UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

Sistemi Operativi

Gestione dei Deadlock

LEZIONE 18

prof. Antonino Staiano

Corso di Laurea in Informatica – Università di Napoli Parthenope

antonino.staiano@uniparthenope.it

Introduzione

- Cosa è un deadlock?
- Deadlock nell'allocazione delle risorse
- Gestione dei deadlock
  - Individuazione e risoluzione dei deadlock
  - Prevenzione dei deadlock
  - Evitare i deadlock

Cosa è un Deadlock?

- Deadlock
  - Una situazione che coinvolge un insieme D di processi in cui ogni processo P_i in D soddisfa due condizioni:
    1. Il processo P_i è bloccato su qualche evento e_j
    2. L'evento e_j può essere solo causato da azioni di altri processi in D
- Deadlock di risorsa -> preoccupazione primaria di un SO

Process P_i	*Process P_j*
Request tape drive; Request printer; Use tape drive and printer; Release printer; Release tape drive;	Request printer; Request tape drive; Use tape drive and printer; Release tape drive; Release printer;

  - P_i e P_j sono in deadlock dopo le rispettive seconde richieste
- I deadlock possono anche verificarsi nella sincronizzazione e la comunicazione di messaggi <- **preoccupazione utente**

Deadlock nell'Allocazione di Risorse

- Un SO può contenere molte risorse di un certo tipo
  - Un'*unità di risorsa* si riferisce ad una risorsa di un tipo specifico
  - Una classe di risorse si riferisce all'insieme di tutte le unità di risorsa di un tipo
- Indichiamo con R_i una classe di risorse e con r_j un'unità di risorsa di una classe
- L'allocazione delle risorse in un sistema implica tre tipi di eventi:
  - Richiesta di una risorsa
  - Effettiva allocazione della risorsa
  - Rilascio della risorsa
    - La risorsa rilasciata può essere allocata ad un altro processo

Deadlock nell'Allocazione di Risorse (cont.)

- Eventi legati all'allocazione di risorse

Event	Description
Request	A process requests a resource through a system call. If the resource is free, the kernel allocates it to the process immediately; otherwise, it changes the state of the process to *blocked*.
Allocation	The process becomes the *holder* of the resource allocated to it. The resource state information is updated and the state of the process is changed to *ready*.
Release	A process releases a resource through a system call. If some processes are blocked on the allocation event for the resource, the kernel uses some tie-breaking rule, e.g., FCFS allocation, to decide which process should be allocated the resource.

Condizioni per il Deadlock di Risorsa

- Condizioni per il deadlock di risorsa
  - Non condivisibile
  - Non prelazionabile
  - Possesso e attesa
  - Attesa circolare

Condition	Explanation
Nonshareable resources	Resources cannot be shared; a process needs exclusive access to a resource.
No preemption	A resource cannot be preempted from one process and allocated to another process.
Hold-and-wait	A process continues to hold the resources allocated to it while waiting for other resources.
Circular waits	A circular chain of hold-and-wait conditions exists in the system; e.g., process P_i waits for P_j, P_j waits for P_k, and P_k waits for P_i.

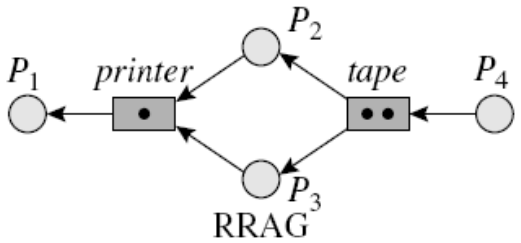
- E' essenziale anche un'altra condizione per il deadlock:
  - Nessun annullamento di richieste di risorse: un processo bloccato su una richiesta di risorsa non può annullarla

Modellare lo stato di Allocazione delle Risorse

- Stato di allocazione (**risorsa**):
  - Informazioni sulle risorse allocate ai processi e sulle richieste pendenti di risorse
  - Usato per determinare se un insieme di processi è in deadlock
- Sono usati due tipi di modelli per rappresentare lo stato di allocazione di un sistema:
  - Modello a grafo
    - Un processo può richiedere e usare esattamente un'unità di risorsa di ogni classe di risorse
  - Modello a matrice
    - Un processo può richiedere un qualsiasi numero di unità di una classe di risorsa

Grafo di Richiesta e di Allocazione di Risorsa (RRAG)

