

# Deadlock



Corso di Laurea in Ingegneria Informatica  
Università degli Studi di Napoli Federico II  
Anno Accademico 2024/2025, Canale San Giovanni



# Deadlock

## ● Sommario

- Deadlock: definizioni e generalità
- Trattamento del deadlock:
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection

## ● Riferimenti

- Ancillotti Boari, "Sistemi Operativi", par 3.7
- [www.ostep.org](http://www.ostep.org), Cap. 32
- Stallings, "Operating Systems" 6th ed., par. 6.3

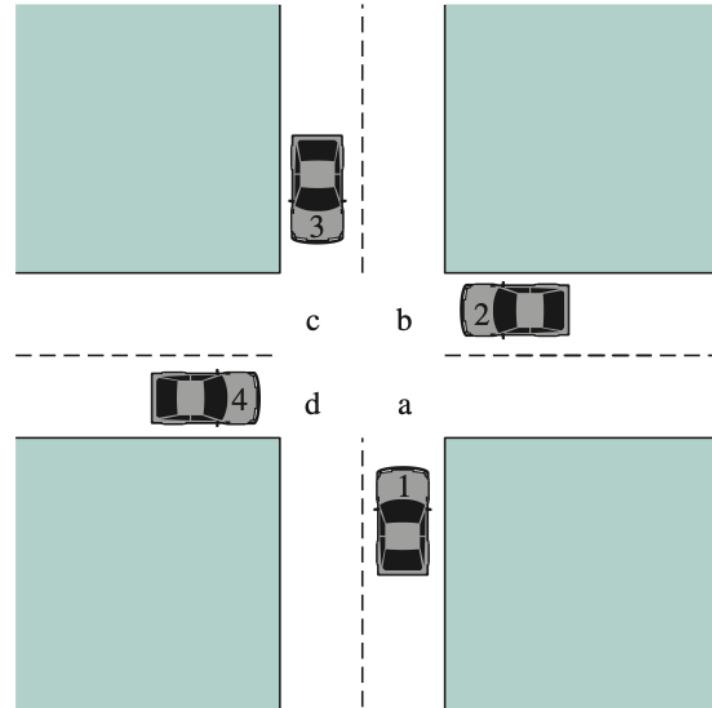
# Deadlock



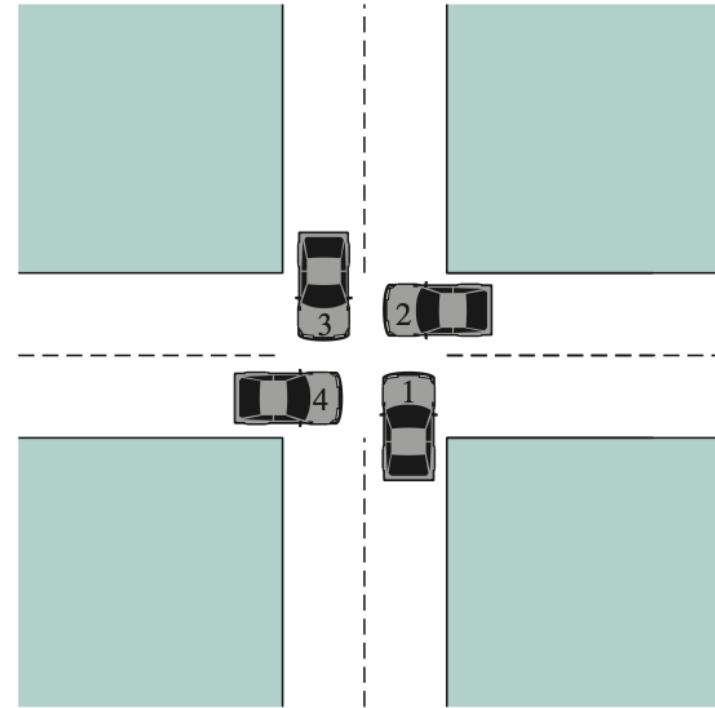
- Indica una situazione di **blocco permanente** di un gruppo di processi in competizione per le risorse di sistema
- Problema complesso e di rilievo, che può provocare gravi malfunzionamenti



# Esempio: attraversamento di un incrocio



(a) Deadlock possible



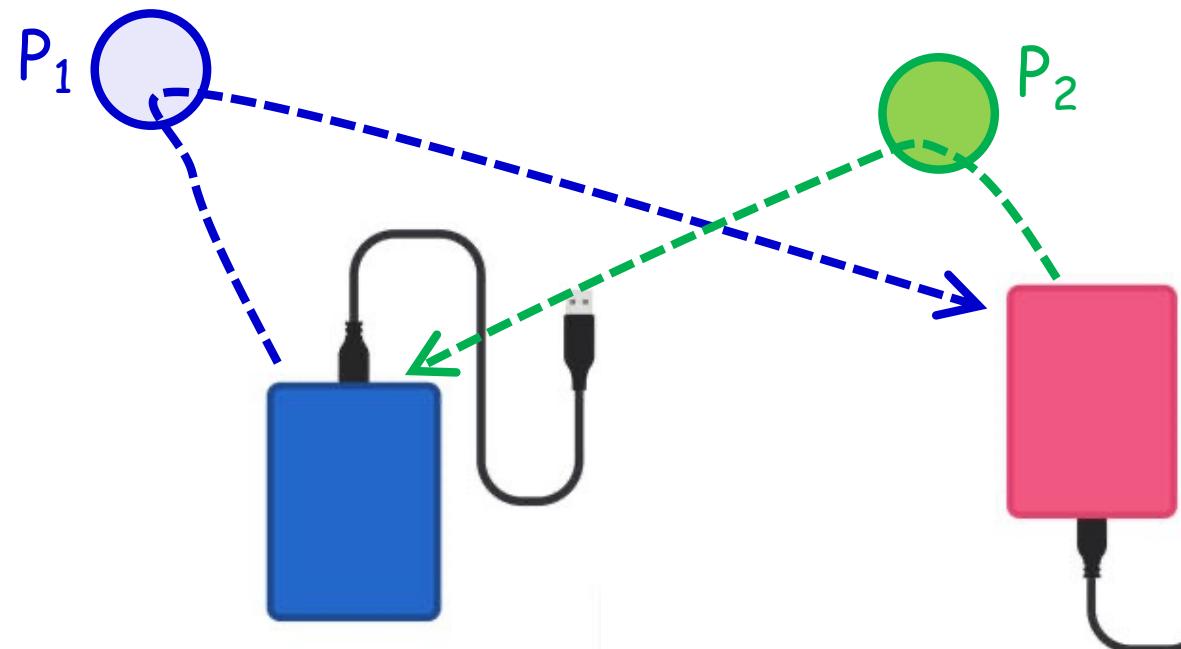
(b) Deadlock

- Ogni auto ha bisogno di attraversare due quadranti
- I quadranti dell'incrocio possono essere visti come **risorse**



# Esempio: copia di un file

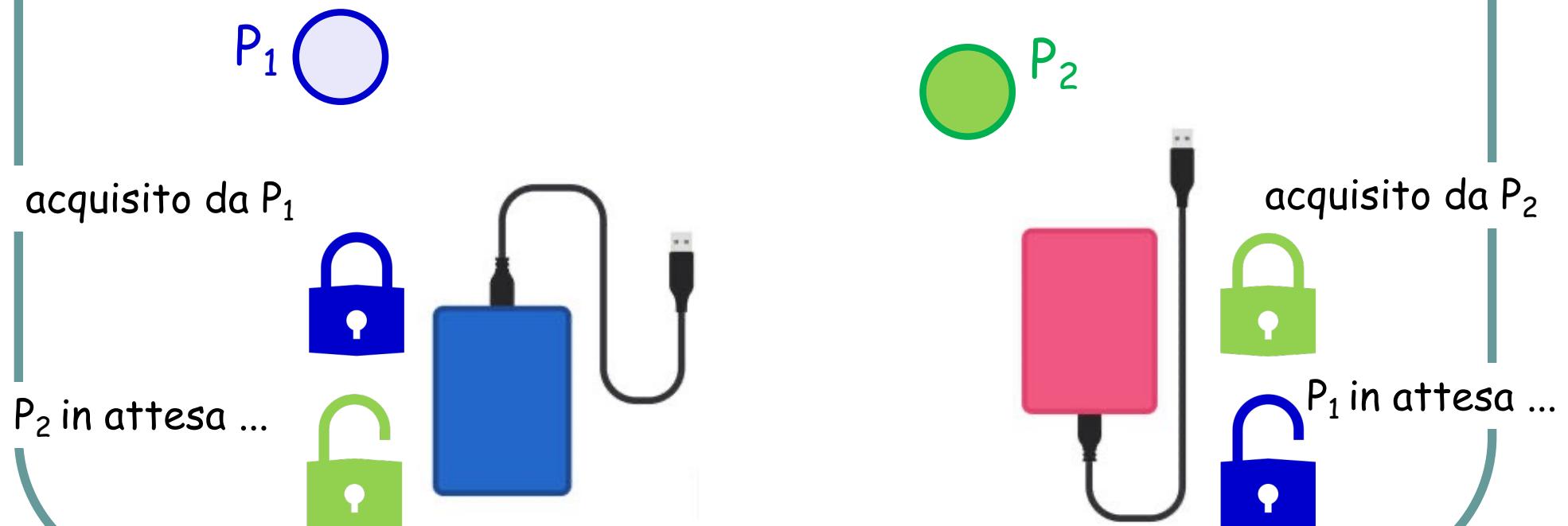
- Un sistema ha 2 dischi esterni
- $P_1$  e  $P_2$  copiano un grosso file da un disco all'altro
- Si suppone sia necessaria la **mutua esclusione**





# Esempio: copia di un file

- $P_1$  e  $P_2$  "acquisiscono" l'esclusiva ad uno dei dischi
- Ciascuno ha bisogno di **accedere anche all'altro disco** acquisito dall'altro processo





# Il problema del deadlock

- Per la sincronizzazione si usano 2 semafori *mutex1* e *mutex2*, inizializzati a 1



# Il problema del deadlock

$P_1$

```
wait (mutex1)
<inizio uso disco 1>
...
[ wait (mutex2)
  <inizio uso disco 2>
  ...
  signal (mutex2)
  <rilascio disco 2>
  ...
  signal (mutex1)
  <rilascio disco 1>
```

$P_2$

```
wait (mutex2)
<inizio uso disco 2>
...
[ wait (mutex1)
  <inizio uso disco 1>
  ...
  signal (mutex1)
  <rilascio disco 1>
  ...
  signal (mutex2)
  <rilascio disco 2>
```

I due processi potrebbero sospendersi entrambi su queste wait()



# Il problema del deadlock

- Se si verifica la sequenza di azioni:
  - $P_1$  esegue la `wait (mutex1)` e acquisisce il disco 1
  - $P_2$  esegue la `wait (mutex2)` e acquisisce il disco 2
  - $P_1$  esegue la `wait (mutex2)` e si blocca
  - $P_2$  esegue la `wait (mutex1)` e si blocca
- I due processi rimangono bloccati

Il deadlock può verificarsi saltuariamente,  
in base alla **velocità relativa di esecuzione**  
**dei processi**

# Deadlock vs starvation



**Deadlock  $\neq$  Starvation**

attesa **infinita**

attesa **indefinita**



# Grafo di assegnazione delle risorse

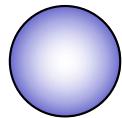
Un grafo è un insieme di vertici (o nodi)  $V$  e un insieme di archi  $E$ .

