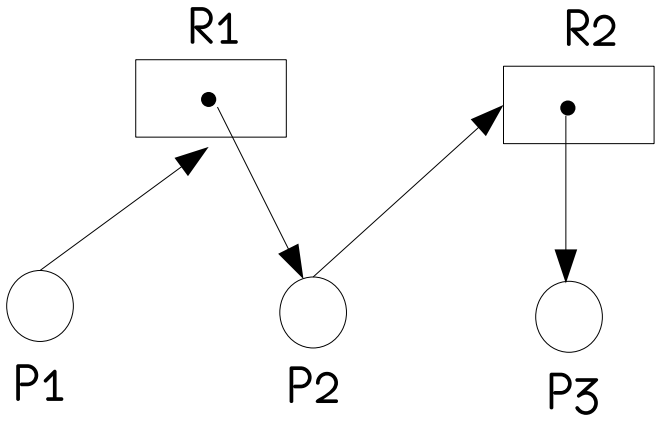
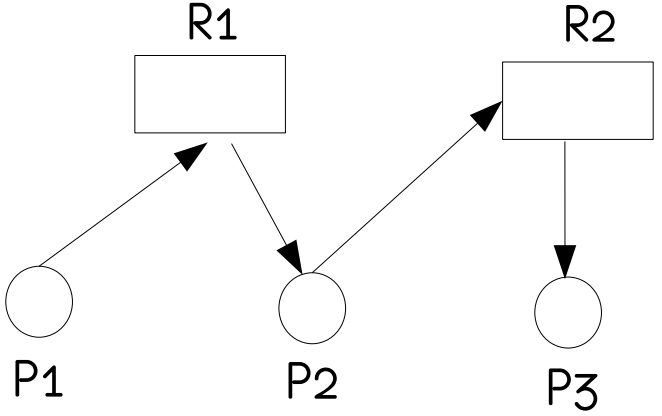
Che fare col deadlock?

- **Rilevare il deadlock**:
  - è una capacità fondamentale se non abbiamo metodi, come i precedenti, che a priori ne evitano il generarsi: rilevato un dedlock è possibile attuare una politica di ripristino dalla condizione di stallo. Due casi:
    - istanza singola per ogni classe di risorsa
    - istanze multiple
- Rompere il deadlock quando si presenta:
  - richiede la capacità di monitorare le richieste/assegnazioni di risorse
- Prevenire il deadlock:
  - occorre definire opportuni protocolli di assegnazione delle risorse
- Far finta che il deadlock sia impossibile:
  - è la tecnica più usata, poco costosa perché non richiede né risorse aggiuntive né l'attuazione di politiche particolari

Istanza singola di risorsa

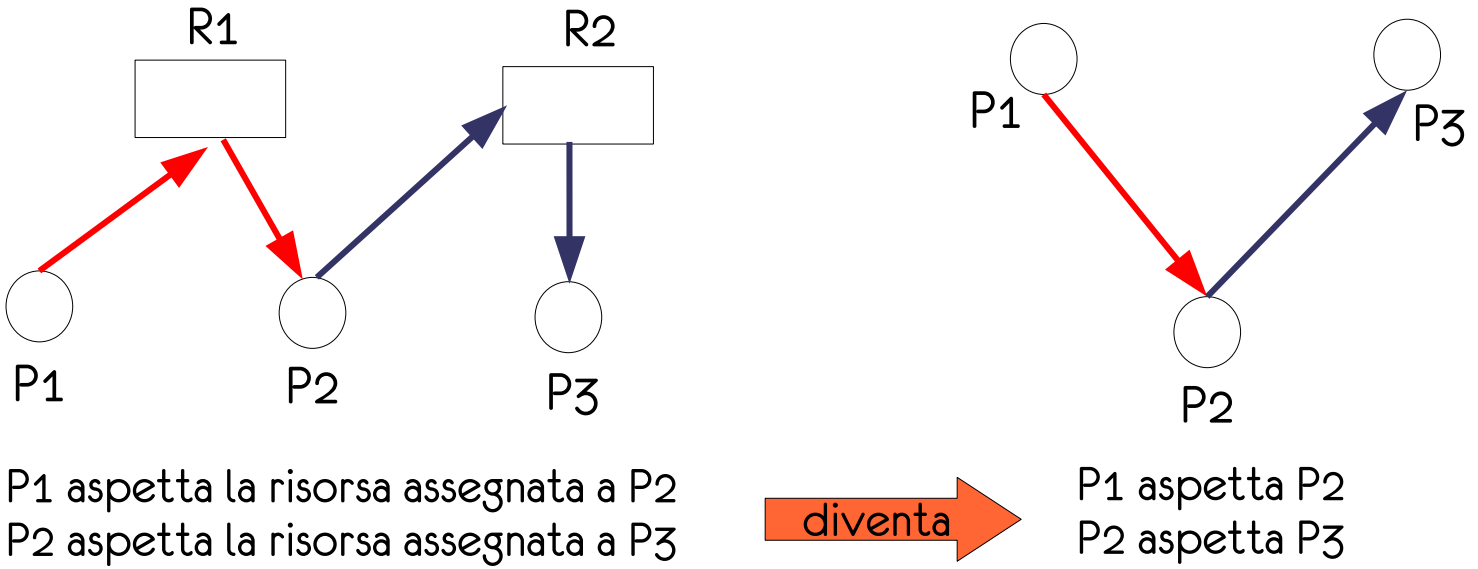
- Caso in parte già discusso
- Si basa su di una semplificazione del grafo di assegnazione delle risorse detto **Grafo d'Attesa**
- Un grafo d'attesa ha un solo tipo di vertici: i **processi**
- **Si può costruire un grafo di attesa partendo da un grafo di allocazione**
- <passo 1> Osserviamo che se l'istanza è singola, è ridondante mantenere una notazione per, la classe di risorse e una per le istanze: una classe un'istanza