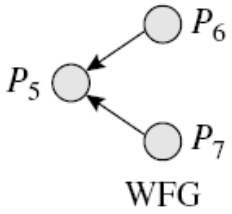
- I nodi e gli archi in un RRAG
  - Esistono due tipi di nodi in un RRAG
    - Un cerchio è un processo
    - Un rettangolo è una classe di risorse
      - Ciascun punto in un rettangolo è un'unità di risorsa
  - Gli archi possono anch'essi essere di due tipi
    - Un arco da una classe di risorse ad un processo è un'allocazione di risorsa
    - Un arco da un processo ad una classe di risorse è una richiesta pendente di risorsa
      - Il processo è bloccato sulla richiesta dell'unità di risorsa di quella classe



- Un arco di allocazione (R_k, P_j) è cancellato quando il processo P_j rilascia un'unità di risorsa della classe R_k
- Un arco di richiesta (P_i, R_k) è cancellato ed è aggiunto un arco di allocazione (R_k, P_i) quando è concessa una richiesta in attesa dal processo P_i per un'unità di una classe R_k

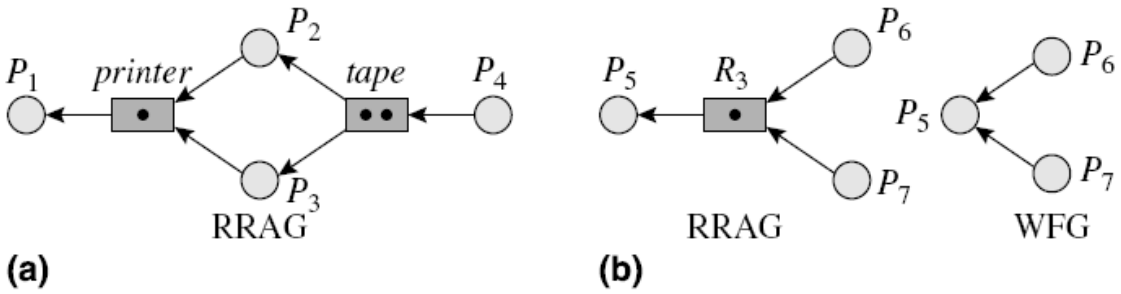
Grafo di Attesa (Wait-for Graph – WFG)

- Un WFG può essere usato per descrivere lo stato delle risorse di un sistema in cui ogni classe di risorsa contiene solo un'unità di risorsa
- Esiste un solo tipo di nodo che rappresenta un processo
- Un arco è una relazione wait-for tra processi
  - Un arco wait-for (P_i,P_j) indica che
    - Il processo P_j occupa l'unità di risorsa della classe di risorsa
    - Il processo P_i ha richiesto la risorsa e si è bloccato su di essa
    - In sostanza, P_i attende che P_j rilasci la risorsa



WFG

RRAG vs WFG



(a) Grafo di richiesta e allocazione di risorse (RRAG)

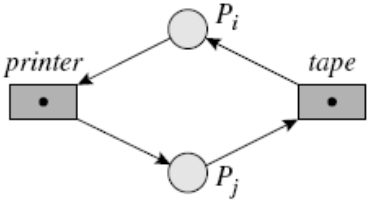
(b) Equivalenza di RRAG e WFG quando ogni classe di risorsa contiene una sola unità

Cammini nei WFG e RRAG

- Un cammino in un grafo è una sequenza di archi tali che il nodo destinazione di un arco è il nodo sorgente dell'arco seguente
  - Consideriamo un cammino in un RRAG: P_1 - R_1 - P_2 - R_2 … P_{n-1} - R_{n-1} - P_n Questo cammino indica che
    - Al processo P_n è stata allocata un'unità di R_{n-1}
    - Al processo P_{n-1} è stata allocata un'unità di R_{n-2} ed aspetta un'unità di R_{n-1}, ecc.
  - Nel WFG, lo stesso cammino sarebbe P_1 - P_2 - …P_{n-1} – P_n
- Supponiamo che ogni classe di risorsa contenga un'unica unità
  - I cammini P_1 - R_1 - P_2 - R_2 … P_{n-1} - R_{n-1} – P_n nel RRAG e P_1 - P_2 - …P_{n-1} – P_n nel WFG sono privi di deadlock

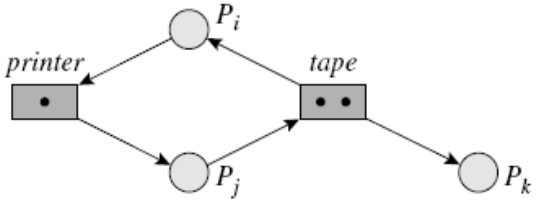
Modelli di Grafo (cont.)