- $V$  è partizionato in due tipi :
  - $P = \{P_1, P_2, \dots, P_n\}$ , è l'insieme costituito da tutti i processi nel sistema.
  - $R = \{R_1, R_2, \dots, R_m\}$ , è l'insieme costituito da tutti i tipi di risorse nel sistema.
- Arco di richiesta (arco orientato)  $P_i \rightarrow R_j$
- Arco di assegnazione (arco orientato)  $R_j \rightarrow P_i$

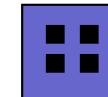


# Grafo di allocazione delle risorse

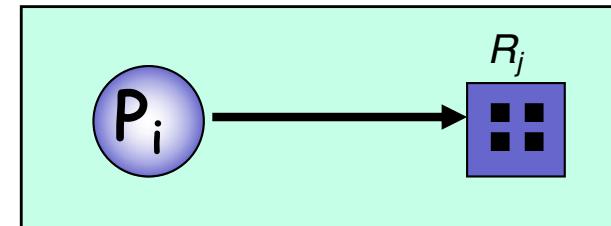
- Processo



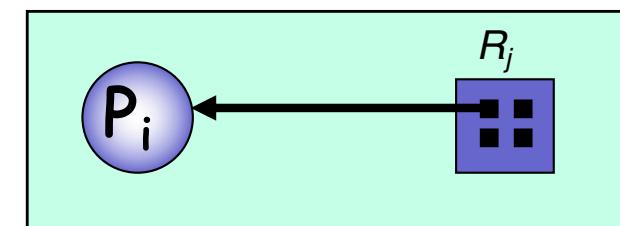
- Tipo di risorsa con 4 istanze



- $P_i$  richiede un'istanza di  $R_j$

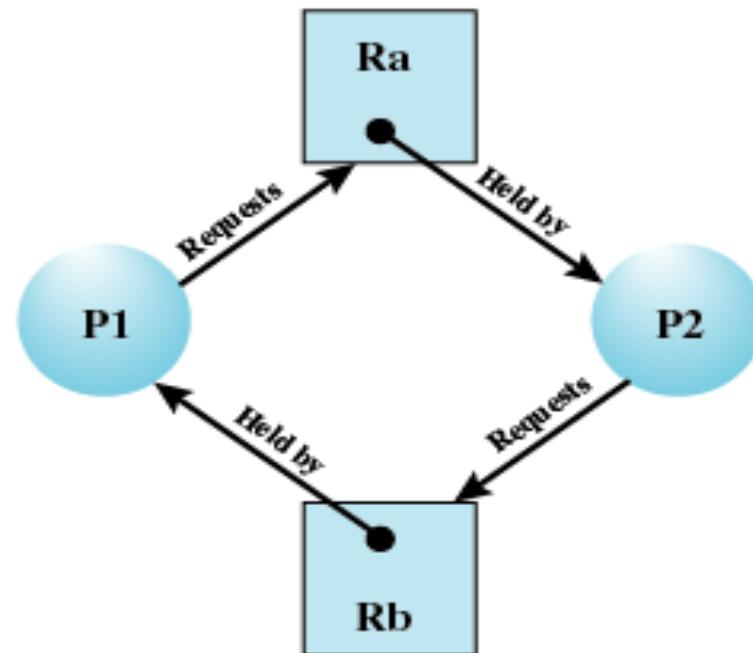


- $P_i$  possiede un'istanza di  $R_j$

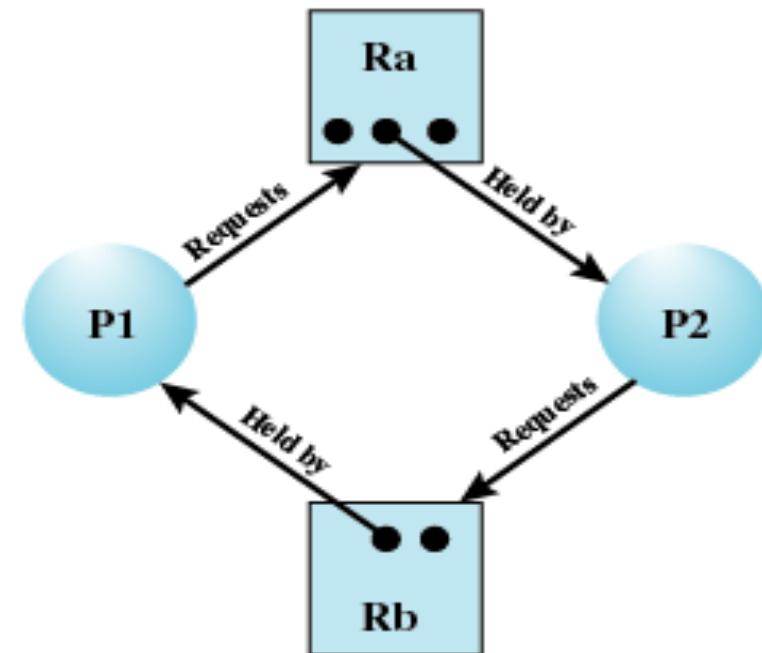




# Grafo di allocazione delle risorse



(c) Circular wait



(d) No deadlock



# Osservazioni

- Se il grafo **non contiene cicli**  $\Rightarrow$  non si verificano situazioni di stallo
- Se il grafo **contiene un ciclo**  $\Rightarrow$ 
  - se c'è solo **un'istanza** per tipo di risorsa, allora **si verifica** una situazione di stallo
  - se vi sono **più istanze** per tipo di risorsa, allora **c'è la possibilità** che si verifichi una situazione di stallo



# Metodi per la gestione dei deadlock

1. **Prevenzione dei deadlock (prevention):**  
rendere **impossibile** il verificarsi delle **condizioni per un deadlock**, ma al costo di un basso utilizzo risorse
2. **Evitare i deadlock (avoidance):**  
le condizioni per il deadlock sono consentite, ma il sistema **evita di entrare** in uno stato di deadlock
3. **Rilevazione del deadlock:**  
Si permette al sistema di entrare in uno stato di deadlock, per poi risolvere il problema (**ripristino il sistema**)

*La maggioranza dei sistemi operativi general-purpose, inclusi UNIX e Windows, **non dispone di una soluzione generale ed efficiente** al problema del deadlock*



# Condizioni per il deadlock

- **Possibilità** di un deadlock se sussistono le prime tre condizioni
  - Mutua esclusione
  - Impossibilità di prelazione
  - Possesso ed attesa
- **Esistenza** del deadlock se sussistono tutte e quattro le condizioni
  - Mutua esclusione
  - Impossibilità di prelazione
  - Possesso ed attesa
  - Attesa Circolare



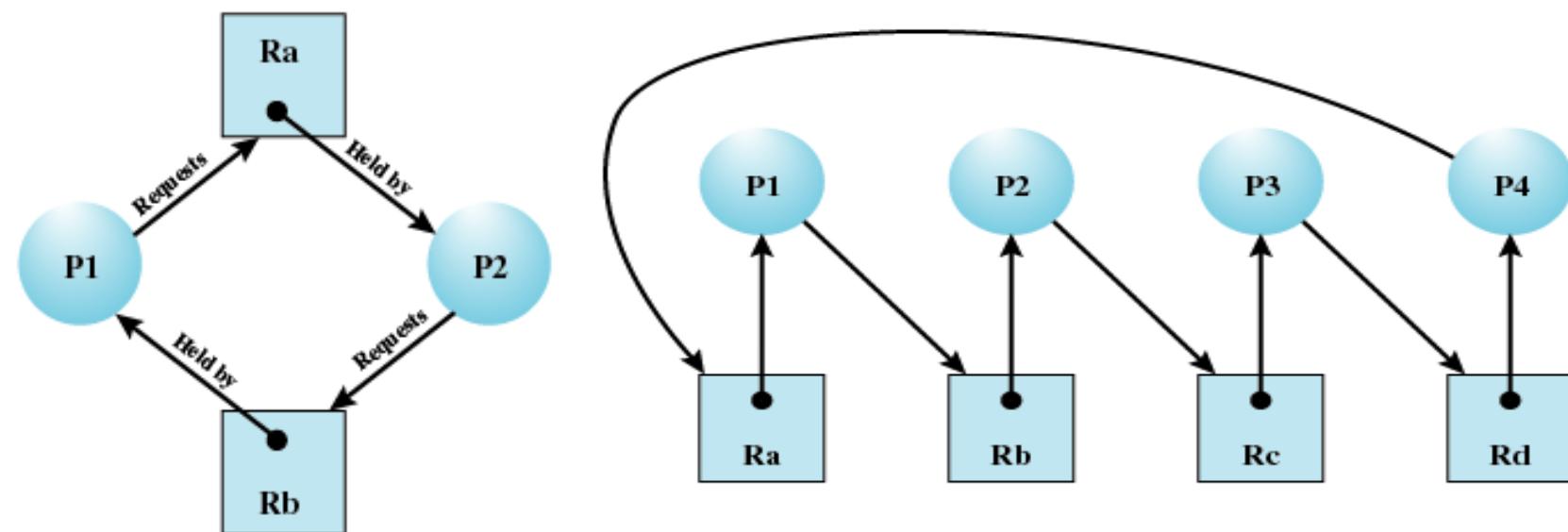
# Condizioni per il deadlock

- **Mutua esclusione:** un solo processo alla volta può usare una risorsa
- **Possesso ed attesa:** un processo che possiede almeno una risorsa, attende di acquisire **ulteriori risorse** già possedute da altri processi
- **Impossibilità di prelazione:** una risorsa può essere **rilasciata solo volontariamente** dal processo che la possiede, al termine del suo compito



# Condizioni per il deadlock

- **Attesa circolare:** esiste un insieme  $\{P_0, P_1, \dots, P_n\}$  di processi in attesa, tali che
  - $P_0$  è in attesa per una risorsa che è posseduta da  $P_1$
  - $P_1$  è in attesa per una risorsa che è posseduta da  $P_2$
  - ...
  - $P_{n-1}$  è in attesa per una risorsa che è posseduta da  $P_n$
  - $P_n$  è in attesa per una risorsa che è posseduta da  $P_0$





# Deadlock Prevention

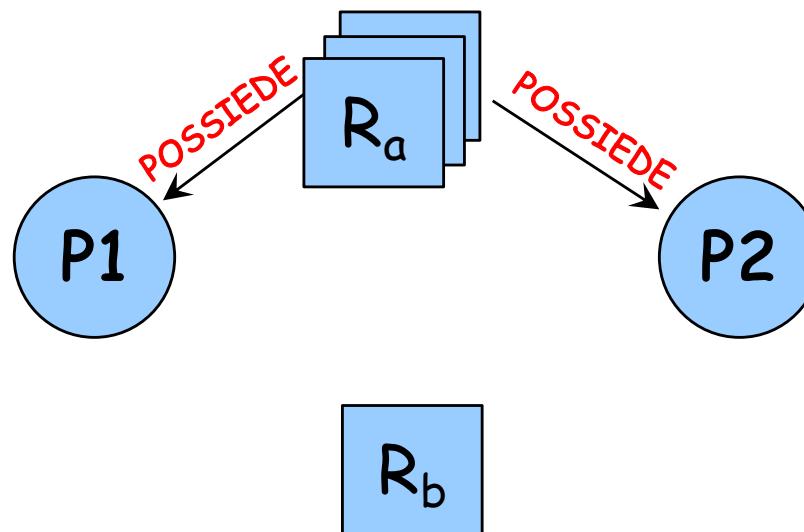
- Nella deadlock prevention, si evita il deadlock **invalidando** una delle quattro condizioni
- Causa **inefficienza** nell'uso delle risorse
  - mancato uso di risorse che sono disponibili
  - esecuzione rallentata dei processi



