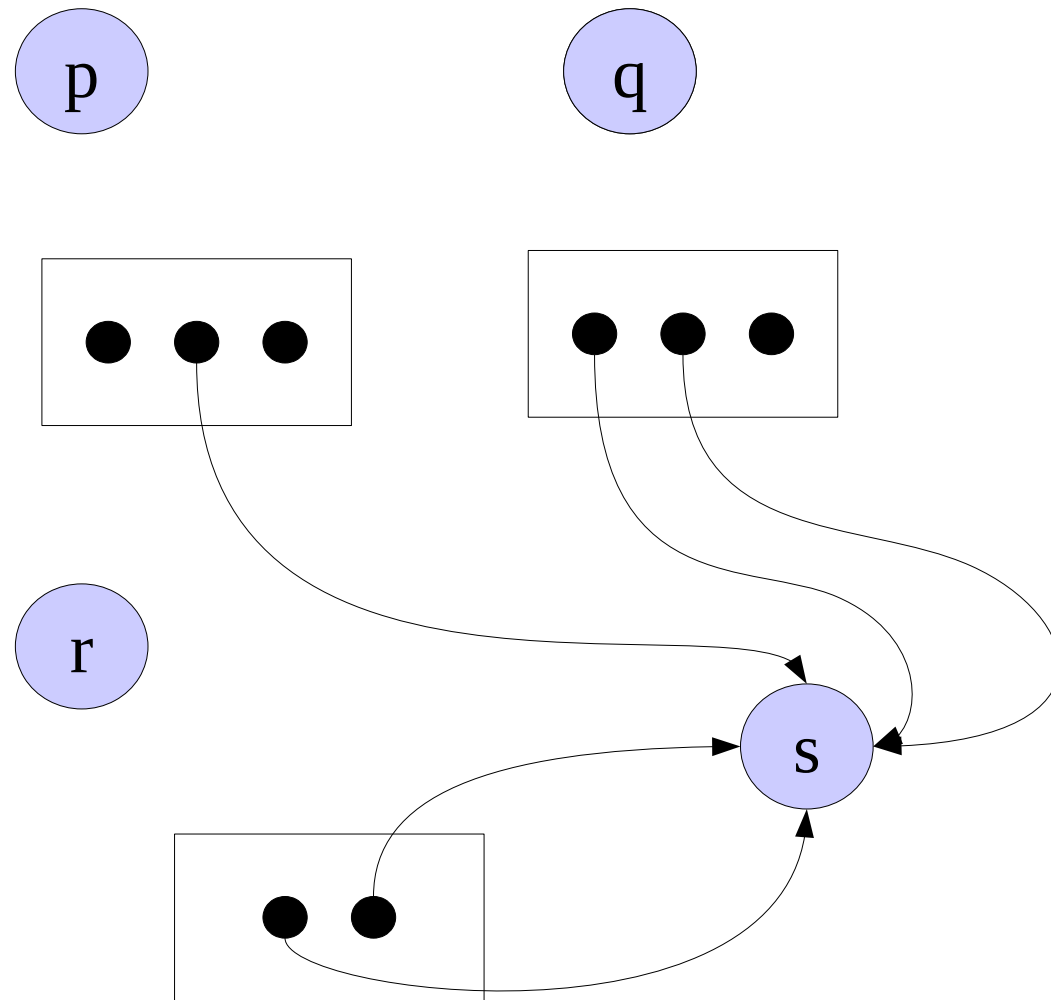


*Riduzione di p  
Assegnamento  
risorsa a r*

## Esercizio 2 - 4/6/2002



## Esercizio 2 - 4/6/2002

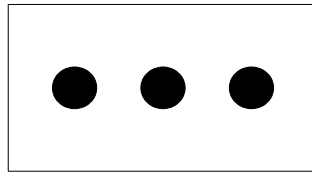
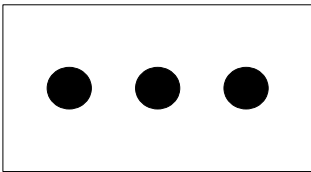