- Non può esistere un deadlock se un RRAG o un WFG **NON** contiene un ciclo



- Un ciclo in un RRAG non implica necessariamente un deadlock se una classe di risorse ha unità multiple

Quando P_k finisce,
La sua unità a nastro
può essere allocata a P_j



Modello a Matrice

- Lo stato di allocazione è rappresentato da due matrici
  - Risorse allocate
  - Risorse richieste
- Se un sistema ha **n** processi e **r** classi di risorse, ciascuna di tali matrici ha dimensione **nxr**

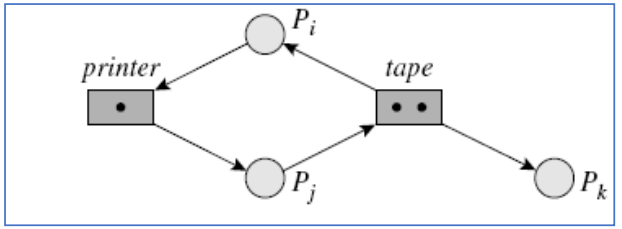
	Printer	Tape
P_i	0	1
P_j	1	0
P_k	0	1

Allocated resources

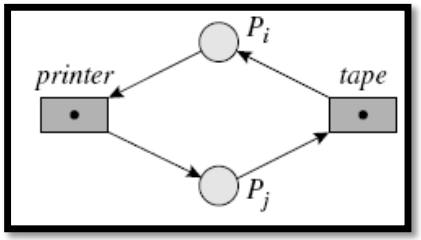
	Printer	Tape
P_i	1	0
P_j	0	1
P_k	0	0

Requested resources

	Printer	Tape
Total resources	1	2
Free resources	0	0



Gestione dei Deadlock



Approcci alla gestione dei deadlock

Approach	Description
Deadlock detection and resolution	The kernel analyzes the resource state to check whether a deadlock exists. If so, it aborts some process(es) and allocates the resources held by them to other processes so that the deadlock ceases to exist.
Deadlock prevention	The kernel uses a resource allocation policy that ensures that the four conditions for resource deadlocks mentioned in Table 8.2 do not arise simultaneously. It makes deadlocks impossible.
Deadlock avoidance	The kernel analyzes the allocation state to determine whether granting a resource request can lead to a deadlock in the future. Only requests that cannot lead to a deadlock are granted, others are kept pending until they can be granted. Thus, deadlocks do not arise.

Individuazione e Risoluzione dei Deadlock

- Un processo bloccato non è coinvolto in un deadlock se la richiesta su cui è bloccato può essere soddisfatta con la sequenza di eventi *completamento processo - rilascio risorsa - allocazione risorsa*
- Se ciascuna classe di risorsa in un sistema contiene una singola unità, il controllo avviene verificando la presenza di cicli nel RRAG o WFG
  - Non applicabile se le classi di risorse hanno più unità
    - sarebbero necessari algoritmi complessi per RRAG
- Useremo il modello a matrice
  - Applicabile in tutte le situazioni
  - Si cerca di costruire una sequenza ammissibile di eventi dove tutti i processi bloccati possono ottenere le risorse che hanno richiesto
    - Se il tentativo va a buon fine allora non c'è deadlock
    - Se il tentativo fallisce c'è deadlock

Esempio: Individuazione Deadlock

- Lo stato di allocazione di un sistema che contiene 10 unità di una classe di risorse R_1 e tre processi P_1, P_2 e P_3:

	R_1
P_1	4
P_2	4
P_3	2

Allocated resources

	R_1
P_1	6
P_2	2
P_3	0

Requested resources

	R_1
Total resources	10
Free resources	0

- Il processo P_3 è nello stato *running*
  - Simuliamo il completamento di P_3
    - Allocchiamo le sue risorse a P_2
  - Tutti i processi in questo modo possono completare
    - Non esistono processi bloccati quando la simulazione termina
      - Quindi nessun deadlock

Un Algoritmo di Individuazione Deadlock

Inputs

n	:	Number of processes;
r	:	Number of resource classes;
Blocked	:	**set of** processes;
Running	:	**set of** processes;
Free_resources	:	**array** [1..*r*] **of** *integer*;
Allocated_resources	:	**array** [1..*n*, 1..*r*] **of** *integer*;
Requested_resources	:	**array** [1..*n*, 1..*r*] **of** *integer*;

Data structures

Finished	:	**set of** *processes*;