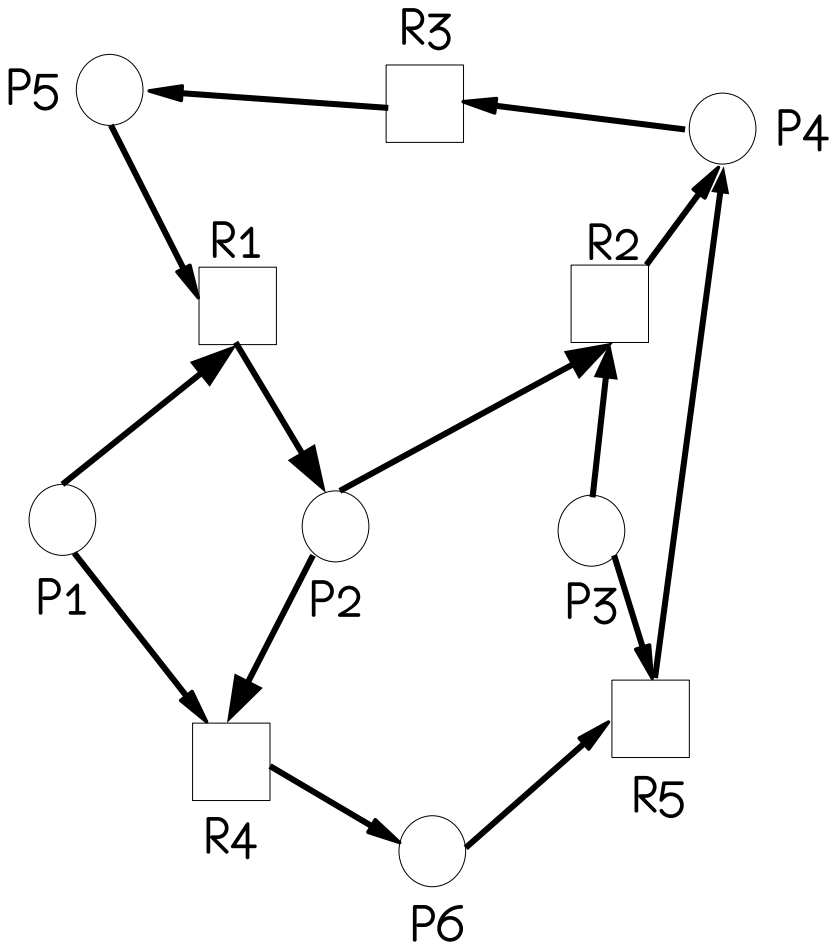


Istanza singola di risorsa

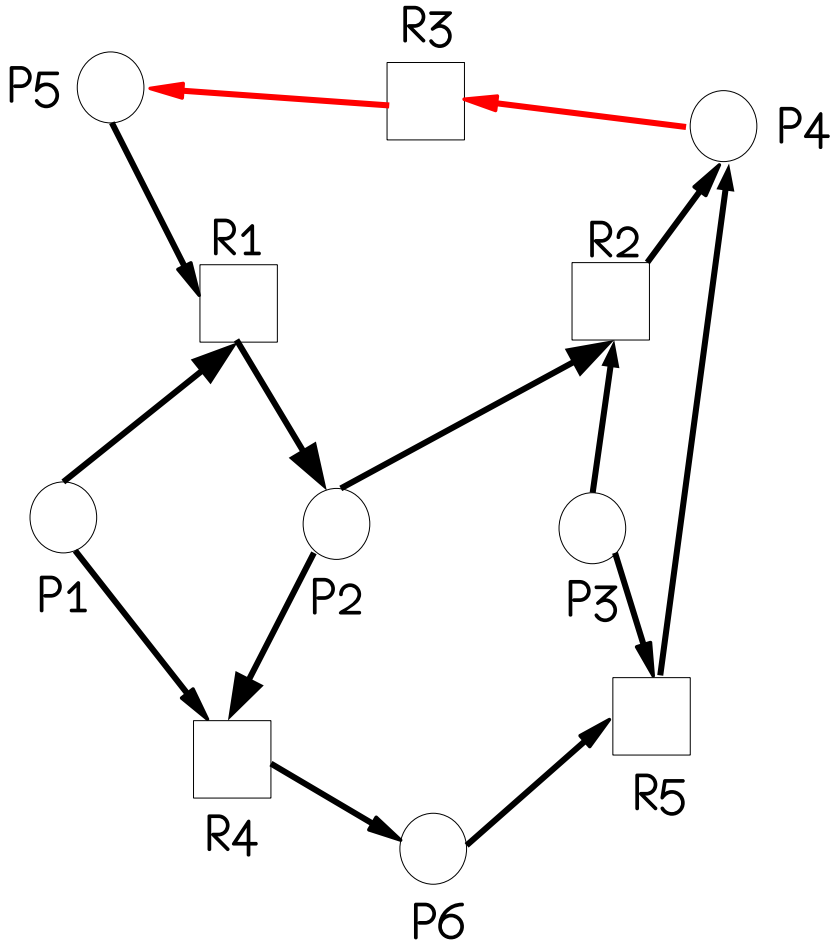
- Un grafo d'attesa è ancora più semplice infatti ha un solo tipo di vertici: i processi
- Un arco Pi → Pj indica che Pi è in attesa di una risorsa assegnata a Pj
- <passo 2> Un arco Pi → Pj corrisponde a una coppia di archi Pi → Rs e Rs → Pj nel grafo di allocazione:



