



deadlock

capitolo 7 del libro (VII ed.)

Introduzione

- **Deadlock:** situazione per cui un insieme di processi sono fermi in attesa di un evento che solo uno dei processi appartenenti all'insieme stesso potrebbe causare
- Noi vedremo il problema del deadlock in relazione al problema della gestione di risorse, anche se si tratta di un **problema più generale**

Introduzione

- **Deadlock:** situazione per cui un insieme di processi sono fermi in attesa di un evento che solo uno dei processi appartenenti all'insieme stesso potrebbe causare
- **Problema della gestione delle risorse:**
 - un sistema fisico può essere visto come un insieme di **risorse**
 - ogni risorsa può essere presente in un certo numero di **istanze equivalenti**, es:
 - CPU: 2 (sistema biprocessore)
 - stampanti: 3
 - CD-reader: 1

Introduzione

- Un processo che vuole usare N istanze di una certa risorsa deve farne richiesta al gestore delle risorse, cioè al SO
- Si distinguono 3 fasi:

- **richiesta:** può causare attesa ●

- uso

- **rilascio** ●

sono effettuati tramite system call

The diagram consists of a horizontal line connecting the red dots after 'richiesta' and 'rilascio'. From the midpoint of this line, a vertical line descends to a red arrowhead pointing at the text 'sono effettuati tramite system call'.

-
- Il SO tiene traccia di quali risorse sono assegnate a quale processo
 - Richiesta/rilascio: tramite **system call ad hoc** (es. open e close di file) oppure tramite **strumenti di sincronizzazione** (es. semafori)

Condizioni al deadlock

- Es. di deadlock fra due processi: P1 detiene l'unico lettore di fotografie e vuole la stampante per stampare delle foto, P2 ha la stampante e vuole il lettore perché deve, anch'esso, stampare delle foto ...
- **Condizioni necessarie al deadlock**

<1> **ME**: risorse non condivisibili

<2> **possesso e attesa**: un processo attende le risorse non disponibili, anche detenendo già il possesso di alcune delle risorse a lui necessarie

<3> **no prelazione**: il rilascio non viene forzato

<4> **attesa circolare**: siano i processi in questione P_1, \dots, P_n , allora P_1 attende risorse da P_2 , che attende risorse da P_3 , ecc. e P_n attende risorse da P_1

basta che una **non** sia vero per evitare il deadlock

Esempio



prendi aceto
prendi sale
prendi olio

i cuochi prendono le risorse, non
condivisibili, una per volta

non rilasciano le risorse in loro
possesso

si aspettano a vicenda ... per sempre



prendi sale
prendi olio
prendi aceto



prendi olio
prendi aceto
prendi sale



In codice

Cuoco1

(1) P(olio);
(2) P(aceto);
(3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

(1) P(sale);
(2) P(olio);
(3) P(aceto);

... cucina ...

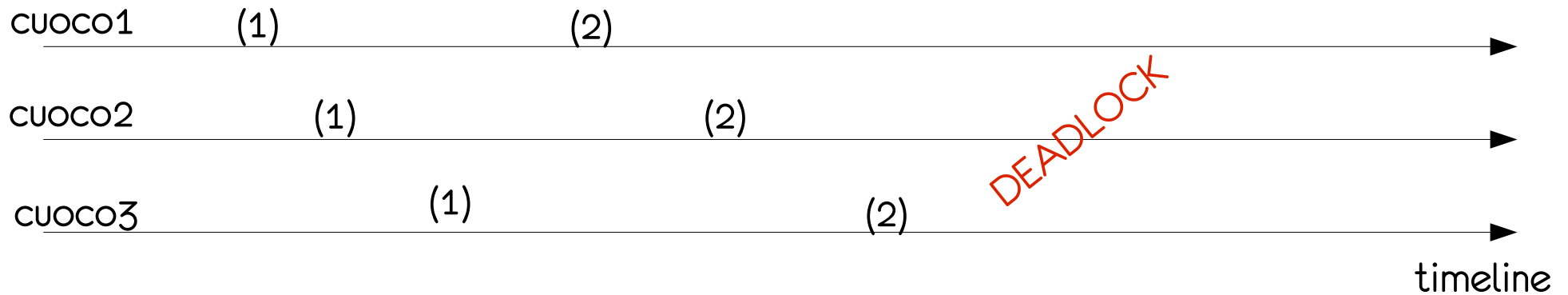
rilascia risorse

Cuoco3

(1) P(aceto);
(2) P(sale);
(3) P(olio);

... cucina ...

rilascia risorse



In codice

Cuoco1

(1) P(olio);
(2) P(aceto);
(3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

(1) P(sale);
(2) P(olio);
(3) P(aceto);

... cucina ...