1. **repeat until** *set Running is empty*
   - **a.** *Select a process* P_i *from set Running*;
   - **b.** *Delete* P_i *from set Running and add it to set Finished*;
   - **c.** **for** $k = 1..r$
     $Free\_resources[k] := Free\_resources[k] + Allocated\_resources[i,k]$;
   - **d.** **while** *set Blocked contains a process* P_l *such that*
     for $k = 1..r$, $Requested\_resources[l,k] \leq Free\_resources[k]$
     - **i.** **for** $k = 1, r$
       $Free\_resources[k] := Free\_resources[k] - Requested\_resources[l,k]$;
       $Allocated\_resources[l,k] := Allocated\_resources[l,k] + Requested\_resources[l,k]$;
     - **ii.** *Delete* P_l *from set Blocked and add it to set Running*;
2. **if** *set Blocked is not empty* **then**
   declare processes in set Blocked to be deadlocked.

Esempio

	R_1	R_2	R_3
P_1	2	1	0
P_2	1	3	1
P_3	0	1	1
P_4	1	2	2

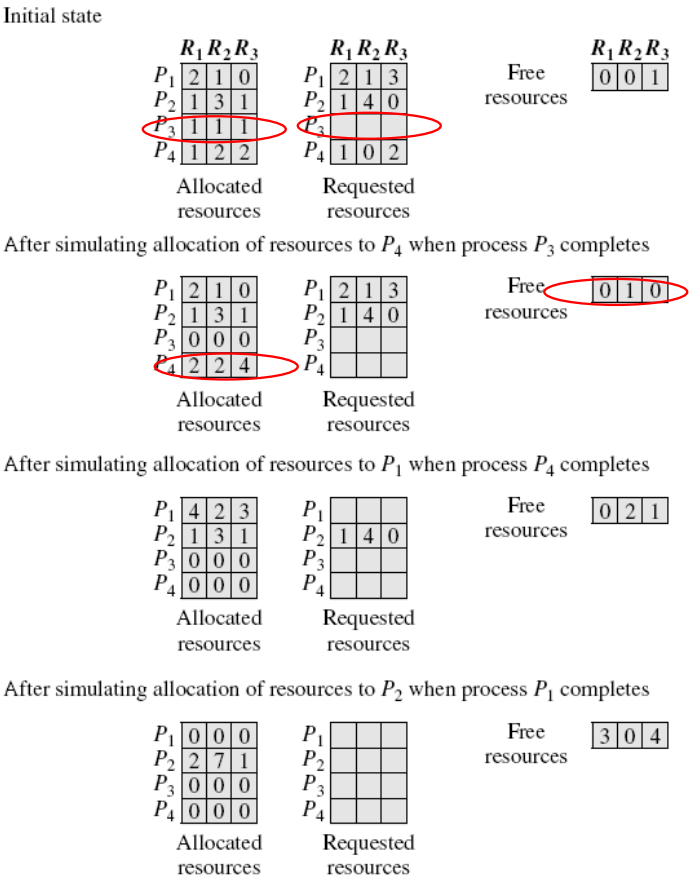
Allocated resources

	R_1	R_2	R_3
P_1	2	1	3
P_2	1	4	0
P_3			
P_4	1	0	2

Requested resources

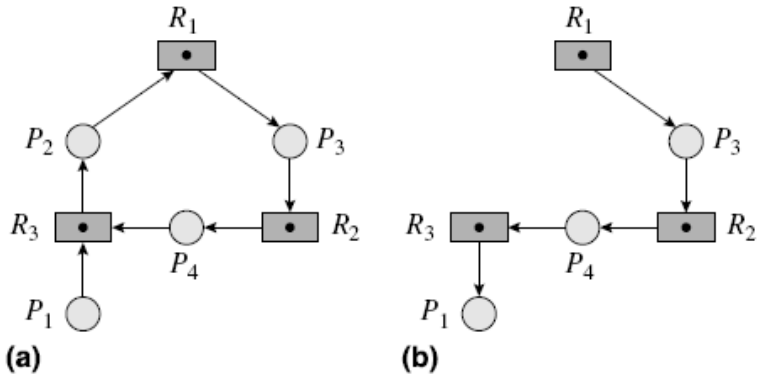
	R_1	R_2	R_3
Total resources	5	7	5
Free resources	1	0	1