esempio 1/5



esempio 2/5



P1 → R1 → P2
P1 → R4 → P6

P2 → R4 → P6
P2 → R2 → P4

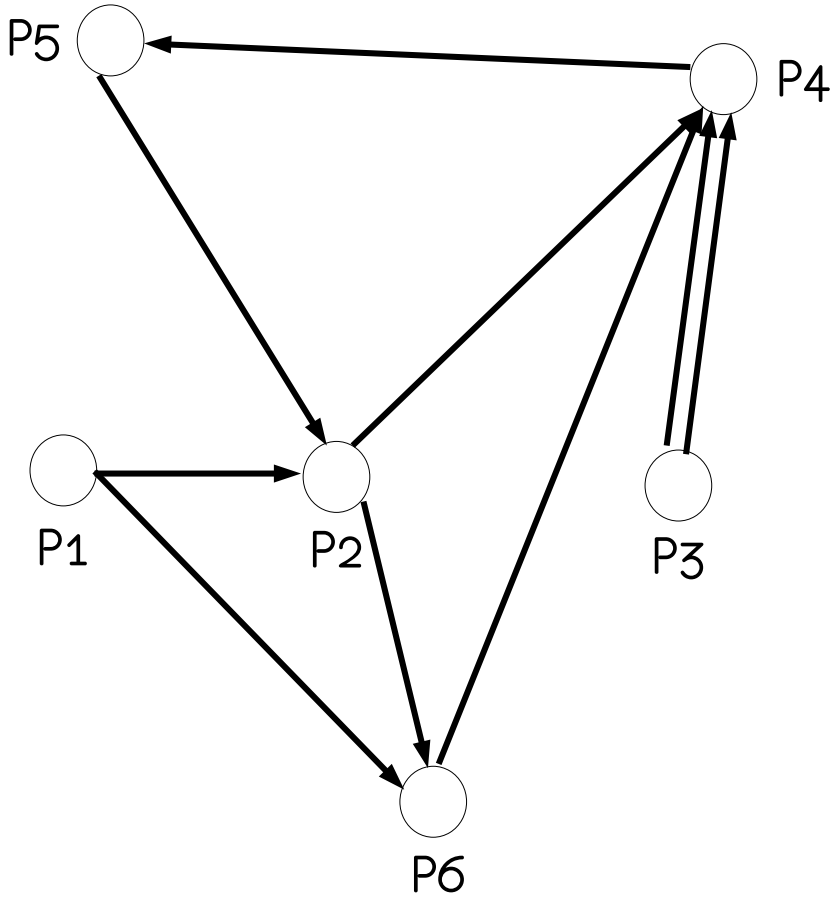
P3 → R2 → P4
P3 → R5 → P4

P4 → R3 → P5

P5 → R1 → P2

P6 → R5 → P4

esempio 3/5



P1 → R1 → P2
P1 → R4 → P6

P2 → R4 → P6
P2 → R2 → P4

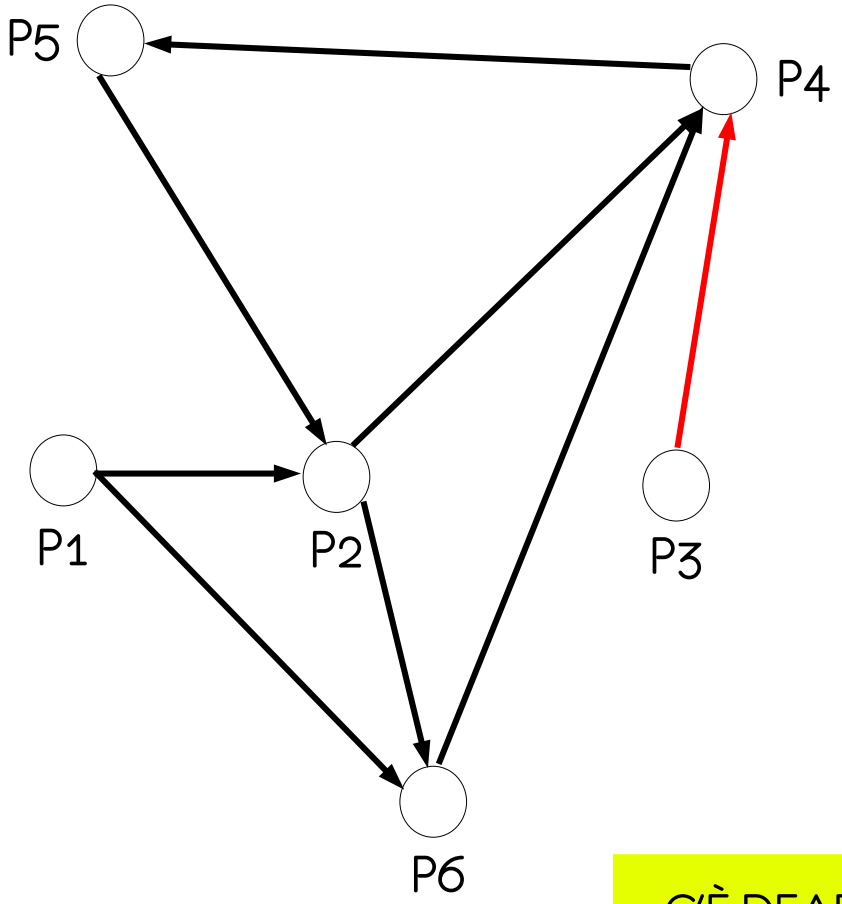
P3 → R2 → P4
P3 → R5 → P4

P4 → R3 → P5

P5 → R1 → P2

P6 → R5 → P4

esempio 4/5



P1 → R1 → P2
P1 → R4 → P6

P2 → R4 → P6
P2 → R2 → P4

P3 → R2 → P4
P3 → R5 → P4

nella visualizzazione li fondiamo in un arco solo

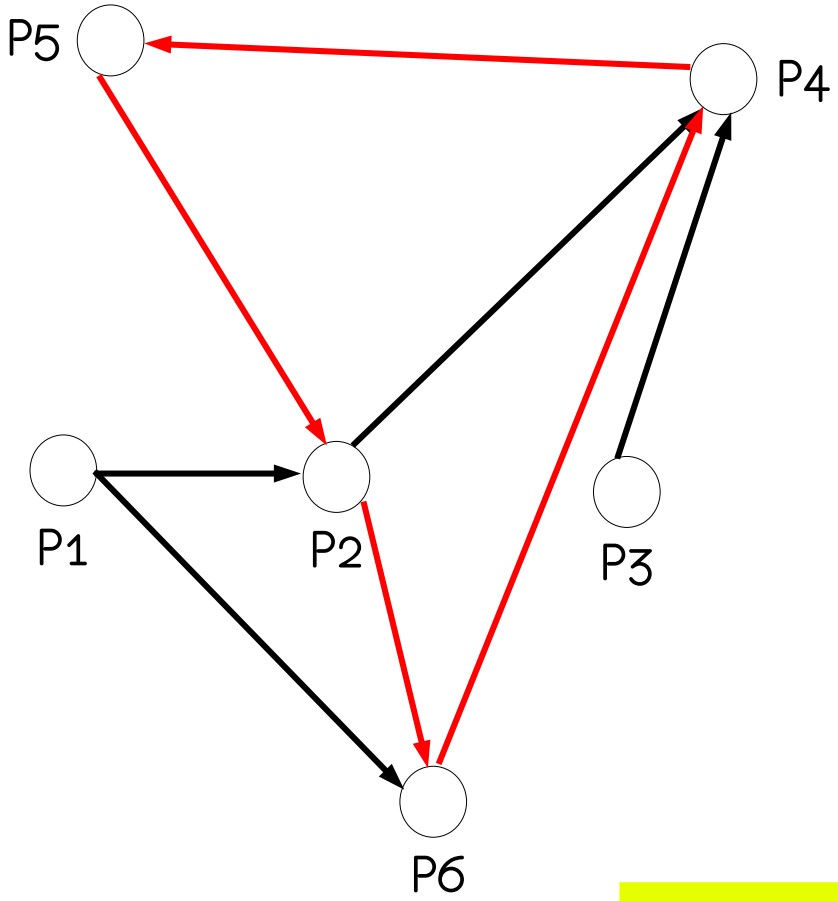
P4 → R3 → P5

P5 → R1 → P2

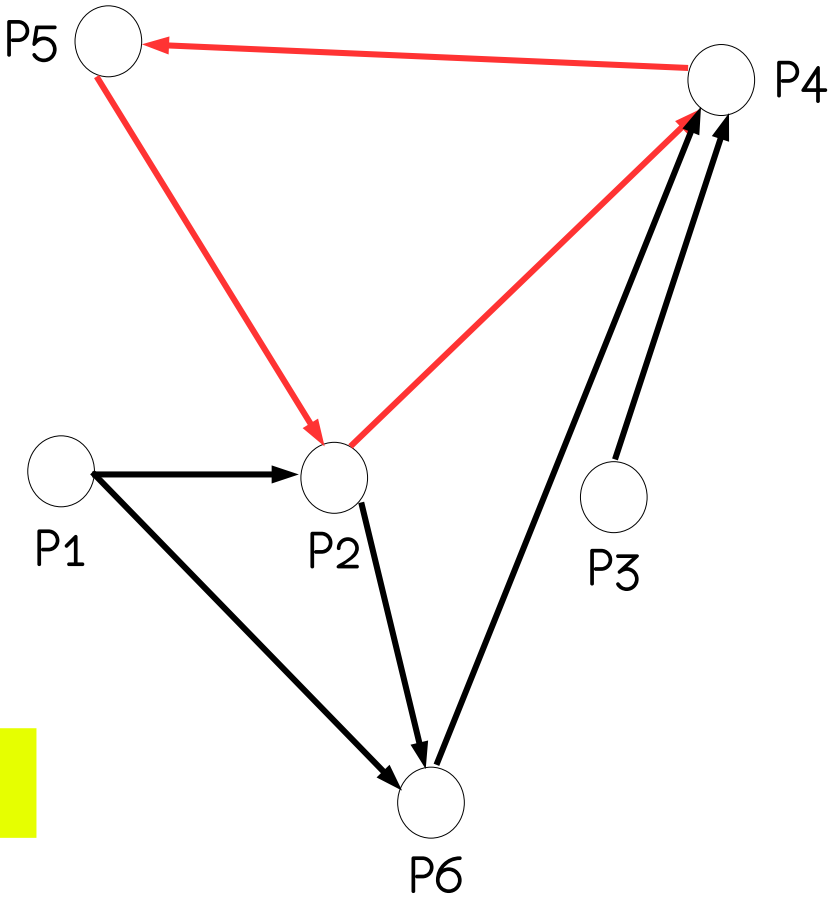
P6 → R5 → P4

C'È DEADLOCK?
BISOGNA VERIFICARE SE CI SONO CICLI

esempio 4/5



CI SONO CICLI?
PIÙ DI UNO ...



DEADLOCK

Istanze multiple di risorsa

- Il metodo precedente è applicabile se e solo se per ogni classe di risorsa esiste un'unica istanza
- In generale per ogni classe di risorsa R^i si possono avere molte istanze, in formule: $| R^i | = N^i \geq 1$

Istanze multiple di risorsa

- Occorre utilizzare *strutture dati analoghe a quelle usate nell'algoritmo del banchiere*:
- **Disponibili[M] = {n1, ..., nk}** mantiene il numero di istanze disponibili di ogni risorsa
- **Assegnate[N][M]**: ogni riga della matrice indica quante istanze di ciascun tipo di risorsa sono state assegnate a un certo processo; Assegnate[i] indica l'attuale assegnazione per processo Pi
- **Richieste[N][M]**: ogni riga della matrice indica la **richiesta attuale** di ogni processo, tale richiesta può comprendere istanze di risorse differenti (non si tratta di claim, cioè di richieste future)
- NB: Assegnate e Richieste catturano le situazioni di possesso e attesa correnti

NB: lo scopo è rilevare il deadlock, accorgersi se c'è e non prevenirlo

Esempio

Disponibili == {3, 1, 0, 2}

Abbiamo 4 classi di risorse di cui sono attualmente disponibili rispettivamente 3, 1, 0 e 2 istanze