# Prevenzione dei deadlock

**Mutua Esclusione** - è imposta dalle caratteristiche della risorsa, e spesso non è una condizione evitabile

- Può essere rilassata in alcuni casi di risorse condivisibili
- Maggiori costi, minore efficienza



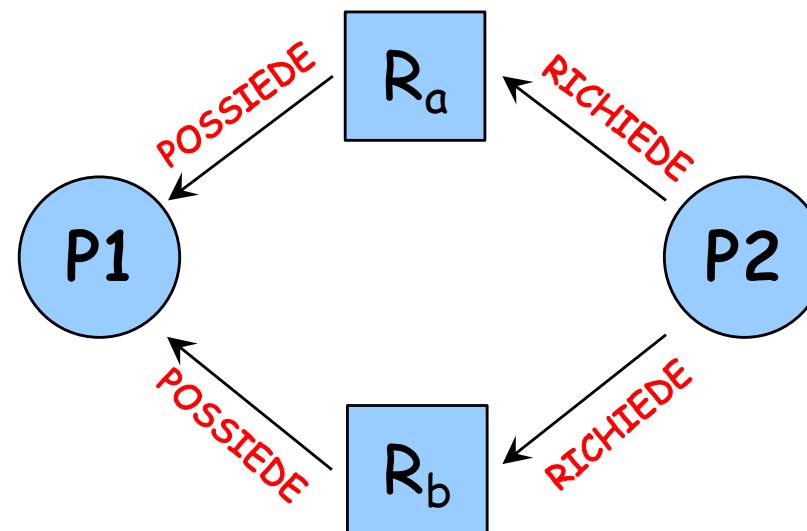


# Prevenzione dei deadlock

**Possesso e attesa** - si forza un processo a richiedere una risorsa solo quando non ne possiede altre (es. all'avvio)

- Bassa efficienza nell'uso delle risorse
- Approccio soggetto a starvation

*P1 acquisisce insieme sia Ra sia Rb  
(anche se non usa subito Rb)*



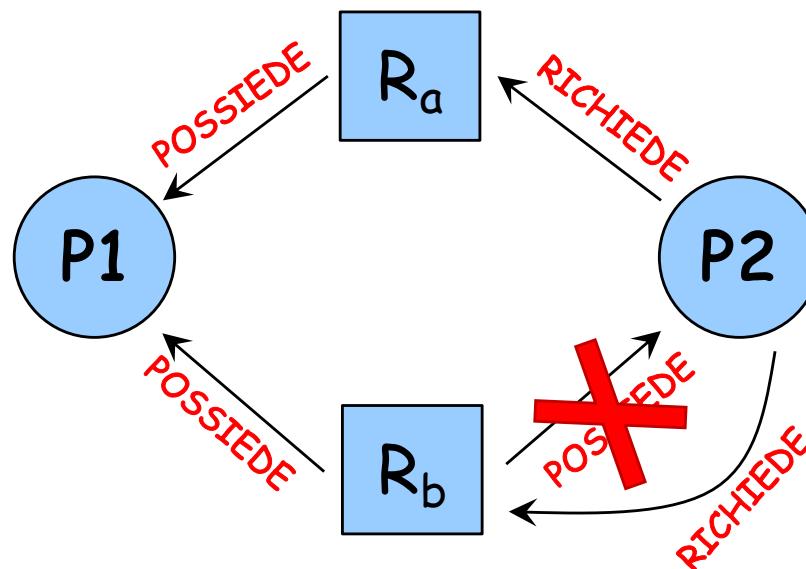
*P2 rimane sospeso  
(nonostante possa usare Rb nel frattempo)*



# Prevenzione dei deadlock

- **Impossibilità di prelazione**

- Se un processo già possiede alcune risorse, e ne richiede un'altra che non gli può essere allocata immediatamente, allora **rilascia tutte le risorse possedute**
- Il processo viene eseguito nuovamente solo quando può riottenere **sia le vecchie sia le nuove risorse**

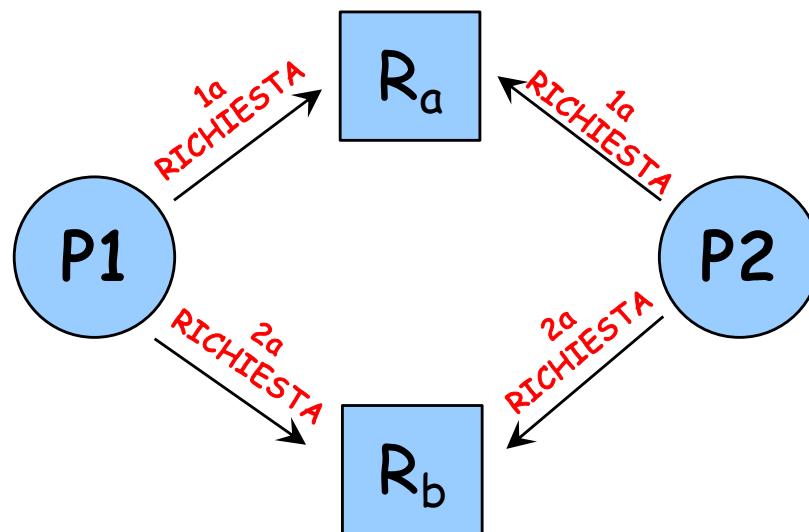




# Prevenzione dei deadlock

- **Attesa circolare**

- si stabilisce a priori un **ordinamento totale** tra tutte le risorse
- si impone che ciascun processo richieda le risorse **seguendo l'ordine prestabilito**





# Prevenzione dei deadlock

Ordine imposto: disco 1, disco 2

```
P1
...
wait (mutex1)
<inizio uso disco 1>
...
wait (mutex2)
<inizio uso disco 2>
...
```

```
P2
...
wait (mutex1)
wait (mutex2)
<inizio uso disco 2>
...
<inizio uso disco 1>
...
```

P1 è impossibilitato  
a usare "disco 1"  
anche se P2 sta  
usando "disco 2"

Si impone qui un determinato  
ordine di acquisizione delle risorse  
(a scapito della efficienza)

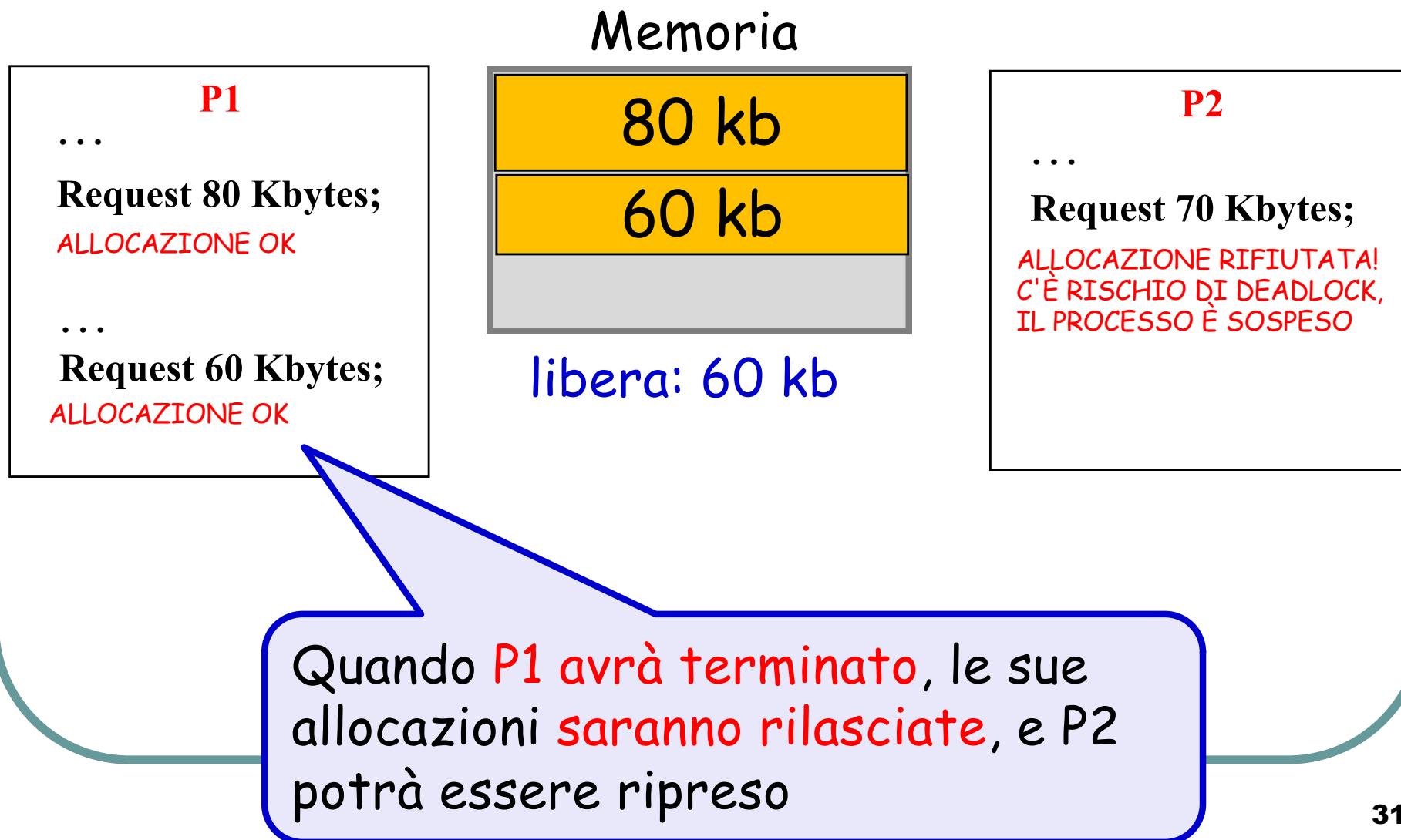


# Deadlock Avoidance

- Il sistema decide **a tempo di esecuzione** se una richiesta di una risorsa può portare ad un deadlock (*prevenzione dinamica*)
  - nessun vincolo a-priori alle richieste
  - se lo stato attuale delle risorse è "rischioso", un algoritmo **rifiuta la richiesta** di allocazione



# Deadlock Avoidance





# Deadlock Avoidance

**Presupposto:** queste tecniche richiedono di conoscere in anticipo tutte le richieste che un processo può fare nell'arco della sua esecuzione

- Caso più semplice: ogni processo dichiara il **numero massimo di risorse** di cui può avere bisogno
  - es. consumo massimo di 150 kb di memoria
  - non necessariamente usati tutti subito



# Due approcci

- All'avvio di un nuovo processo  
*(Process Initiation Denial)*

Non si avvia un processo se le sue richieste potrebbero portare ad un deadlock

- Al momento di una richiesta di allocare una risorsa *(Resource Allocation Denial)*

Si consente l'avvio, ma le richieste di allocazione possono essere rifiutate se possono portare ad un deadlock



# Process Initiation Denial

Sia  $n$  = numero di processi, e  $m$  = numero di tipi di risorse.