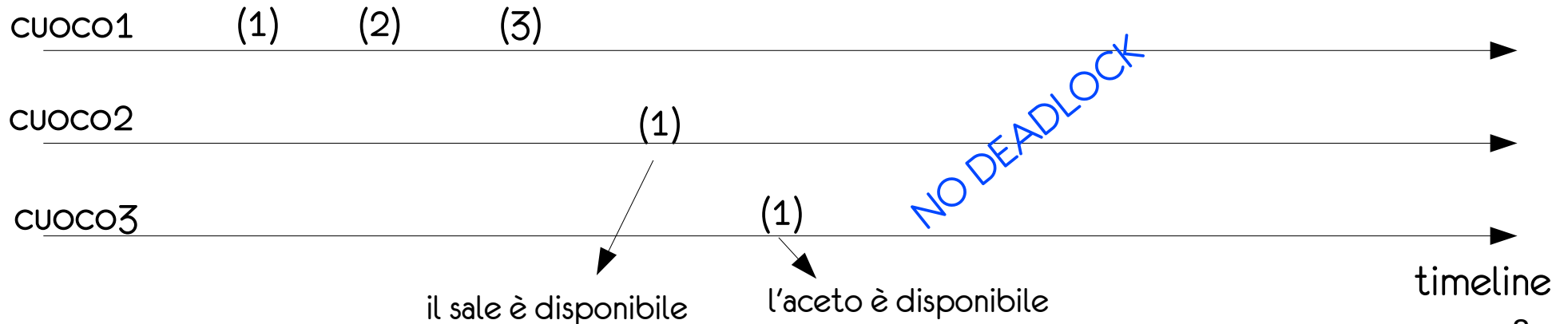
rilascia risorse

Cuoco3

(1) P(aceto);
(2) P(sale);
(3) P(olio);

... cucina ...