*Riduzione di  $q$*

## Esercizio 2 - 4/6/2002

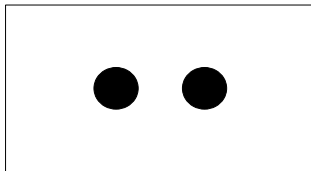
p

q



r

s



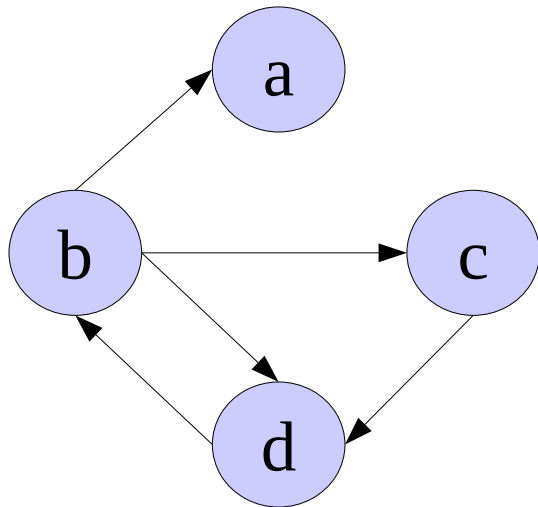
*Riduzione di s  
Non c'è deadlock*



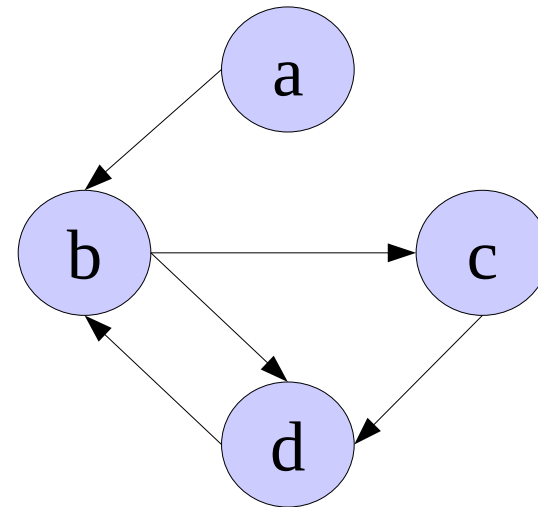
# Deadlock detection - Knot

## • 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 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 a richiesta bloccante, non condivisibili 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

# 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 incoerente**
  - ♦ se un processo viene terminato nel mezzo di una sezione critica

# 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

---

- ♦ **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*
    - ♦ è possibile richiedere che un processo richieda 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 allocarla 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 dei clienti
- **IC:** capitale iniziale
- **$c_i$ :** limite di credito del cliente  $i$  ( $c_i \leq IC$ )
- **$p_i$ :** denaro prestato al cliente  $i$  ( $p_i \leq c_i$ )
- **$n_i = c_i - p_i$**  credito residuo del cliente  $i$
- **$COH = IC - \sum_{i=1..N} p_i$**   
saldo di cassa



# Algoritmo del banchiere

- ♦ **Definizione: *Stato SAFE***

- ♦ sia  $s$  una permutazione dei valori  $1...N$ 
  - ♦ esempio, con  $N=4$ :  $s = 1, 3, 4, 2$
  - ♦ indichiamo con  $s(i)$  l' $i$ -esima posizione della sequenza

- ♦ si calcoli il vettore *avail* come segue

$$avail[1] = COH$$

$$avail[j+1] = avail[j] + p_{s(j)}, \text{ con } j=1...N-1$$

- ♦ uno stato del sistema si dice safe se vale la seguente condizione:

$$n_{s(j)} \leq avail[j], \text{ con } j=1...N$$

# Algoritmo del banchiere

---

- ♦ **Lo stato UNSAFE**
  - ♦ è condizione necessaria *ma non sufficiente* per avere deadlock
  - ♦ i.e., un sistema in uno stato UNSAFE può evolvere senza procurare alcun deadlock

# Algoritmo del banchiere - Situazione iniziale

<i>Capitale Iniziale</i>	<i>IC</i>	<i>150</i>
--------------------------	-----------	------------

<i>Saldo di cassa</i>	<i>COH</i>	<i>150</i>
-----------------------	------------	------------

<i>Cliente</i>	<i>MAX</i>	<i>Prestito Attuale</i>	<i>Credito residuo</i>
<i>i</i>	<i>c<sub>i</sub></i>	<i>p<sub>i</sub></i>	<i>n<sub>i</sub></i>
<i>1</i>	<i>100</i>	<i>0</i>	<i>100</i>
<i>2</i>	<i>20</i>	<i>0</i>	<i>20</i>
<i>3</i>	<i>30</i>	<i>0</i>	<i>30</i>
<i>4</i>	<i>50</i>	<i>0</i>	<i>50</i>
<i>5</i>	<i>70</i>	<i>0</i>	<i>70</i>