Resource =  $R = (R_1, \dots, R_m)$

Risorse totali nel sistema.  $R_i$  è il numero di istanze presenti nel sistema della risorsa  $R_i$ .

Available =  $V = (V_1, \dots, V_m)$

Numero di istanze per ogni risorsa non allocate ad alcun processo.  $V_i$  rappresenta il n.ro di istanze della risorsa  $R_i$  non ancora allocata

Claim =  $C$  = matrice  $n \times m$

$C_{ij}$  = richiesta del processo  $P_i$  per la risorsa  $R_j$

Allocation =  $A$  = matrice  $n \times m$

$A_{ij}$  = allocazione corrente al processo  $P_i$  della risorsa  $R_j$



# Process Initiation Denial

- La matrice  $C$  (di necessità) indica il **numero massimo di richieste** per ciascun processo (righe) di una certa risorsa (colonne)
- Deve essere fornita prima dell'avvio dei processi

$$C = \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}$$

*(Claim)* process risorse



# Process Initiation Denial

- Un processo  $P_{n+1}$  viene eseguito solo se:

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^n C_{ij} \quad \forall j$$

...cioé un processo viene eseguito se il numero massimo di richieste di tutti i processi più quelle del nuovo possono essere soddisfatte



# Process Initiation Denial

- Tre tipi di risorse:
  - 100 MB di memoria
  - 1 file di log
  - 1 porta seriale
- Claims:
  - P1: 70 MB di memoria, 1 porta seriale
  - P2: 70 MB di memoria, 1 porta seriale
  - P3: 50 MB di memoria, 1 file di log

$$\begin{array}{l} \text{Risorse} \\ R = (100 \quad 1 \quad 1) \\ \text{Claim} \\ C = \begin{pmatrix} 70 & 1 & 0 \\ 70 & 1 & 0 \\ 50 & 0 & 1 \end{pmatrix} \end{array}$$

$$V = (100 \quad 1 \quad 1)$$
$$A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Stato iniziale  
(nessun processo)

$$V = (30 \quad 0 \quad 1)$$
$$A = \begin{pmatrix} 0 & 0 & 0 \\ \textcolor{blue}{70} & \textcolor{blue}{1} & \textcolor{blue}{0} \\ 0 & 0 & 0 \end{pmatrix}$$

Avvio di P2  
(OK)

$$V = (30 \quad 0 \quad 1)$$
$$A = \begin{pmatrix} \textcolor{blue}{70} & \textcolor{blue}{1} & \textcolor{blue}{0} \\ 70 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

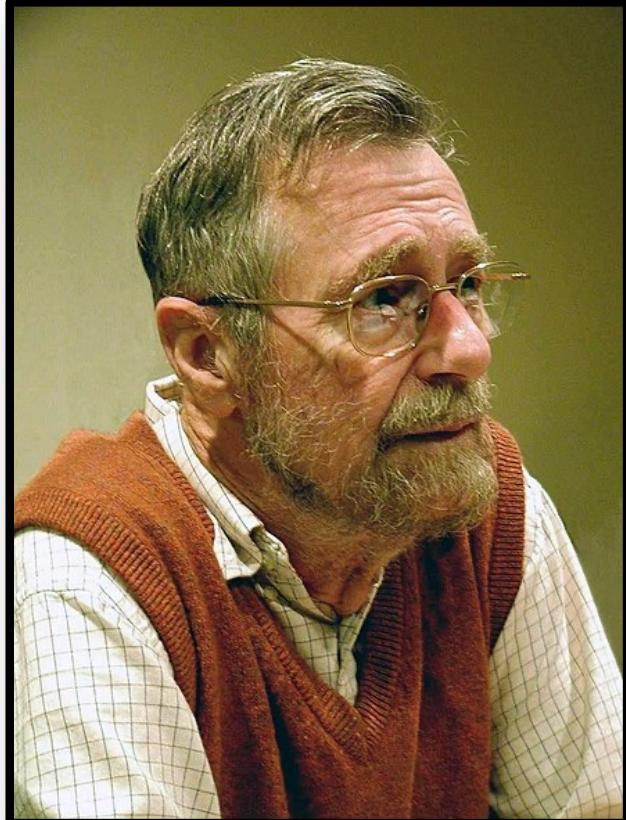
Avvio di P1  
(denial, **potenziale deadlock!**)

# Resource Allocation Denial



- Chiamato anche **algoritmo del banchiere**
- Viene eseguito ad ogni tentativo di allocazione
- Se la allocazione può portare a uno stato "non-sicuro", viene rifiutata

# Algoritmo del banchiere



E.W. Dijkstra

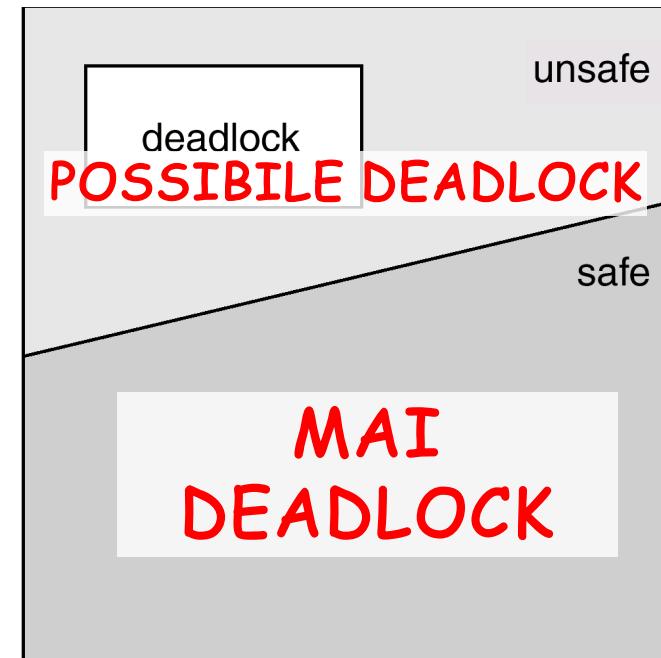
*"Il test di un  
programma può essere  
usato per mostrare la  
presenza di bug, ma  
mai per **mostrare la**  
**loro assenza**"*

-- Notes on Structured  
Programming, 1970

# Stato sicuro



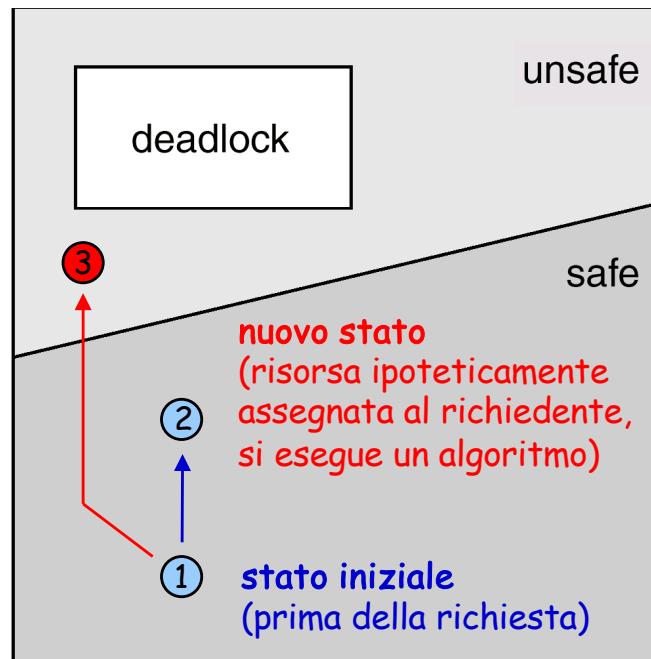
- L'algoritmo fa in modo che lo stato del sistema (processi e risorse) **non sia mai uno stato non-sicuro**





# Strategia

Lo stato 3 non è sicuro, la risorsa non è concessa  
(resource allocation denial)



Lo stato 2 è sicuro, la risorsa è concessa

La "sicurezza" di uno stato dipende dalle risorse disponibili, e dalle richieste di tutti i processi nel sistema



# Esempio di stato non-sicuro

- Processo P1:

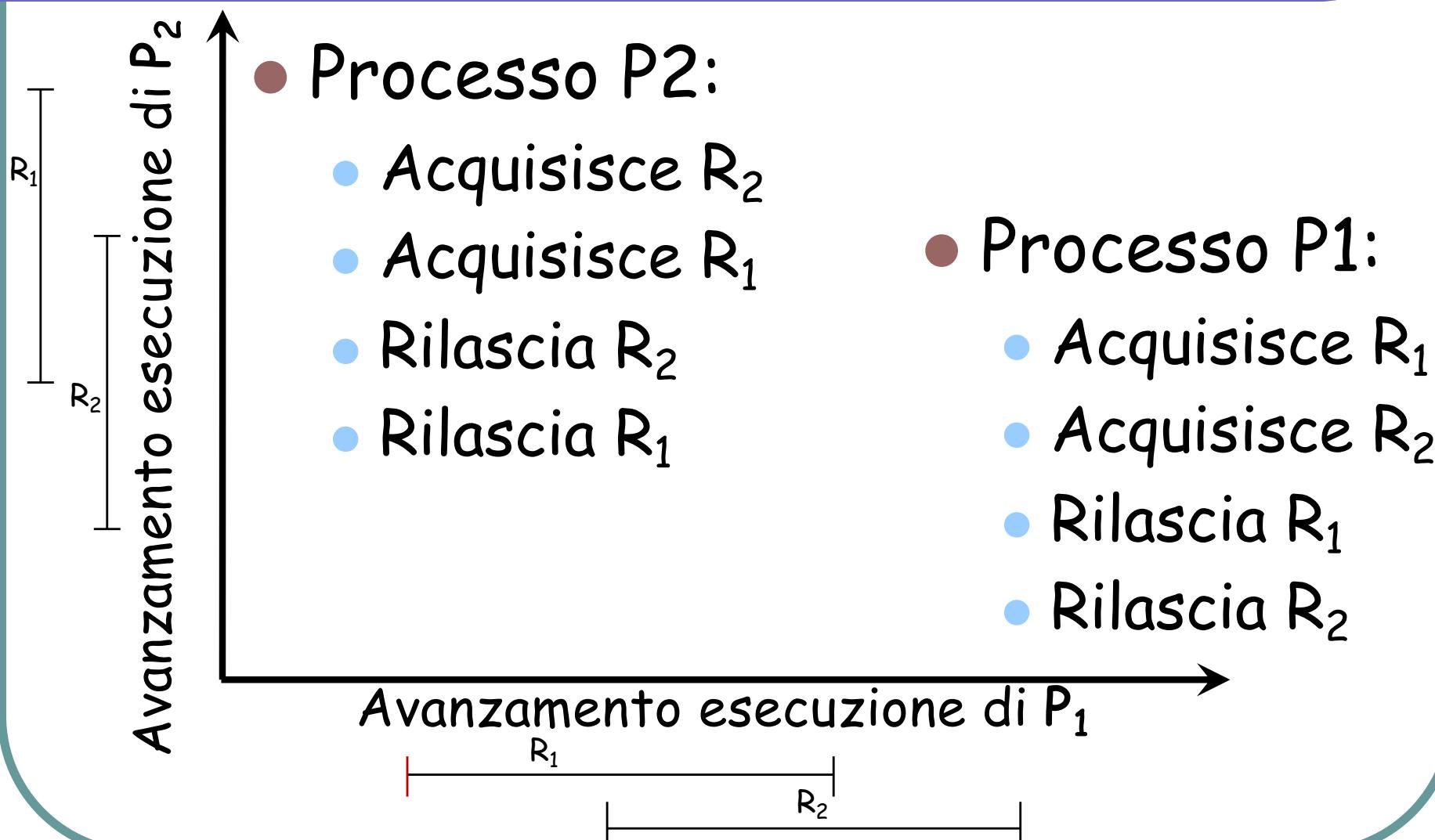
- Acquisisce  $R_1$
- Acquisisce  $R_2$
- Rilascia  $R_1$
- Rilascia  $R_2$