Stato prima che P_3 faccia richiesta di 1 unità di R_1



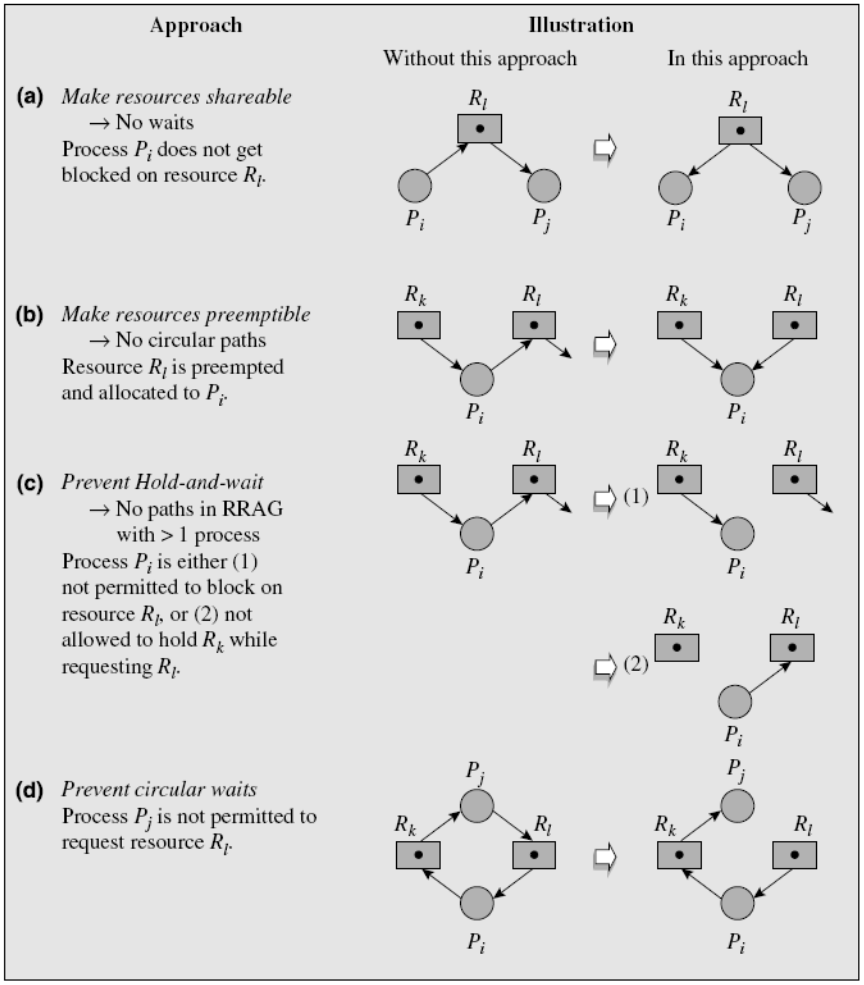
Risoluzione Deadlock

- La risoluzione del deadlock per un insieme D di processi in deadlock consiste nello spezzare il deadlock per assicurare il progresso per alcuni di essi
  - Si ottiene forzando la terminazione di uno o più processi in D
    - Ogni processo terminato è chiamato *vittima*
      - Le risorse della vittima sono allocate ad altri processi
    - La scelta della vittima è fatta sulla base di criteri quali priorità processo, risorse consumate dal processo, ecc.



Risoluzione deadlock. (a) un deadlock (b) stato allocazione risorse dopo la risoluzione del deadlock

Prevenzione Deadlock



Prevenzione deadlock: Allocazione simultanea

- E' la più semplice strategia di prevenzione deadlock
- Un processo deve chiedere tutte le risorse di cui necessita con un'unica richiesta
  - Il kernel le alloca tutte insieme al processo richiedente
    - Un processo blocked non possiede alcuna risorsa
      - La condizione Hold-and-wait non è mai soddisfatta
- Strategia interessante per piccoli SO
- Ha un unico difetto da un punto di vista pratico:
  - Pregiudica l'efficienza delle risorse

Prevenzione deadlock: Ranking delle risorse

- Ad ogni classe di risorse è associato un *rank di risorsa*
- Alla richiesta di risorsa, il kernel applica un **vincolo di validità** per decidere se soddisfare la richiesta
  - Il rank della risorsa richiesta deve essere maggiore del rank della risorsa con rank più elevato correntemente allocata al processo -> vincolo di validità TRUE
    - La risorsa è allocata al processo, se disponibile, altrimenti il processo è bloccato in attesa che la risorsa sia rilasciata
  - Se il vincolo è FALSE, la richiesta è rifiutata ed il processo richiedente è interrotto
- Risultato: assenza di attesa circolare
- Funziona al meglio quando tutti i processi richiedono le rispettive risorse in ordine crescente di rank di risorsa
  - Nel caso peggiore, la strategia può degenerare nella strategia di "allocazione simultanea"