# Algoritmo del banchiere - Esempio stato SAFE

Capitale Iniziale	$IC$	150
----------------------	------	-----

Saldo di cassa	$COH$	0
-------------------	-------	---

la sequenza 3,2,1,4,5 consente il  
soddisfacimento di tutte le richieste

Cliente	$MAX$	Prestito Attuale	Credito residuo
$i$	$c_i$	$p_i$	$n_i$
1	100	70	30
2	20	10	10
3	30	30	0
4	50	10	40
5	70	30	40

# Algoritmo del banchiere

---

- ♦ **Regola pratica (per il banchiere a singola valuta)**
  - ♦ lo stato SAFE può essere verificato usando la sequenza che ordina in modo crescente i valori di  $n_i$
  - ♦ infatti, se esiste una sequenza di verificare la safety di uno stato, sicuramente anche la sequenza che ordina i valori di  $n_i$  consente di fare altrettanto

# Algoritmo del banchiere - Stato SAFE

Capitale Iniziale	$IC$	150
-------------------	------	-----

Cliente	$MAX$	Prestito Attuale	Credito residuo	
$i$	$c_i$	$p_i$	$n_i$	$avail[i]$

Saldo di cassa	$COH$	0
----------------	-------	---

la sequenza 3,2,1,4,5 consente il soddisfacimento di tutte le richieste

3	30	30	0	0
2	20	10	10	30
1	100	70	30	40
4	50	10	40	110
5	70	30	40	120

# Algoritmo del banchiere - Esempio Stato UNSAFE

Capitale Iniziale	$IC$	150
-------------------	------	-----

Saldo di cassa	$COH$	10
----------------	-------	----

Cliente	$MAX$	Prestito Attuale	Credito residuo
$i$	$c_i$	$p_i$	$n_i$
1	100	65	35
2	20	10	10
3	30	5	25
4	50	15	35
5	70	35	35

# Algoritmo del banchiere - Stato UNSAFE

Capitale Iniziale	$IC$	150
-------------------	------	-----

Ciente	$MAX$	Prestito Attuale	Credito residuo	
$i$	$c_i$	$p_i$	$n_i$	$avail[i]$

Saldo di cassa	$COH$	10
----------------	-------	----

se esistesse una sequenza,  
questa sarebbe 2,3,1,5,4

in corsivo sono indicati i casi  
nei quali la condizione di  
safety fallisce

2	20	10	10	10
3	30	5	25	20
1	100	65	35	25
5	70	35	35	100
4	50	15	35	135



# Algoritmo del banchiere - Stato UNSAFE

Capitale Iniziale	$IC$	150
-------------------	------	-----

Ciente	$MAX$	Prestito Attuale	Credito residuo	
$i$	$c_i$	$p_i$	$n_i$	$avail[i]$

Saldo di cassa	$COH$	0
----------------	-------	---

*UNSAFE non implica deadlock*

*se il cliente 5 restituisce il suo prestito di 35 euro la situazione ritorna SAFE*

2	20	10	10	45
3	30	5	25	55
1	100	75	35	60
4	50	15	35	135
5	70	0	70	150

# Algoritmo del banchiere

---

- ♦ **La similitudine fra banchieri e sistemi operativi ora è chiara...**
  - ♦ il denaro sono le risorse
  - ♦ il sistema le deve allocare ai processi senza che si possa verificare deadlock
  - ♦ le definizioni viste fino a questo punto riguardano il caso teorico elementare di un sistema avente un'unica classe di risorse
- ♦ **Algoritmo del banchiere multivaluta**
  - ♦ è l'estensione del problema del banchiere
  - ♦ si ipotizza che il banchiere debba fare prestiti usando valute diverse (euro, dollari, yen, etc.)
  - ♦ le diverse valute rappresentano diverse classi di risorse

# Algoritmo del banchiere multivaluta

- $N$ : numero dei clienti
- $\overline{IC}$ : capitale iniziale (*vettore, un elemento per ogni valuta*)
- $\overline{c}_i$ : limite di credito del cliente  $i$  ( $\overline{c}_i \leq \overline{IC}$ )
- $\overline{p}_i$ : denaro prestato al cliente  $i$  ( $\overline{p}_i \leq \overline{c}_i$ )
- $\overline{n}_i = \overline{c}_i - \overline{p}_i$  credito residuo del cliente  $i$
- $\overline{COH} = \overline{IC} - \sum_{i=1..N} \overline{p}_i$   
saldo di cassa in valuta  $k$