- Processo P2:

- Acquisisce  $R_2$
- Acquisisce  $R_1$
- Rilascia  $R_2$
- Rilascia  $R_1$

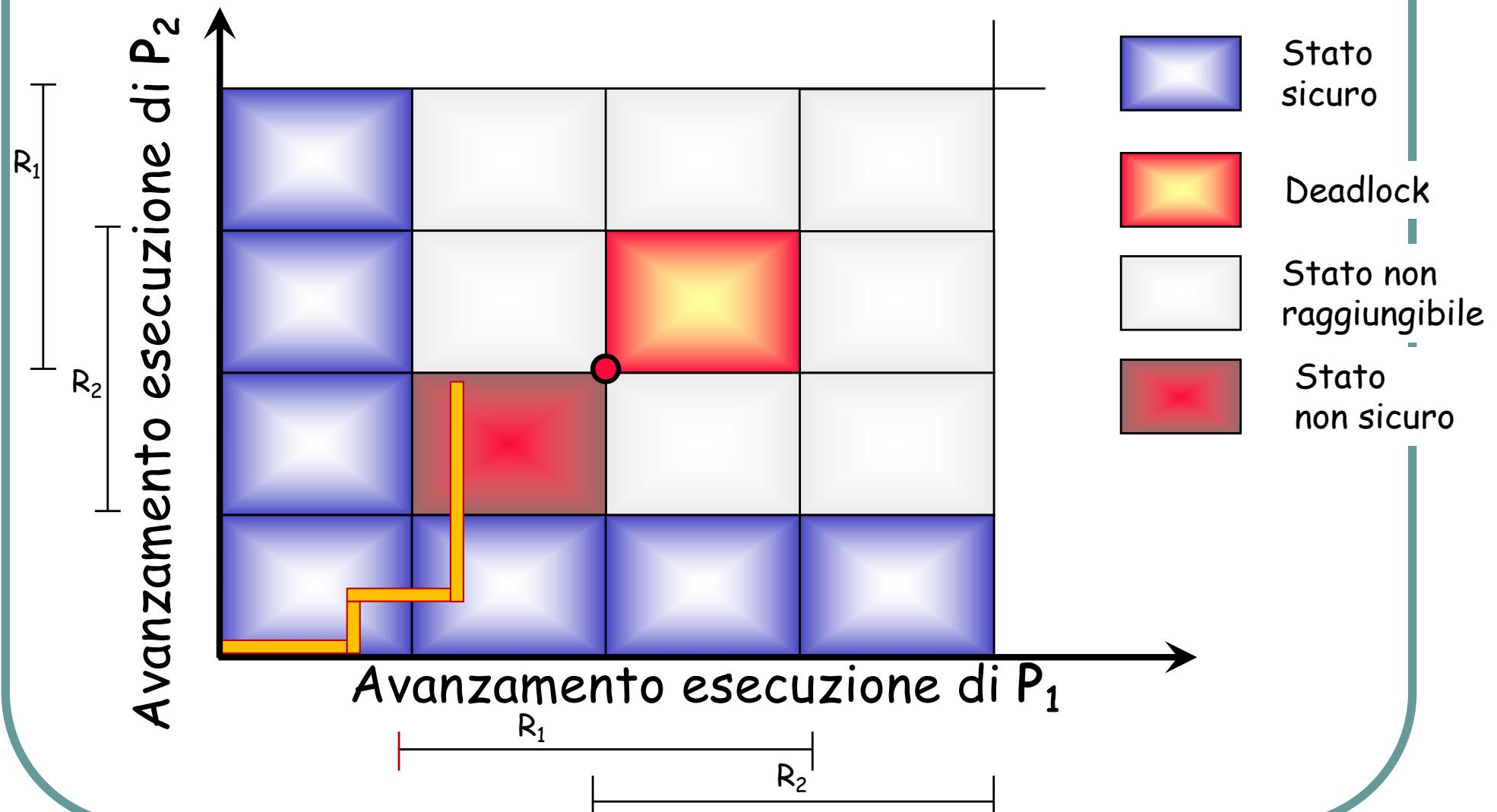


# Esempio di stato non-sicuro





# Esempio di stato non-sicuro





# Sequenza sicura

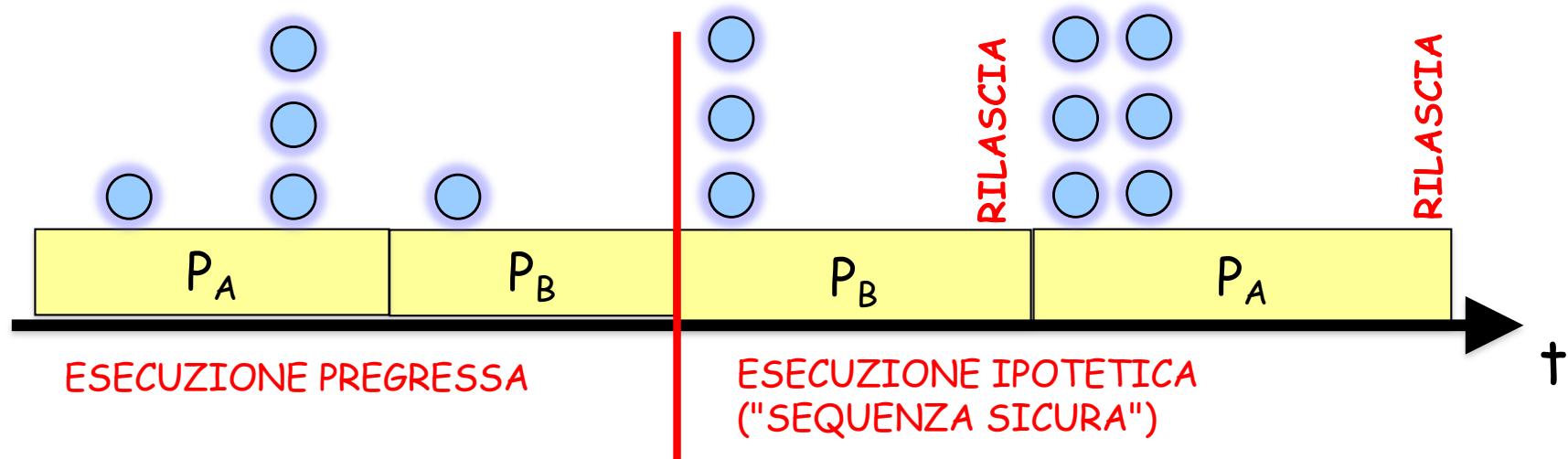
Il sistema è in uno **stato sicuro** se, partendo da questo stato, **esiste una sequenza sicura** di esecuzione di tutti i processi nel sistema

- È una sequenza di **esecuzione "ipotetica"** dei processi nel sistema ( $P_A, P_B, P_C, \dots$ )
- È sufficiente che tale sequenza "esista"

# Sequenza sicura



Unità disponibili



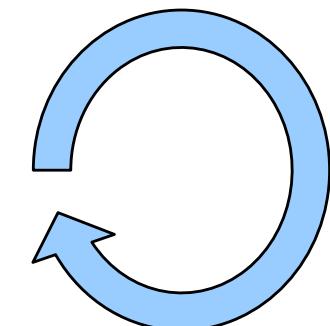
$P_B$  richiede una unità di risorsa,  
si esegue l'algoritmo

Si è trovato " $P_B, P_A$ "  
come **sequenza valida**,  
lo stato è sicuro!



# Sequenza sicura

- Si sceglie un processo (es.  $P_X$ ) da aggiungere alla sequenza
- Si assegnano (ipoteticamente) al processo  $P_X$  tutte le risorse del suo claim, se disponibili
- Per definizione,  $P_X$  ha tutte le risorse per completare la sua esecuzione
- Si ipotizza di eseguire  $P_X$  completamente
- Sono rilasciate tutte le risorse di  $P_X$



Si ripete  
per  $P_Y, P_Z \dots$



# Esempio

4 piatti per bilanciere, tutti uguali  
**(4 unità disponibili della stessa risorsa)**