Assegnate[3][4] ==

0	0	0	0
0	1	1	0
2	0	1	3

Ci sono 3 processi due dei quali hanno assegnate istanze di diversi tipi di risorse (P2 ha complessivamente 2 istanze di due risorse diverse, P3 ha 6 istanze di 3 risorse diverse)

Richieste[3][4] ==

1	2	0	0
0	0	0	0
0	1	0	0

Dei tre processi due hanno una richiesta in corso (P1 richiede 3 istanze di due classi diverse e P3 richiede una sola istanza di una risorsa). Secondo la disponibilità attuale la richiesta di P3 è soddisfacibile quella di P1 no

L'algoritmo che vedremo **individua cicli di processi**, per essere in un ciclo un processo deve avere assegnate alcune risorse ed essere in attesa di altre (avere richieste in corso)

Algoritmo

```
1.  int Lavoro[M];
2.  boolean Fine[N];
3.
4.  /* inizializzazione */
5.  Lavoro = Disponibili;
6.  for (i in [1,N])
7.      if (Assegnate[1] == {0, 0, ..., 0}) Fine[i] = true;
8.      else Fine[i] = false;
9.
10.  /* calcolo */
11.  while ∃ un indice i | Fine[i] == false ∧ Richieste[i] ≤ Lavoro
       1. Lavoro = Lavoro + Assegnate[i]
       2. Fine[i] = true
12.
13.  /* test: c'è deadlock? */
14.  for (i in [1,N])
       1.if (Fine[i] == false) << c'è deadlock >>
```

F sta per False

NB: tutti i processi per cui Fine[i] = false sono in deadlock

esempio

Lavoro == Disponibili == {3, 1, 0, 2}

Fine[N] = {true, false, false}

Assegnate[3][4] ==

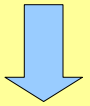
```
0 0 0 0
0 1 1 0
2 0 1 3
```

P1 non può essere parte di un ciclo non ha risorse assegnate

Richieste[3][4] ==

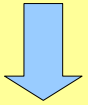
```
1 2 0 0
0 0 0 0
0 1 0 0
```

```
while (∃i | Fine[i]==false ∧ Richieste[i] ≤ Lavoro)
        1. …
```

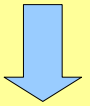


```
condizione vera per i == 2, infatti:
Fine[2] == false
Richieste[2] == {0, 0, 0, 0} < {3, 1, 0, 2}

 Lavoro = Lavoro + Assegnate[2] = {3, 2, 1, 2}
 Fine[2] = true
```