# Algoritmo del banchiere multivaluta

- ♦ **Definizione: *Stato SAFE***

- ♦ sia  $s$  una permutazione dei valori  $1...N$ 
  - ♦ esempio, con  $N=4$ :  $s = \langle 1, 3, 4, 2 \rangle$
  - ♦ indichiamo con  $s(i)$  l' $i$ -esima posizione della sequenza

- ♦ si calcoli il vettore  $avail_k$  come segue

$$\overline{avail[1]} = \overline{COH}$$

$$\overline{avail[j+1]} = \overline{avail[j]} + \overline{p_{s(j)}}$$
 con  $j=1...N-1$

- ♦ uno stato del sistema si dice safe se vale la seguente condizione:

$$\overline{n_{s(j)}} \leq \overline{avail[j]}, \text{ con } j=1...N$$

# Algoritmo del banchiere multivaluta

- ♦ **Problema**

- ♦ la regola di ordinare i processi secondo i valori di  $n_i$  non è applicabile
- ♦ l'ordine può essere in generale diverso fra le diverse valute gestite dal banchiere

- ♦ **Soluzione**

- ♦ si può creare la sequenza procedendo passo passo aggiungendo un processo a caso fra quelli completamente soddisfacibili
- ♦ ovvero, al passo  $j$  si sceglie quelli per cui

$$n_{s(j)} \leq \text{avail}[j]$$

# Teorema dell'algorithmo del banchiere

---

- ♦ **Teorema**

- ♦ se durante la costruzione della sequenza  $s$  si giunge ad un punto in cui nessun processo risulta soddisfacibile, lo stato non è SAFE, i.e. non esiste alcuna sequenza che consenta di soddisfare tutti i processi

- ♦ **Dimostrazione**

- ♦ per assurdo
- ♦ supponiamo che lo stato sia SAFE, ovvero che esista la sequenza che consente di soddisfare tutti i processi
- ♦ sia  $C$  la sequenza interrotta e  $C'$  la sequenza che porta allo stato SAFE

# Teorema dell'algorithmo del banchiere

Siano  $C$  e  $C'$  le sequenze di processi come in figura

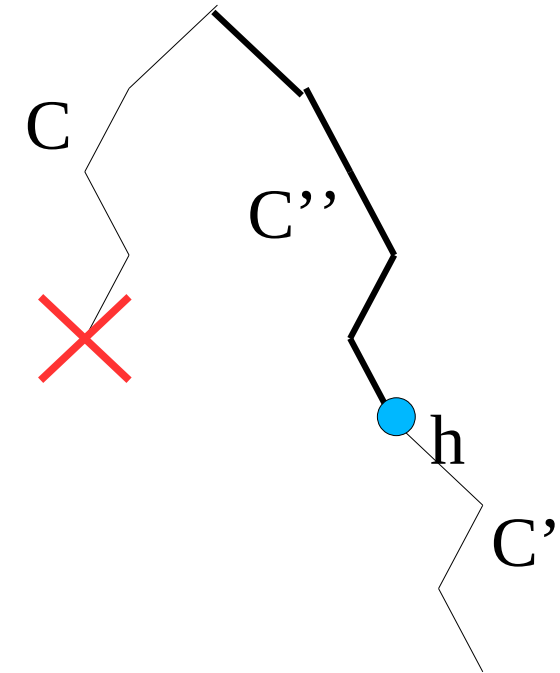
Sia  $H = \{p \in \text{processi} \mid p \notin C\}$

sia  $h$  il primo elemento di  $H$  che compare in  $C'$

tutti gli elementi di  $C'$  prima di  $h$  compaiono in  $C$ . Chiamiamo  $C''$  il segmento iniziale di  $C'$  fino al punto precedente l'applicazione di  $h$

le risorse disponibili all'applicazione di  $h$  in  $C'$  sono  $\text{avail}(C'') = \text{COH} + \sum_{j \in C''} p_j$ ;  $h$  è applicabile quindi  $n \leq \text{avail}(C'')$

ma  $\text{avail}(C) = \text{COH} + \sum_{j \in C} p_j$ , quindi se  $\text{avail}(C) \leq \text{avail}(C'')$  allora  $h$  è applicabile alla fine di  $C$  contro l'ipotesi che  $C$  non fosse ulteriormente estendibile



# 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