**3 pesisti**, con programmi di allenamento diversi





# Esempio

OHP            2  
Bench Press 4  
Rows            2

**Claim = 4 unità**



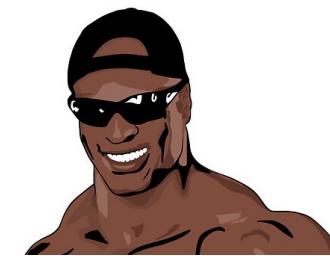
Back ext.     1  
Rows            2

**Claim = 2 unità**



Back ext.     1  
Rows            2

**Claim = 2 unità**



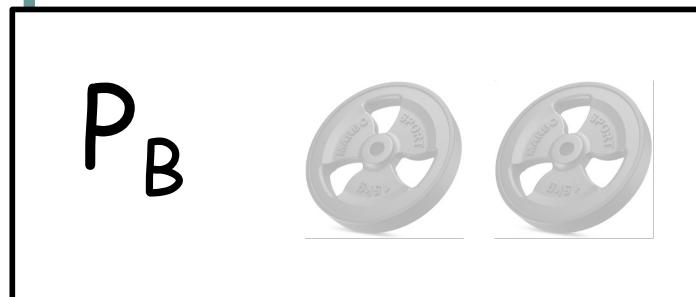


# Esempio (1/11)



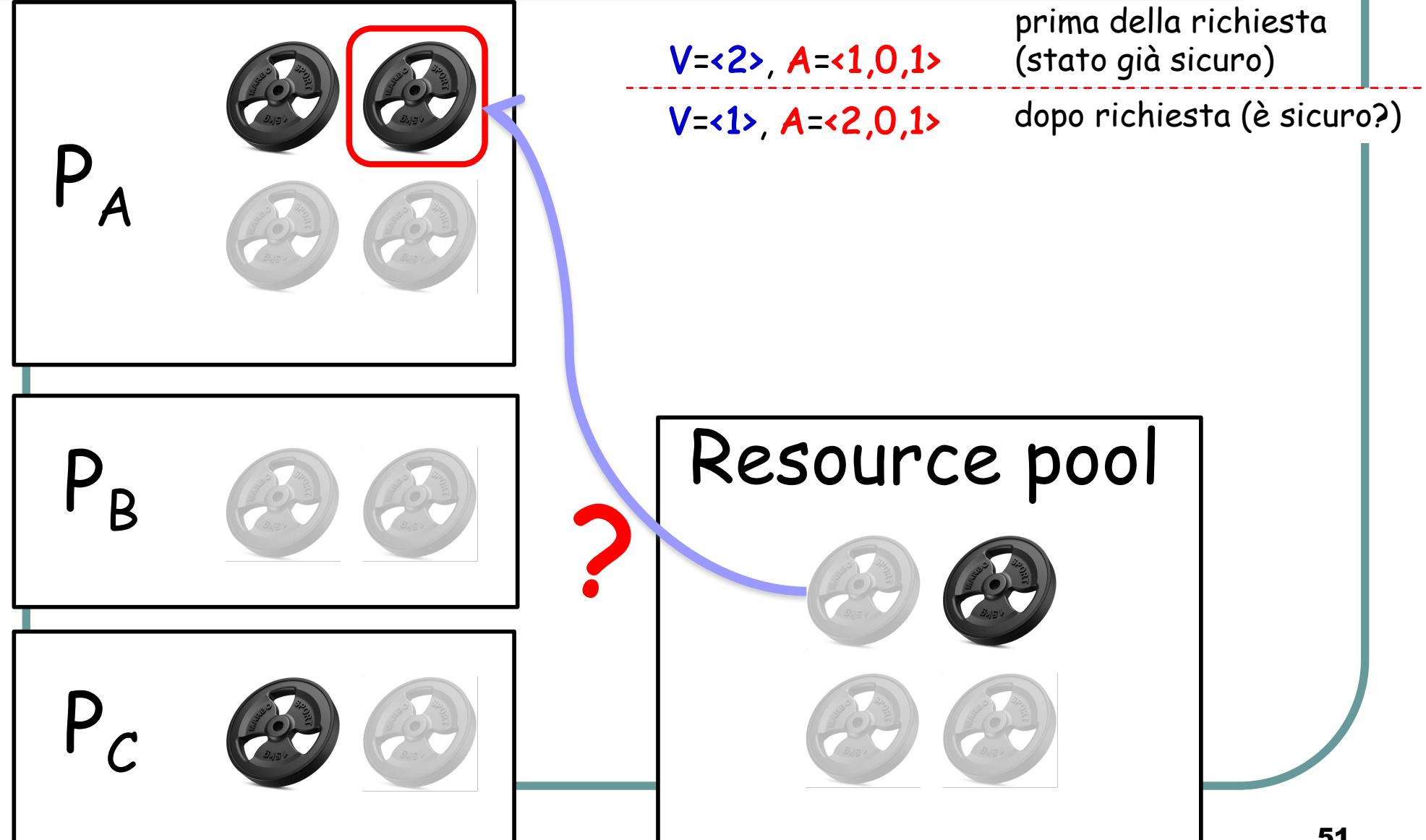
$V=<2>$ ,  $A=<1,0,1>$

prima della richiesta  
(stato già sicuro)



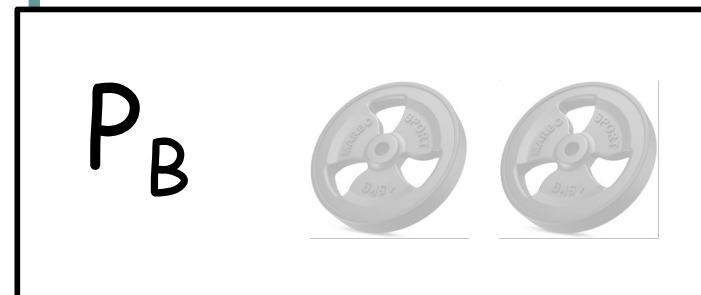
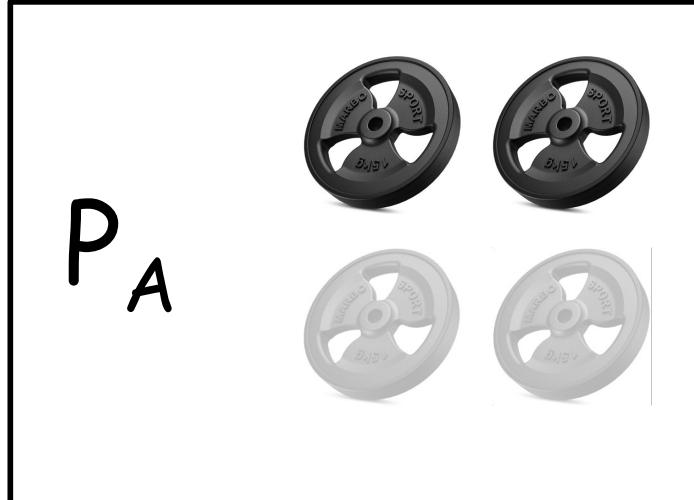


# Esempio (2/11)





# Esempio (3/11)



$V = \langle 2 \rangle, A = \langle 1, 0, 1 \rangle$

prima della richiesta  
(stato già sicuro)

$V = \langle 1 \rangle, A = \langle 2, 0, 1 \rangle$

dopo richiesta (è sicuro?)

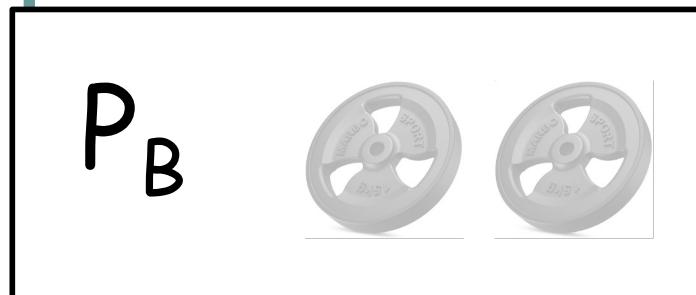
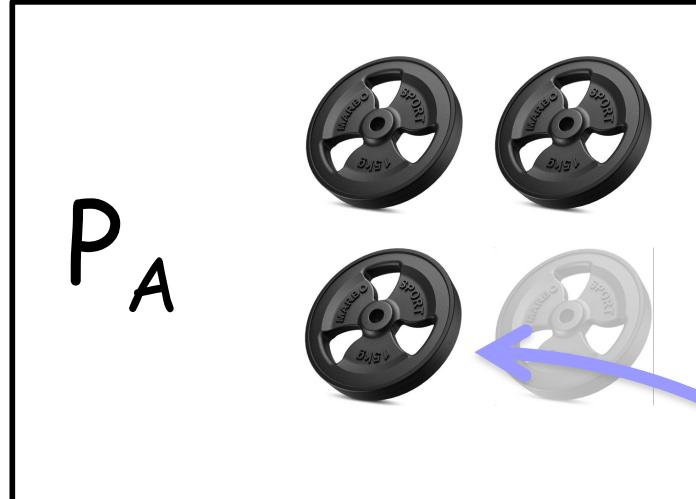
... si prova a costruire da qui  
una **sequenza sicura**  
(è una esecuzione ipotetica)

Resource pool





# Esempio (4/11)



$V=<2>, A=<1,0,1>$

prima della richiesta  
(stato già sicuro)

$V=<1>, A=<2,0,1>$

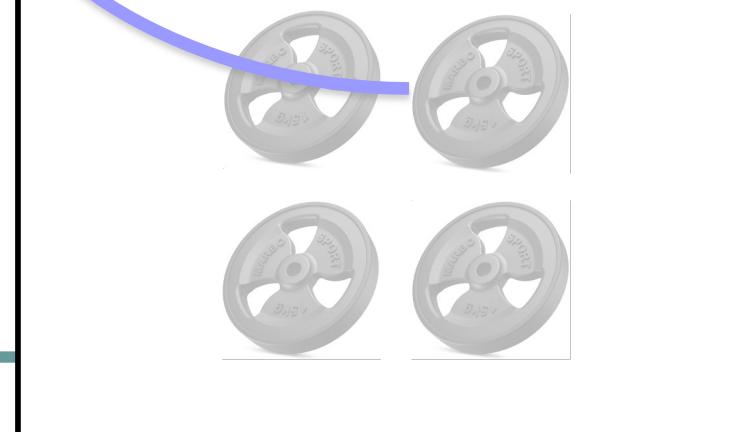
dopo richiesta (è sicuro?)

$V=<0>, A=<3,0,1>$

$\times$

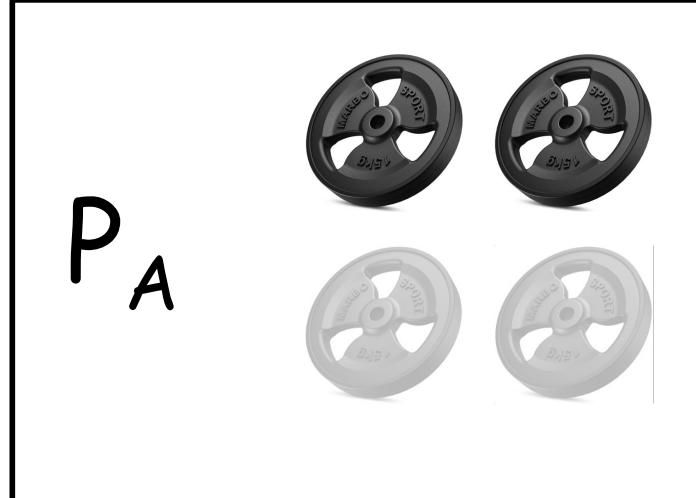
$P_A$  non riuscirebbe a terminare  
con le risorse che possiede già (2)  
+ quelle disponibili (1)

Resource pool





# Esempio (5/11)



$V = <2>, A = <1,0,1>$

prima della richiesta  
(stato già sicuro)

$V = <1>, A = <2,0,1>$

dopo richiesta (è sicuro?)

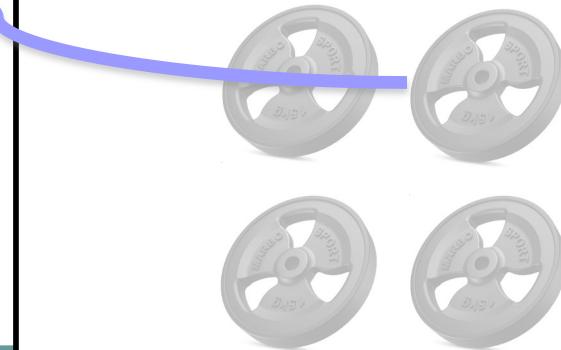
$V = <0>, A = <2,1,1>$

X

Neanche  $P_B$  potrebbe terminare  
con le risorse che possiede già (0)  
+ quelle disponibili (1)

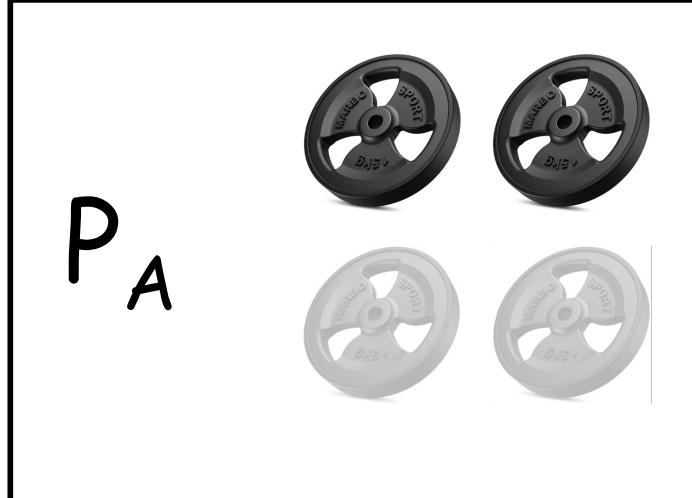


Resource pool





# Esempio (6/11)



$V = <2>, A = <1,0,1>$

prima della richiesta  
(stato già sicuro)

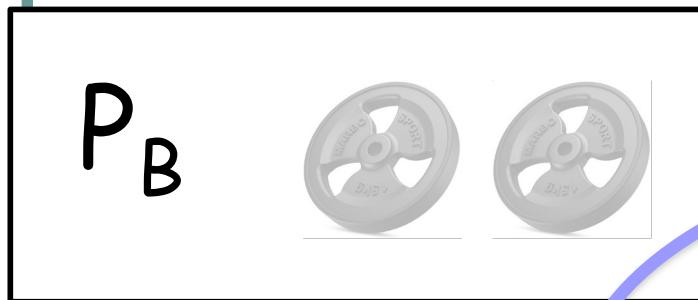
$V = <1>, A = <2,0,1>$

dopo richiesta (è sicuro?)