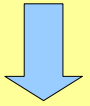


```
while (∃i | Fine[i]==false ∧ Richieste[i] ≤ Lavoro)
        1. …
```



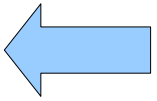
```
condizione vera per i == 3, infatti:
Fine[3] == false
Richieste[3] == {0, 1, 0, 0} < {3, 2, 1, 2}

 Lavoro = Lavoro + Assegnate[3] = {5, 2, 2, 5}
 Fine[3] = true
```



```
while (∃i | Fine[i]==false ∧ Richieste[i] ≤ Lavoro)
LA CONDIZIONE È FALSA
```

NON C'È DEADLOCK
FINE = {TRUE, TRUE, TRUE}



esempio con deadlock

Lavoro == Disponibili == {3, 1, 0, 2}

Fine[N] = {true, false, false}

Assegnate[3][4] ==

```
0 0 0 0
0 1 1 2
2 0 1 3
```

P1 non può essere parte di un ciclo non ha risorse assegnate

Richieste[3][4] ==

```
1 0 0 0
2 0 1 0
0 1 0 3
```

Se volete provare un caso più generale simulate sulla matrice Assegnate:

```
1 0 0 0
0 1 1 2
2 0 1 3
```

```
while (∃i | Fine[i]==false ∧ Richieste[i] ≤ Lavoro)
        1. …
```

condizione falsa, infatti:

Richieste[2] == {0, 1, 1, 2} non è < {3, 1, 0, 2}
infatti una componente di Richieste è > della corrispondente componente di Lavoro

Richieste[3] == {2, 0, 1, 0} non è < {3, 1, 0, 2}

ESCO DAL WHILE

C'È DEADLOCK?
FINE = {TRUE, FALSE, FALSE}

Sì, i processi in deadlock sono P2 e P3

Uso degli algoritmi visti

- Quando usare gli algoritmi di rilevamento del deadlock?
- Dipende:
  - dalla frequenza con cui si verificano i deadlock
  - dal numero di processi mediamente coinvolti
- Si possono definire alcune euristiche:
  - effettuare la verifica **quando un processo che richiede una risorsa la trova occupata** (può capitare di frequente)
  - effettuare la verifica **quando l'utilizzo della CPU scende al di sotto di una certa soglia** o a intervalli fissi
- In generale, si può dire che esiste **un** processo responsabile del deadlock?
  - no, tutti i processi coinvolti in un ciclo sono corresponsabili

Che fare col deadlock?

- “Rompere” il deadlock - quando viene identificata una situazione di deadlock:

1) terminare i processi coinvolti: tutti, uno, qualcuno?
2) effettuare la prelazione delle risorse
3) riassegnare le risorse

Soluzione 1: terminazione

- Terminare un processo è costoso perché il lavoro da esso svolto svolto viene perduto
- Si possono adottare diverse politiche:
  - Terminare tutti i processi coinvolti

    molto oneroso!!!
  - Terminare un processo per volta fino alla risoluzione del deadlock

    occorre applicare l'algoritmo di rilevamento dopo l'abort di ciascun processo
- Abort di un processo: può comportare l'insorgere di problemi di consistenza in presenza di transazioni atomiche. In questo caso il SO deve effettuare il rollback delle transazioni interrotte

Come scegliere la vittima?

- Desiderio: scegliere il/i processo/i la cui terminazione è meno onerosa
- Problema: non esiste una misura precisa a cui fare riferimento
- Alcune misure di riferimento sono:
  - priorità dei processi (scelgo processi a bassa priorità)
  - tempo di computazione effettuata rapportato al tempo stimato di computazione residua
  - processo interattivo / processo batch
  - ...

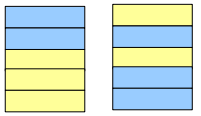
Soluzione 2: prelazione

- Idea: sottrarre successivamente risorse ad alcuni processi per assegnarle ad altri
- Anche in questo caso occorre identificare una vittima e la scelta è effettuata su criteri economici, la prelazione di risorse deve essere poco costosa
- Che fare dei processi a cui sono state sottratte risorse? Come per la terminazione potrebbe essere necessario riportare lo stato del sistema a una condizione di consistenza
- Occorre evitare che vengano sottratte risorse sempre allo stesso processo impedendogli così di continuare (insorgenza di starvation!!)

Che fare col deadlock?

- Far finta di niente:
  - è la tecnica più usata, poco costosa perché non richiede né risorse aggiuntive né l'attuazione di politiche particolari
  - quando un utente si accorge che si è verificato un deadlock, lo risolve manualmente