Evitare i deadlock

- Algoritmo del Banchiere
  - Analogia: i banchieri ammettono i prestiti che collettivamente superano i fondi della banca e quindi concedono il prestito di ciascun mutuatario a rate
  - Usa la nozione di stato di allocazione sicuro
    - Quando il sistema è in questo stato, tutti i processi possono completare le proprie operazioni senza la possibilità di deadlock
  - Implementato portando il sistema da un'allocazione sicura ad un'altra

Evitare i deadlock

Notazioni dell'algoritmo del Banchiere

Notation	Explanation
$Requested\_resources_{j,k}$	Number of units of resource class R_k currently requested by process P_j
$Max\_need_{j,k}$	Maximum number of units of resource class R_k that may be needed by process P_j
$Allocated\_resources_{j,k}$	Number of units of resource class R_k allocated to process P_j
$Total\_alloc_k$	Total number of allocated units of resource class R_k, i.e., $\Sigma_j\ Allocated\_resources_{j,k}$
$Total\_resources_k$	Total number of units of resource class R_k existing in the system

Stato di allocazione sicuro: è uno stato di allocazione in cui è possibile costruire una sequenza di eventi *completamento processo*, *rilascio risorsa* e *allocazione risorsa* con cui ogni processo P_j nel sistema può ottenere $Max\_need_{j,k}$ risorse per ogni classe di risorsa R_k e completare le proprie operazioni.

Evitare i deadlock (cont.)

- Schema dell'approccio:
  1. Quando un processo fa una richiesta, fa una *proiezione dello stato di allocazione*
     - Sarebbe lo stato se la richiesta fosse soddisfatta
  2. Se tale proiezione è sicura, concede le risorse e aggiorna *Allocated_resources* e *Total_alloc*; altrimenti, mantiene la richiesta pendente
     a) La sicurezza è verificata con una simulazione (cerca di costruire una sequenza **completamento-rilascio-allocazione** con cui tutti i processi possono terminare)
     b) E' assunto che un processo completi le sue operazioni solo se può prendere il massimo richiesto di ciascuna risorsa simultaneamente, ovvero, per tutti i k

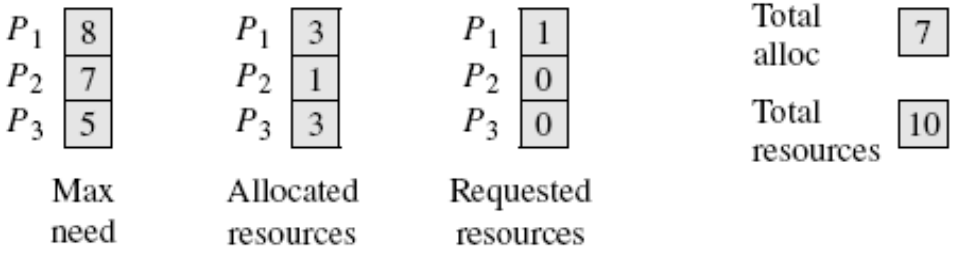
$$Total\_resource_k - Total\_alloc_k >= Max\_need_{l,k} - Allocated\_resource_{l,k}$$

  3. Quando un processo rilascia una qualsiasi risorsa o termina le operazioni, si esaminano le richieste pendenti e si allocano quelle che pongono il sistema in un nuovo stato sicuro

Esempio: algoritmo del Banchiere per una sola classe di risorse

Un sistema contiene 10 unità di risorse di una singola classe R_k

- Il requisito di risorse massime dei tre processi è 8, 7, 5 e l'allocazione corrente è 3, 1, 3
- P1 fa una richiesta di una risorsa, quindi in total_alloc ci saranno 8 risorse



Soddisfatta la richiesta di P_1 la nuova allocazione di risorse è **4, 1, 3**

- Consideriamo ora le seguenti richieste:
  1. P_1 fa una richiesta per 2 unità di risorsa
  2. P_2 fa una richiesta per 2 unità di risorsa
  3. P_3 fa una richiesta per 2 unità di risorsa
  - Le richieste di P_1 e P_2 non pongono il sistema in uno stato di allocazione sicuro, quindi non saranno soddisfatte
  - La richiesta di P_3 sarà soddisfatta

Total_resource(k) – Total_alloc(k) >= Max_need(I,k) – Allocated_resource(I,k)

Algoritmo del Banchiere

Inputs

n	:	Number of processes;
r	:	Number of resource classes;
Blocked	:	**set of** processes;
Running	:	**set of** processes;
$P_{requesting\_process}$	:	Process making the new resource request;
Max_need	:	**array** $[1..n, 1..r]$ **of** *integer*;
Allocated_resources	:	**array** $[1..n, 1..r]$ **of** *integer*;
Requested_resources	:	**array** $[1..n, 1..r]$ **of** *integer*;
Total_alloc	:	**array** $[1..r]$ **of** *integer*;
Total_resources	:	**array** $[1..r]$ **of** *integer*;