$V = <0>, A = <2,0,2>$



Iniziamo con  $P_C$ , dandogli la risorsa disponibile

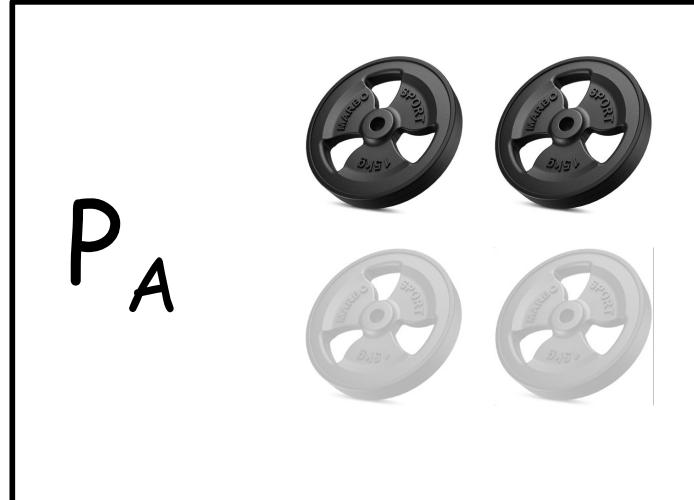


Resource pool





# Esempio (7/11)



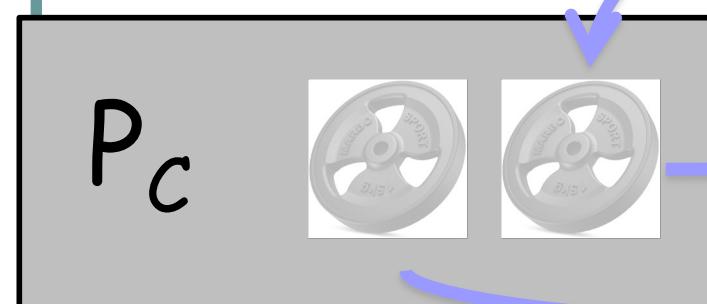
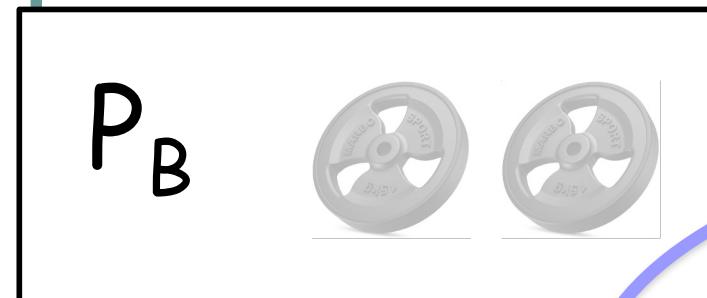
$V=<2>, A=<1,0,1>$

prima della richiesta  
(stato già sicuro)

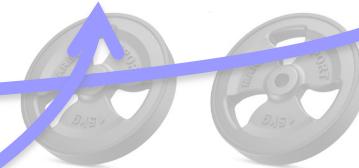
$V=<1>, A=<2,0,1>$

dopo richiesta (è sicuro?)

$V=<0>, A=<2,0,2>$



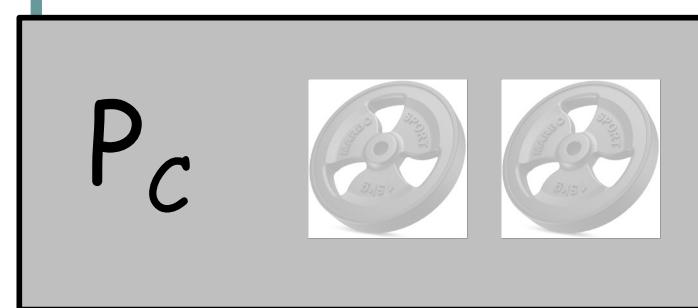
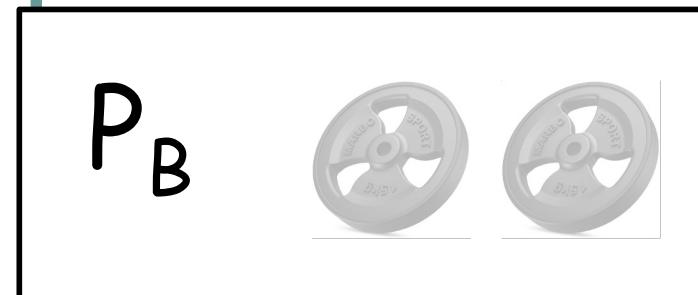
Resource pool



$P_C$  è in grado di terminare, e rilasciare le sue 2 istanze



# Esempio (8/11)



$V=<2>, A=<1,0,1>$

prima della richiesta  
(stato già sicuro)

$V=<1>, A=<2,0,1>$

dopo richiesta (è sicuro?)

$V=<0>, A=<2,0,2>$

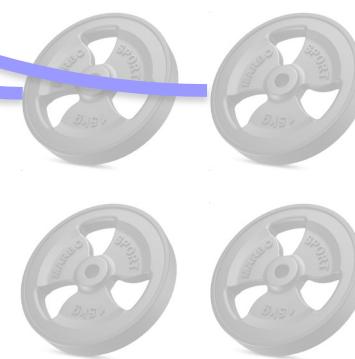


$V=<0>, A=<4,0,x>$



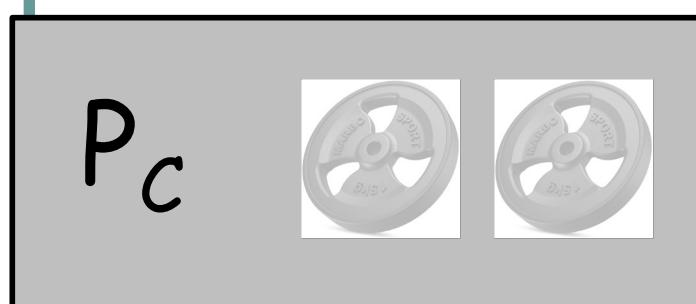
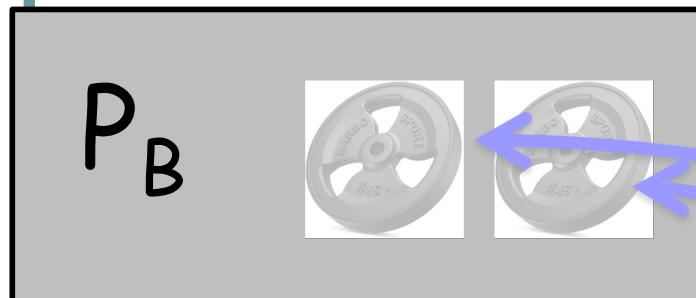
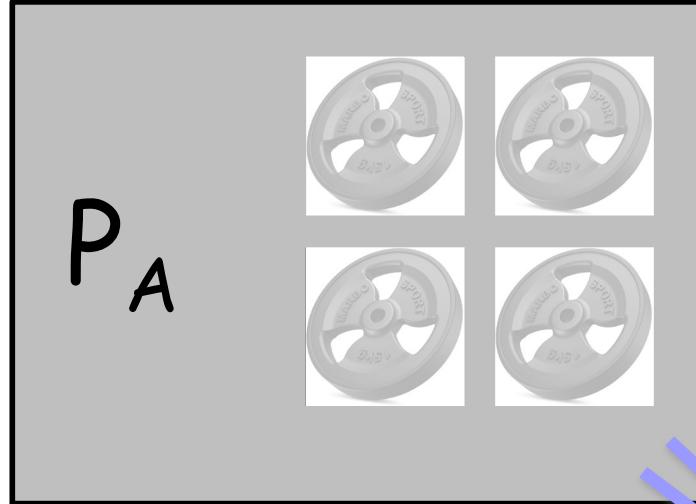
A questo punto della sequenza, anche  $P_A$  è in grado di terminare, e rilasciare le sue istanze

Resource pool





# Esempio (9/11)



$V=<2>, A=<1,0,1>$

prima della richiesta  
(stato già sicuro)

$V=<1>, A=<2,0,1>$

dopo richiesta (è sicuro?)

$V=<0>, A=<2,0,2>$



$V=<0>, A=<4,0,x>$



$V=<2>, A=<x,2,x>$



$V=<4>, A=<x,x,x>$



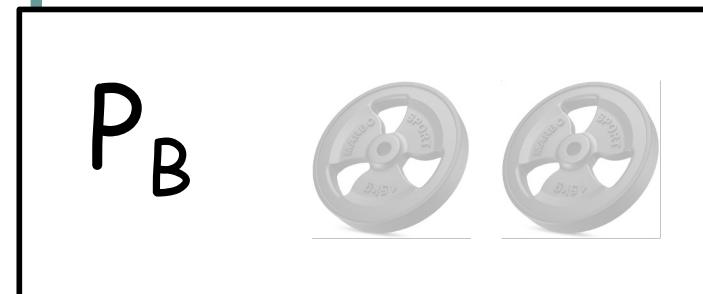
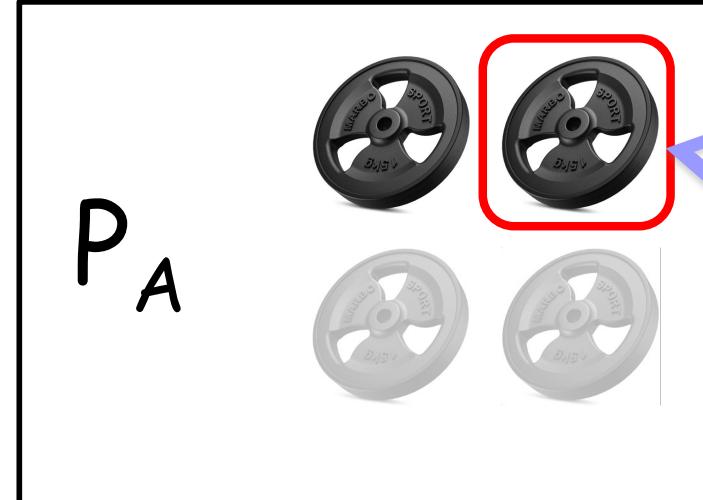
La sequenza può  
terminare con  $P_B$

Resource pool





# Esempio (10/11)



$V = \langle 2 \rangle, A = \langle 1, 0, 1 \rangle$

$V = \langle 1 \rangle, A = \langle 2, 0, 1 \rangle$

prima della richiesta  
(stato già sicuro)

dopo richiesta (assegnata)

L'algoritmo del banchiere conclude  
che questo è uno stato sicuro  
(è stata trovata una sequenza)