mappa dei principali concetti visti

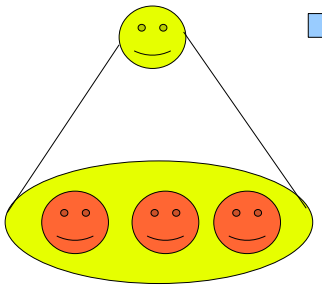


atomicità logica
transazioni
op. conflittuali

Da considerare ogni volta che si fa abort di un processo

Processo

Multi-tasking



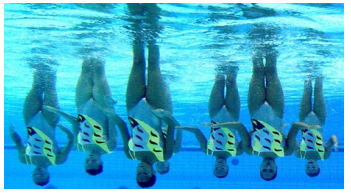
Thread

eseguono in //

cooperazione
concorrono a un risultato

competizione
uso risorse

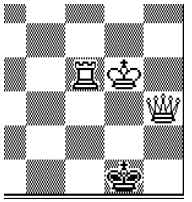
scambio dati



sincronizzazione

starvation
priorità

deadlock
stallo



memoria condivisa
scambio messaggi

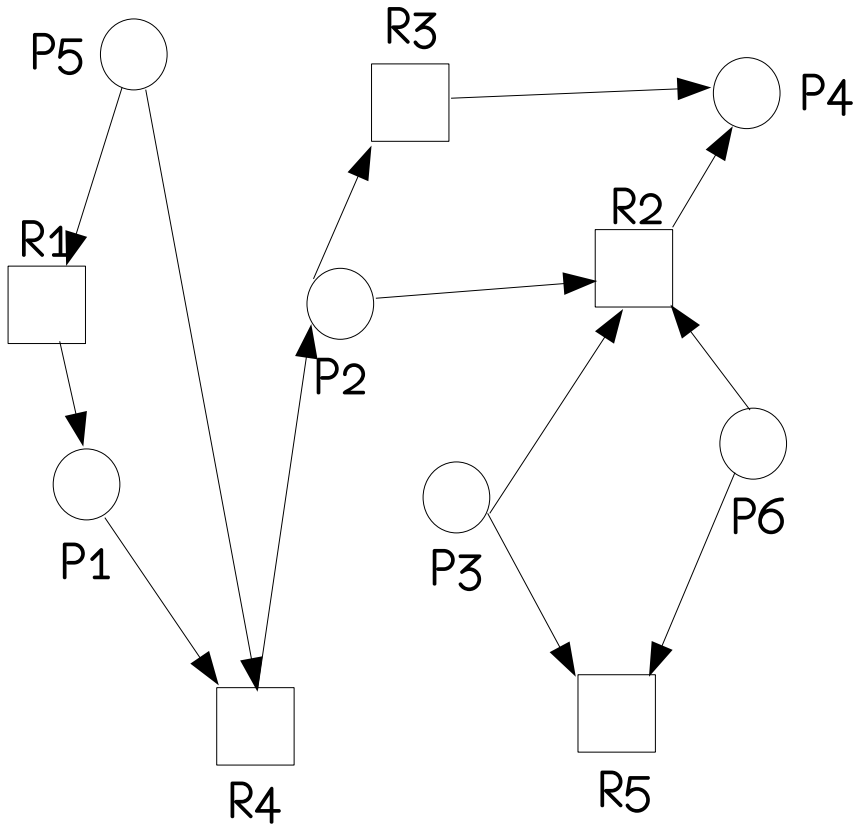
sezione critica
semafori
monitor

Esercizi

- Grafo allocazione risorse – grafo di attesa – presenza di deadlock
- Evoluzioni dello stato di allocazione delle risorse e deadlock
- Grafi di allocazione con archi di reclamo: generazione di stati sicuri e non sicuri
- Presenza di deadlock in caso di risorse con istanze multiple
- Deadlock avoidance: Algoritmo del banchiere
- Transazioni equivalenti

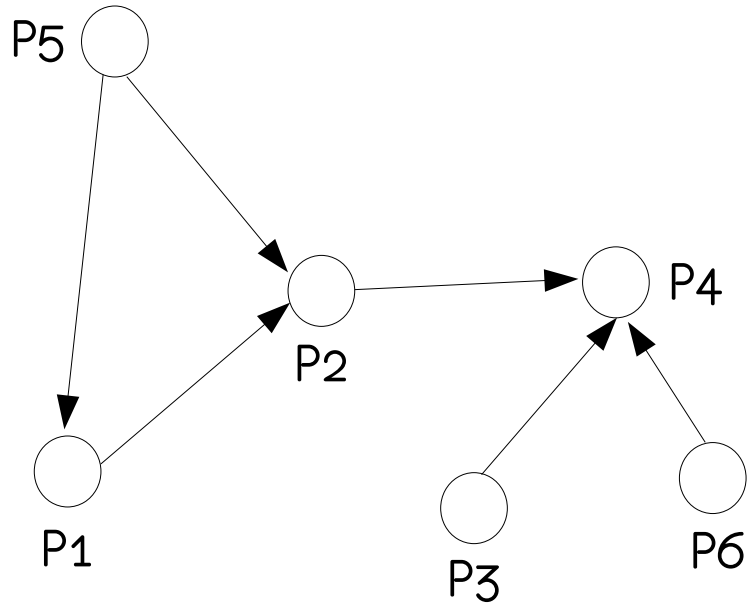
esercizio

Si consideri il seguente grafo di assegnazione delle risorse, trasformarlo in un grafo di attesa e verificare se vi è deadlock o meno motivando la risposta