Data structures

Active	:	**set of** *processes*;
feasible	:	*boolean*;
New_request	:	**array** $[1..r]$ **of** *integer*;
Simulated_allocation	:	**array** $[1..n, 1..r]$ **of** *integer*;
Simulated_total_alloc	:	**array** $[1..r]$ **of** *integer*;

1. $Active := Running \bigcup Blocked;$
for $k = 1..r$
$New\_request[k] := Requested\_resources[requesting\_process, k];$

Dopo step 1 l'algoritmo è invocato con id processo che ha fatto la richiesta

Algoritmo del Banchiere

```
2. Simulated_allocation := Allocated_resources;
   for k = 1..r      /* Compute projected allocation state */
       Simulated_allocation[requesting_process, k] :=
               Simulated_allocation[requesting_process, k] + New_request[k];
       Simulated_total_alloc[k] := Total_alloc[k] + New_request[k];
3. feasible := true;
   for k = 1..r      /* Check whether projected allocation state is feasible */
       if Total_resources[k] < Simulated_total_alloc[k] then feasible := false;
4. if feasible = true
   then      /* Check whether projected allocation state is a safe allocation state */
       while set Active contains a process P_l such that
                   For all k, Total_resources[k] − Simulated_total_alloc[k]
                       ≥ Max_need[l, k] − Simulated_allocation[l, k]
           Delete P_l from Active;
           for k = 1..r
               Simulated_total_alloc[k] :=
                   Simulated_total_alloc[k] − Simulated_allocation[l, k];
5. if set Active is empty
   then      /* Projected allocation state is a safe allocation state */
       for k = 1..r      /* Delete the request from pending requests */
           Requested_resources[requesting_process, k] := 0;
           for k = 1..r      /* Grant the request */
               Allocated_resources[requesting_process, k] :=
                   Allocated_resources[requesting_process, k] + New_request[k];
               Total_alloc[k] := Total_alloc[k] + New_request[k];
```

Esempio: algoritmo del Banchiere per più classi di risorse

P2 ha effettuato la richiesta (0,1,1,0) che l'algoritmo deve elaborare

Total_resource(k) – Total_alloc(k) >= Max_need(I,k) – Allocated_resource(I,k)

(a) State after Step 1

	R_1	R_2	R_3	R_4
P_1	2	1	2	1
P_2	2	4	3	2
P_3	5	4	2	2
P_4	0	3	4	1

Max need

	R_1	R_2	R_3	R_4
P_1	1	1	1	1
P_2	2	0	1	0
P_3	2	0	2	2
P_4	0	2	1	1

Allocated resources

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	1	1	0
P_3	0	0	0	0
P_4	0	0	0	0

Requested resources

	R_1	R_2	R_3	R_4
Total alloc	5	3	5	4
Total exist	6	4	8	5

Active $\{P_1, P_2, P_3, P_4\}$

(b) State before while loop of Step 4

	R_1	R_2	R_3	R_4
P_1	2	1	2	1
P_2	2	4	3	2
P_3	5	4	2	2
P_4	0	3	4	1

Max need

	R_1	R_2	R_3	R_4
P_1	1	1	1	1
P_2	2	1	2	0
P_3	2	0	2	2
P_4	0	2	1	1

Simulated allocation

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	1	1	0
P_3	0	0	0	0
P_4	0	0	0	0

Requested resources

	R_1	R_2	R_3	R_4
Simulated total_alloc	5	4	6	4
Total exist	6	4	8	5

Active $\{P_1, P_2, P_3, P_4\}$

Esempio: algoritmo del Banchiere per più classi di risorse

```
4. if feasible = true
   then      /* Check whether projected allocation state is a safe allocation state */
      while set Active contains a process P_l such that
                  For all k, Total_resources[k] − Simulated_total_alloc[k]
                     ≥ Max_need[l, k] − Simulated_allocation[l, k]
         Delete P_l from Active;
         for k = 1..r
            Simulated_total_alloc[k] :=
               Simulated_total_alloc[k] − Simulated_allocation[l, k];
```

Total_resource(k) – Total_alloc(k) >= Max_need(l,k) – Allocated_resource(l,k)