Resource pool





# Esempio (10/11)

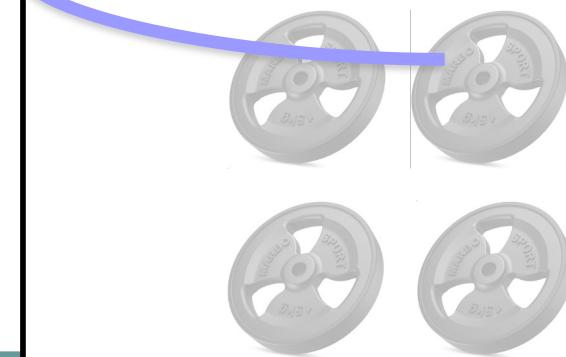


Se  $P_B$  fa richiesta dell'ultima istanza disponibile:

- 1) è possibile assegnargliela? **SI**
- 2) sarebbe uno stato sicuro? **NO**



Resource pool





# Sequenza sicura

- La sequenza sicura  $\langle P_1, P_2, \dots, P_n \rangle$  è un ordine di esecuzione dei processi, tale che:
  - Include tutti i processi attualmente attivi nel sistema
  - Ogni processo  $P_i$  ( $1 \leq i \leq n$ ) esegue nella sua interezza, dopo che tutti i processi precedenti  $P_j$  ( $j < i$ ) abbiano a loro volta eseguito per intero e nell'ordine della sequenza
  - Ogni processo  $P_i$  ottiene tutte le risorse del suo "claim", e le rilascia tutte al termine della sua esecuzione
  - Ogni processo  $P_i$  usa una quantità di risorse non superiori alla somma di:
    - risorse disponibili nello stato  $S$
    - risorse possedute e rilasciate dai processi precedenti  $P_j$  ( $j < i$ ) nella sequenza



# L'algoritmo del banchiere

```
struct state
{
    int resource[m];
    int available[m];
    int claim[n][m];
    int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
    < error >;
else if (request [*] > available [*])
    < suspend process >;
else
    /* simulate alloc */
{
    < define newstate by:
    alloc [i,*] = alloc [i,*] + request [*];
    available [*] = available [*] - request [*] >;
}
if (safe (newstate))
    < carry out allocation >;
else
{
    < restore original state >;
    < suspend process >;
}
```

(b) resource alloc algorithm



# L'algoritmo del banchiere

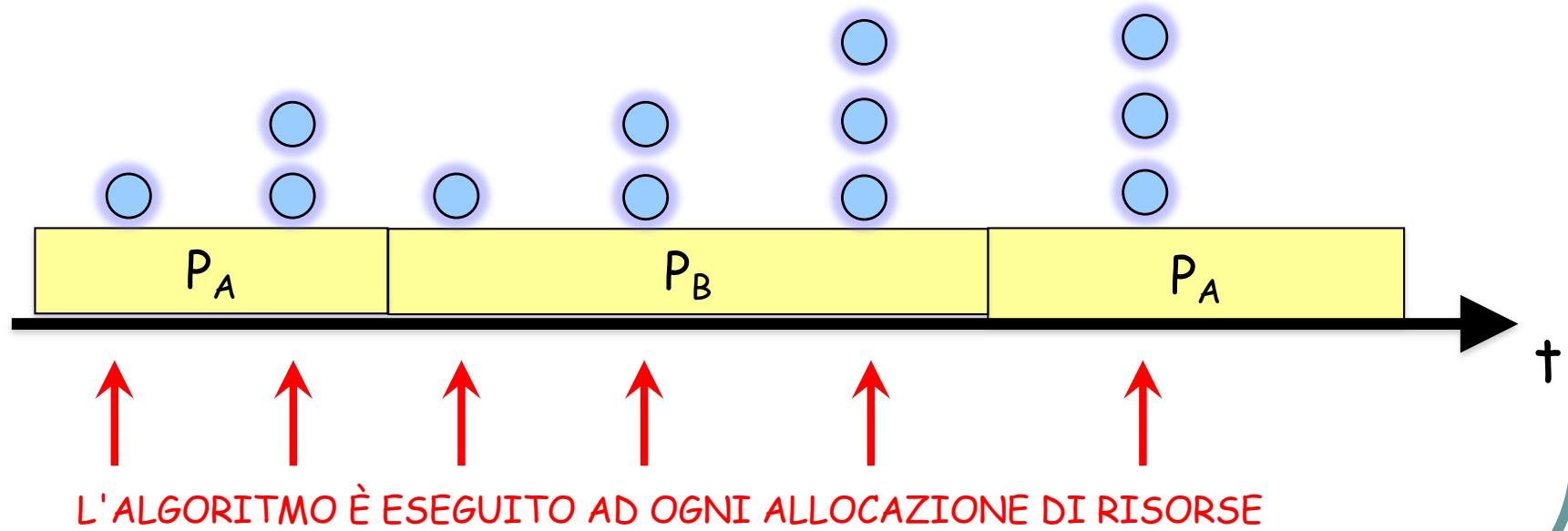
```
boolean safe (state S)
{
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible)
    {
        <find a process Pk in rest such that
        claim [k,*] - alloc [k,*] <= currentavail;>
        if (found)                                /* simulate execution of Pk */
        {
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else
            possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)



# Considerazioni

- È necessario che l'algoritmo sia **sempre eseguito ad ogni tentativo di allocazione**
- Ogni volta che lo stato cambia, si verifica che la sequenza sicura esista sempre





# Considerazioni

- Intuitivamente, l'algoritmo garantisce che **esista sempre almeno una "exit strategy"** che evita il deadlock (la sequenza sicura)
- La sequenza sicura **non è necessariamente** l'ordine con cui eseguiranno i processi!

# Problemi della deadlock avoidance



- È richiesto che sia noto preventivamente il **numero massimo** di risorse che utilizzerà
- I processi che vengono analizzati dall'algoritmo devono essere **indipendenti** (non è prevista la sincronizzazione)
- Ci deve essere un numero **predeterminato e costante di risorse** da allocare
- Nessun processo può terminare mentre è in possesso di una risorsa



# Deadlock detection

- Non vincola le richieste alle risorse, consente il verificarsi del deadlock
- Il sistema esegue un **algoritmo per il rilevamento dell'attesa circolare**
  - Periodicamente
  - oppure, ad ogni richiesta
  - oppure, quando il grado di uso della CPU è basso
- In caso affermativo, il sistema applica un **algoritmo di ripristino** (recovery)

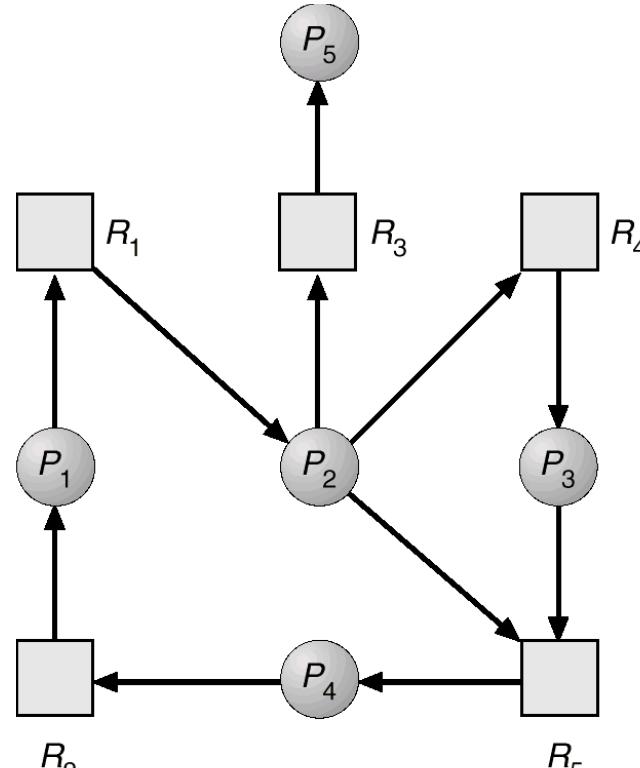


# Deadlock detection

- Una strategia di detection:
  - Impiega un **grafo di attesa**
  - I nodi sono processi.
  - $P_i \rightarrow P_j$  se  $P_i$  è in attesa che  $P_j$  rilasci una risorsa che gli occorre.
  - Periodicamente viene richiamato un algoritmo che ricerca un **ciclo nel grafo**
  - Un algoritmo per trovare un ciclo in un grafo richiede un numero di operazioni dell'ordine di  $n^2$ , dove  $n$  è il numero di vertici del grafo

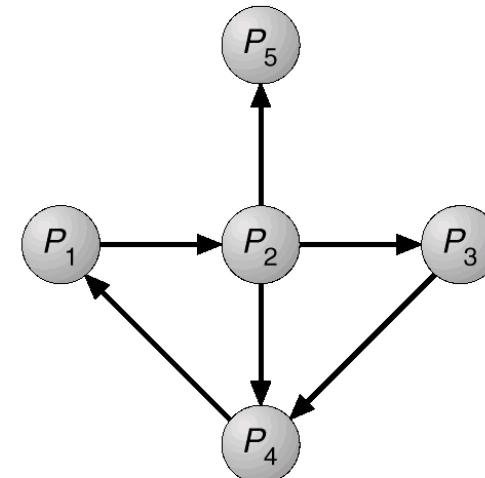


# Deadlock detection



(a)

Grafo di allocazione delle risorse



(b)

Corrispondente grafo di attesa



# Strategie di ripristino

- Si “uccidono” **tutti i processi** in uno stato di deadlock
- Si esegue un **checkpoint** di uno stato precedente al deadlock e si fanno **ripartire i processi**
- Si uccide **un processo alla volta** fino a quando il deadlock non esiste più
- Si **prelazionano le risorse** ai processi bloccati fino a quando il deadlock non esiste più

# Criterio di selezione dei processi da abortire o delle risorse da prelazionare



- Minor tempo di CPU consumato finora
- Minor numero di linee di output prodotte finora
- Maggior tempo stimato per la terminazione
- Minor numero di risorse allocate finora
- Minore priorità

# Tabella di comparazione delle strategie per la gestione del deadlock



Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	<ul style="list-style-type: none"><li>• Works well for processes that perform a single burst of activity</li><li>• No preemption necessary</li></ul>	<ul style="list-style-type: none"><li>• Inefficient</li><li>• Delays process initiation</li><li>• Future resource requirements must be known by processes</li></ul>
		Preemption	<ul style="list-style-type: none"><li>• Convenient when applied to resources whose state can be saved and restored easily</li></ul>	<ul style="list-style-type: none"><li>• Preempts more often than necessary</li></ul>
		Resource ordering	<ul style="list-style-type: none"><li>• Feasible to enforce via compile-time checks</li><li>• Needs no run-time computation since problem is solved in system design</li></ul>	<ul style="list-style-type: none"><li>• Disallows incremental resource requests</li></ul>
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	<ul style="list-style-type: none"><li>• No preemption necessary</li></ul>	<ul style="list-style-type: none"><li>• Future resource requirements must be known by OS</li><li>• Processes can be blocked for long periods</li></ul>
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	<ul style="list-style-type: none"><li>• Never delays process initiation</li><li>• Facilitates on-line handling</li></ul>	<ul style="list-style-type: none"><li>• Inherent preemption losses</li></ul>

# Quiz



1. Quali di queste strategie contro il deadlock è la peggiore, dal punto di vista del basso uso delle risorse? (selezionare una)

- Invalidare una delle condizioni per il deadlock
- Process Initiation Denial
- Algoritmo del banchiere

<https://forms.office.com/r/yQrHLdvtMZ>