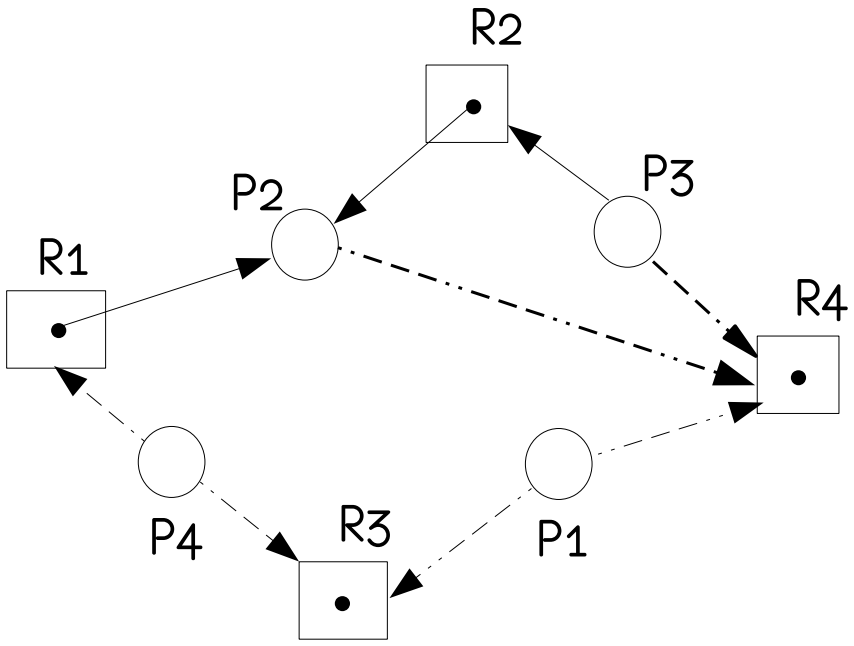
soluzione

Soluzione:
non c'è deadlock perché il grafo di attesa non contiene cicli

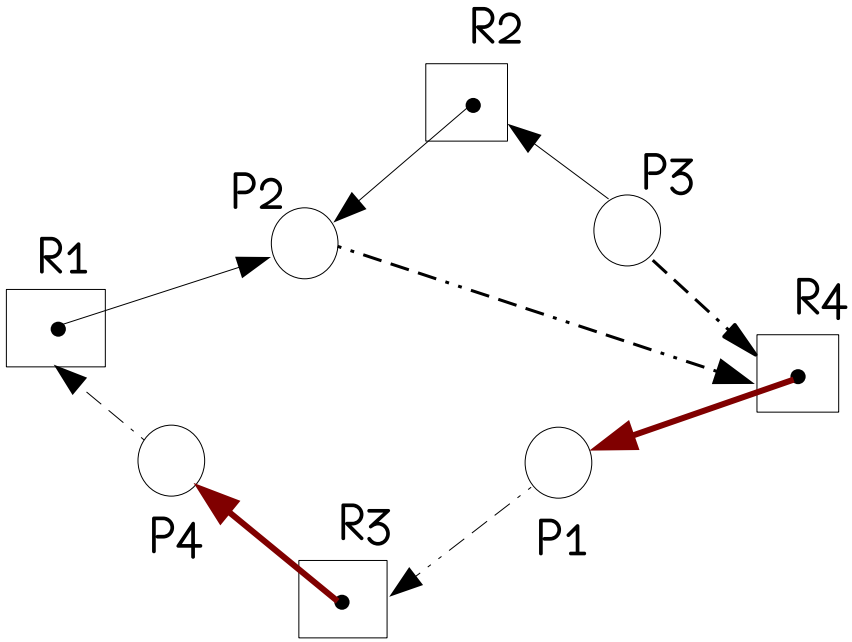


esercizio

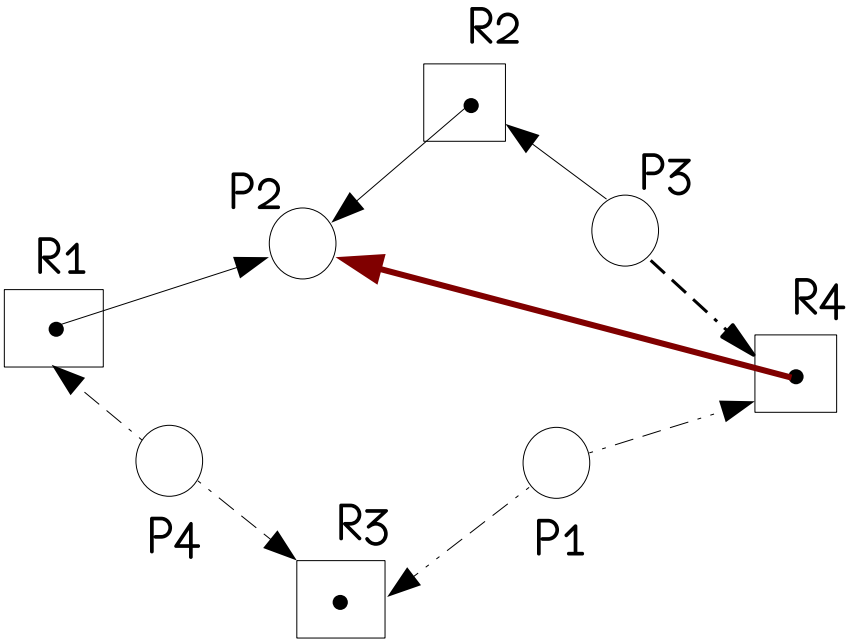
Dato il seguente grafo di assegnazione delle risorse con archi di reclamo individuare un'assegnazione che porta a uno stato non sicuro e un'assegnazione sicura



soluzione



Stato non sicuro: il grafo contiene un ciclo che coinvolge P2, P1, P4



Stato sicuro: il grafo non contiene alcun ciclo

esercizio

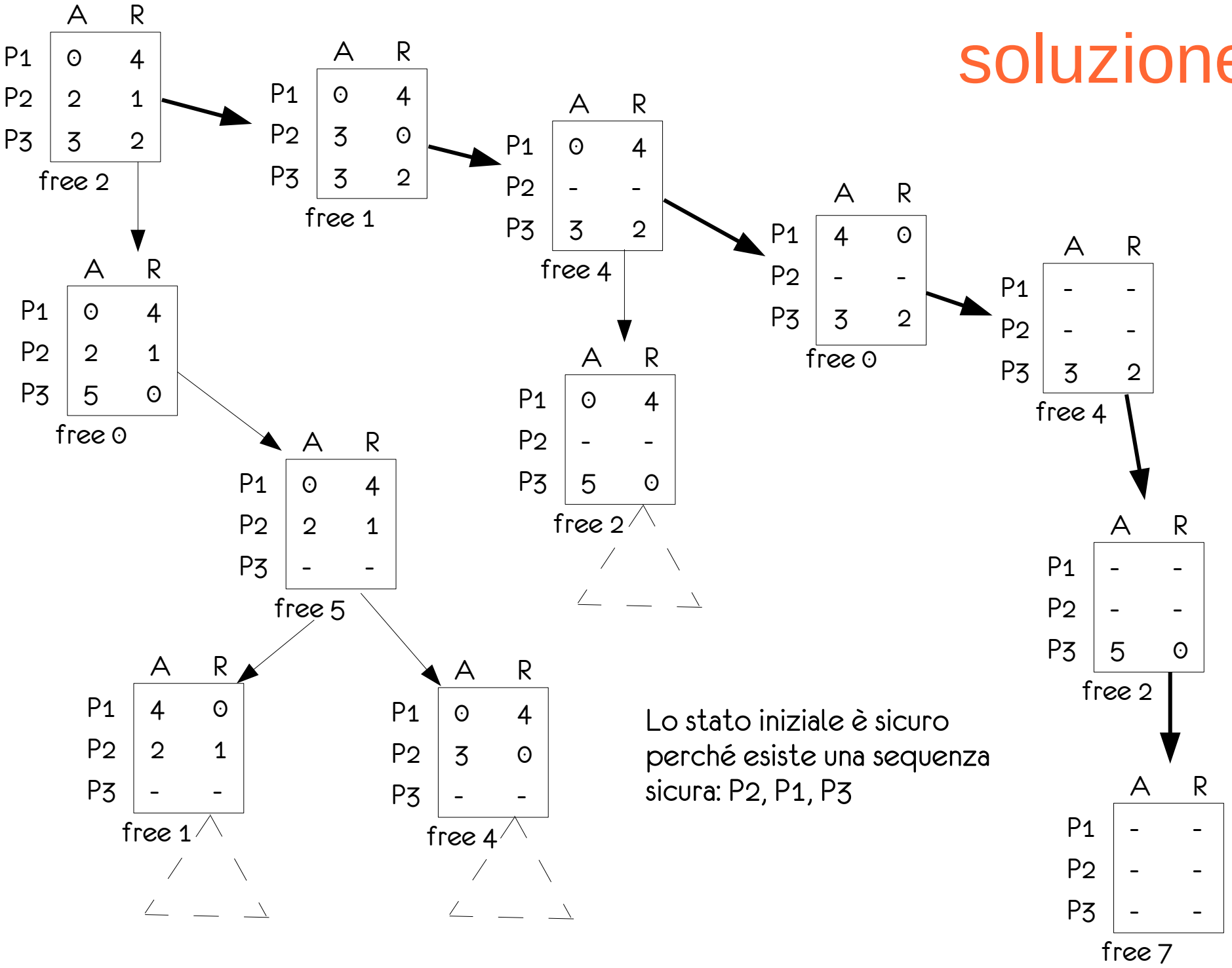
Costruire tutte le possibili evoluzioni dello stato di allocazione delle risorse riportato di seguito. La colonna A indica le risorse in possesso dei processi, R indica il numero di risorse necessarie ma non ancora assegnate, free indica il numero di risorse attualmente libere. Al termine dire se lo stato iniziale è sicuro motivando la risposta.