(c) State after simulating completion of Process P_1

	Max need	Simulated allocation	Requested resources
P_1	2 1 2 1	1 1 1 1	0 0 0 0
P_2	2 4 3 2	2 1 2 0	0 1 1 0
P_3	5 4 2 2	2 0 2 2	0 0 0 0
P_4	0 3 4 1	0 2 1 1	0 0 0 0

Simulated total_alloc: 4 3 5 3

Total exist: 6 4 8 5

Active: $\{P_2, P_3, P_4\}$

(d) State after simulating completion of Process P_4

	Max need	Simulated allocation	Requested resources
P_1	2 1 2 1	1 1 1 1	0 0 0 0
P_2	2 4 3 2	2 1 2 0	0 1 1 0
P_3	5 4 2 2	2 0 2 2	0 0 0 0
P_4	0 3 4 1	0 2 1 1	0 0 0 0

Simulated total_alloc: 4 1 4 2

Total exist: 6 4 8 5

Active: $\{P_2, P_3\}$

(e) State after simulating completion of Process P_2

	Max need	Simulated allocation	Requested resources
P_1	2 1 2 1	1 1 1 1	0 0 0 0
P_2	2 4 3 2	2 1 2 0	0 1 1 0
P_3	5 4 2 2	2 0 2 2	0 0 0 0
P_4	0 3 4 1	0 2 1 1	0 0 0 0

Simulated total_alloc: 2 0 2 2

Total exist: 6 4 8 5

Active: $\{P_3\}$

Figure 8.8 Operation of the banker's algorithm for Example 8.11.

Algoritmo del Banchiere con allocazioni parametriche

- Supponiamo di avere 5 processi, P_0, P_1, P_2, P_3, e P_4 e 4 classi di risorse A, B, C e D, nella seguente configurazione

Processi/Risorse	A	B	C	D
P0	6	4	5	6
P1	10	7	6	8
P2	6	2	0	8
P3	0	3	4	2
P4	9	1	6	9

Max Risorse

Processi/Risorse	A	B	C	D
P0	4	X-1	3	2
P1	8	0	Y-2	2
P2	4	0	0	0
P3	0	0	3	2
P4	2	1	Z+1	4

Risorse allocate

A	B	C	D
2	2	10	4

Risorse disponibili

- Determinare
  - gli intervalli di X, Y e Z per cui il sistema si trova in uno stato sicuro e l'eventuale sequenza sicura
  - Se la richiesta di risorse (2, 0, 0, 2) di P_2 può essere soddisfatta

Algoritmo del Banchiere con allocazioni parametriche (cont.)

- L'obiettivo è determinare se esiste una sequenza sicura in funzione di X, Y e Z
- Se esaminiamo la condizione

$$\text{Risorse\_disponibili}_k >= \text{Max\_risorse}_{I,k} - \text{Risorse\_Allocate}_{I,k}$$

l'unico processo che può essere soddisfatto è P_0 ed in particolare

$$\begin{cases} 4-(X-1) \leq 2 \\ 4-(X-1) \geq 0 \\ X-1 \geq 0 \end{cases}$$ da cui si ricava $3 \leq X \leq 5$,

Dopo che P_0 termina l'esecuzione rilascia le sue risorse e le risorse disponibili diventano

[6, X + 1, 13, 6]

Algoritmo del Banchiere con allocazioni parametriche (cont.)

- A questo punto può essere eseguito solo P_3. Le risorse disponibili al termine di P_3 sono:

  [6, X + 1, 16, 8]

- Ora si può soddisfare solo P_2. Al termine di P_2 le risorse disponibili diventano

  [10, X + 1, 16, 8]

- Ora P_4, che può essere eseguito se

$$\begin{cases} 6-(Z+1) \leq 16 \\ 6-(Z+1) \geq 0 \\ Z+1 \geq 0 \end{cases}$$ da cui si ricava $-1 \leq Z \leq 5$,

- Al rilascio delle risorse di P_4 dopo che è terminato, le risorse disponibili diventano

  [12, X + 2, Z + 17, 12]

Algoritmo del Banchiere con allocazioni parametriche (cont.)

- Infine, resta P_1:

$$\begin{cases} 6-(Y-2) \le Z+17 \\ 6-(Y-2) \ge 0 \\ Y-2 \ge 0 \end{cases} \quad \text{da cui si ricava} \quad 2 \le Y \le 8,$$

- In definitiva, la sequenza sicura è

P_0, P_3, P_2, P_4, P_1 con $3 \le X \le 5$, $-1 \le Z \le 5$, $2 \le Y \le 8$

- $P_2[2,0,0,2]$ non può essere soddisfatta inizialmente, ma solo dopo che P_0 e P_3 hanno rilasciato le loro risorse