Stato iniziale

	A	R
P1	0	4
P2	2	1
P3	3	2

free 2

soluzione



Lo stato iniziale è sicuro perché esiste una sequenza sicura: P2, P1, P3

esercizio

Dati 3 processi e 3 classi di risorse con istanze multiple applicare l'algoritmo di identificazione del deadlock allo stato descritto nel seguito; in caso di deadlock si specifichi quali processi sono coinvolti

Disponibili == {0, 0, 1}

Assegnate[3][3] ==

1	0	0
0	1	1
2	0	1

Richieste[3][3] ==

1	2	0
0	0	0
0	1	0

soluzione

Disponibili == {0, 0, 1}

Assegnate[3][3] ==

1	0	0
0	1	1
2	0	1

Richieste[3][3] ==

1	1	3
0	0	0
0	1	0

Lavoro = Disponibili
Fine = {false, false, false}

c'è un i | !Fine[i] && richieste[i]<Lavoro? Si, richieste[2]
Lavoro = Lavoro + Assegnate[2] = {0,1,2}
Fine = {false, true, false}

c'è un i | !Fine[i] && richieste[i]<Lavoro? Si, richieste[3]
Lavoro = Lavoro + Assegnate[3] = {2,1,3}
Fine = {false, true, true}

c'è un i | !Fine[i] && richieste[i]<Lavoro? Si, richieste[1]
Lavoro = Lavoro + Assegnate[1] = {3,1,3}
Fine = {true, true, true}

Tutti i processi hanno Fine a true quindi non c'è deadlock

esercizio

Dati 3 processi e 3 classi di risorse con istanze multiple applicare l'algoritmo di identificazione del deadlock allo stato descritto nel seguito; in caso di deadlock si specifichi quali processi sono coinvolti

Disponibili == {0, 0, 1}

Assegnate[3][3] ==

1	0	0
0	1	1
2	0	1

Richieste[3][3] ==

1	2	0
0	1	0
0	1	0

soluzione

Disponibili == {0, 0, 1}

Assegnate[3][3] ==

1	0	0
0	1	1
2	0	1

Richieste[3][3] ==

1	1	3
0	1	0
0	1	0

Lavoro = Disponibili
Fine = {false, false, false}

c'è un i | !Fine[i] && richieste[i]<Lavoro? Si, richieste[2]

No, infatti:
Richieste[1] = {1,1,3} non è < {0,0,1}
Richieste[2] = {0,1,0} non è < {0,0,1}
Richieste[3] = {0,1,0} non è < {0,0,1}

C'è deadlock, i processi coinvolti sono P1, P2 e P3
(hanno tutti Fine a false)

esercizio

Applicare l'algoritmo del banchiere per decidere se lo stato riportato qui di seguito è sicuro o meno. Motivare la risposta.

	A	R
P1	5	1
P2	2	2
P3	1	8

free 1

soluzione

	A	R
P1	5	1
P2	2	2
P3	1	8

free 1

```
lavoro=disponibili = 1
necessarie = {1, 2, 8}
assegnate  = {5, 2, 1}
fine = {false, false, false}

c'è i | !Fine[i] && necessarie[i]<=lavoro? Si, i = 1
lavoro = lavoro + assegnate[i] = 6
fine = {true, false, false}

c'è i | !Fine[i] && necessarie[i]<=lavoro? Si, i = 2
lavoro = lavoro + assegnate[i] = 8
fine = {true, true, false}

c'è i | !Fine[i] && necessarie[i]<=lavoro? Si, i = 3
lavoro = lavoro + assegnate[i] = 9
fine = {true, true, true}

lo stato è sicuro perché tutti i processi hanno
fine a true
```

esercizio

Date le transazioni T1, T2 e T3 riportate nel seguito dire, motivando la risposta, se l'esecuzione riportata è equivalente all'esecuzione sequenziale di T2, T1, T3 (nell'ordine indicato)

T1
read(A)
write(C)
write(A)

T2
read(C)
read(A)
write(B)
write(C)

T3
read(B)
read(A)
write(B)
read(C)
write(A)

T1	T2	T3
	read(C)	
		read(B)
read(A)		
	read(A)	
write(C)		
write(A)		
		read(A)
		write(B)
	write(B)	
		read(C)
		write(A)
	write(C)	

soluzione 1/2

T2	T1	T3
read(C)		
read(A)		
write(B)		
write(C)		
	read(A)	
	write(C)	
	write(A)	
		read(B)
		read(A)
		write(B)
		read(C)
		write(A)

T2	T1	T3
read(C)		
		read(B)
	read(A)	
read(A)		
	write(C)	
	write(A)	
		read(A)
		write(B)
write(B)		
		read(C)
		write(A)
write(C)		

Bisogna provare a trasformare l'esecuzione delle 3 transazioni in sequenza (sulla sinistra) in un'esecuzione in cui le operazioni sono eseguite nell'ordine dato a destra. Per farlo occorre vedere, coppia x coppia di operazioni di transazioni diverse, contigue nel tempo e da spostare, se sono conflittuali: se no, si procede, se sì le due sequenze non sono equivalenti.

soluzione 2/2

T2	T1	T3
read(C)		
read(A)		
write(B)		
write(C)		
	read(A)	
	write(C)	
	write(A)	
		read(B)
		read(A)
		write(B)
		read(C)
		write(A)

T2	T1	T3
read(C)		
		read(B)
	read(A)	
read(A)		
	write(C)	
	write(A)	
		read(A)
		write(B)
write(B)		
		read(C)
		write(A)
write(C)		

Per es. read(C) di T2 non cambia posizione, procediamo.
read(B) di T3 diventa la seconda operazione eseguita: per avere l'equivalenza, tale operazione non deve essere in conflitto con nessuna di quelle in blu. In effetti non è in conflitto con le 3 operazioni di T1 e neppure con write(C) di T2, però è in conflitto con write(B) di T2, quindi le due sequenze non sono equivalenti.