rilascia risorse

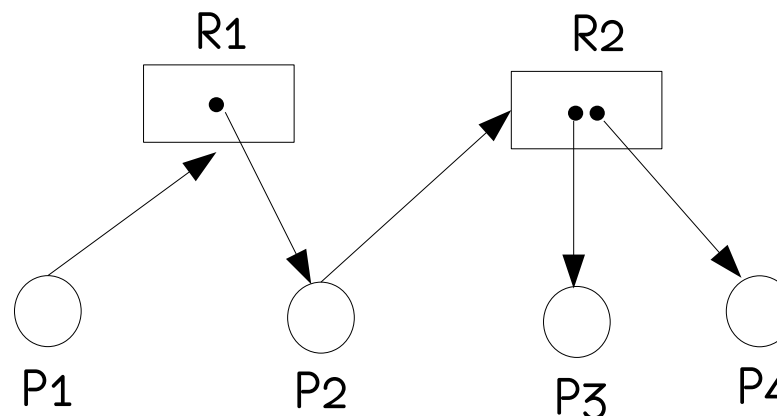


Grafo di assegnazione delle risorse

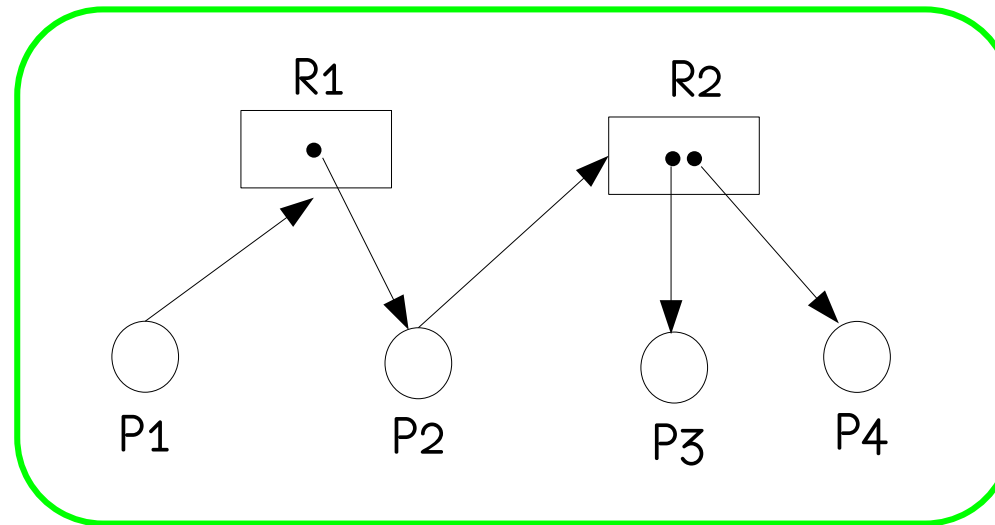
- Rappresentazione delle assegnazioni che permette di rilevare situazioni di deadlock
- È un grafo $G = \langle V, E \rangle$ tale per cui:
 - V è l'insieme dei vertici ed è partizionato in due sottoinsiemi P ed R , $P \cap R = \emptyset$:
 - $P = \{P_1, \dots, P_n\}$ è l'insieme di tutti i processi del sistema
 - $R = \{R_1, \dots, R_m\}$ è l'insieme di tutte le classi di risorse del sistema
 - E è l'insieme degli archi:
 - Un'arco direzionato da R_i a P_j , $R_i \rightarrow P_j$, indica che una risorsa di classe R_i è stata assegnata al processo P_j (arco di assegnazione)
 - Un'arco direzionato da P_j a R_i , $P_j \rightarrow R_i$, indica che il processo P_j ha richiesto ed è in attesa di una risorsa di tipo R_i (arco di richiesta)

Rappresentazione grafica del grafo di assegnazione delle risorse

- Ogni processo P_i è rappresentato da un cerchietto
- Ogni classe di risorsa R_i è rappresentata da un rettangolo contenente tanti puntini quante sono le sue istanze
- Un **arco di assegnazione** parte da una specifica risorsa (un puntino) ed è diretto a un processo
- Un **arco di richiesta** parte da un processo e termina a un rettangolo (la classe della risorsa)



Notazione grafica



P ○ Processo P

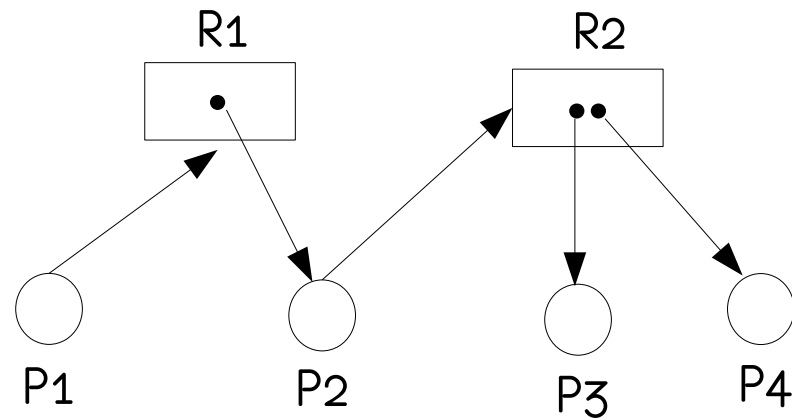
R
□ Classe di risorsa R

R
□ ● ● Classe di risorsa R con due istanze

P ○ → R □ ● ● Arco di richiesta

P ○ ← R □ ● ● Arco di assegnazione

Esempio



Abbiamo 4 processi e 2 classi di risorse
R1 ha una sola istanza, R2 ha 2 istanze
P1 ha richiesto una risorsa di tipo R1
L'unica risorsa di questa classe è assegnata a P2,
che ha richiesto una risorsa di classe R2
Nessuna di queste è libera al momento,
essendo esse assegnate a P3 e P4

I 3 cuochi

Cuoco1

(1) P(olio);
(2) P(aceto);
(3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

(1) P(sale);
(2) P(olio);
(3) P(aceto);

... cucina ...

rilascia risorse

Cuoco3

(1) P(aceto);
(2) P(sale);
(3) P(olio);

... cucina ...

rilascia risorse

Quante classi di risorse abbiamo?

Quante istanze per ogni classe?

Quanti processi?

Chi detiene quale risorsa?

Chi richiede quale risorsa?

informazioni di tipo statico

informazioni note a run-time
il grafo cattura l'andamento
dell'esecuzione

I 3 cuochi

Cuoco1

(1) P(olio);
(2) P(aceto);
(3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

(1) P(sale);
(2) P(olio);
(3) P(aceto);

... cucina ...

rilascia risorse

Cuoco3

(1) P(aceto);
(2) P(sale);
(3) P(olio);

... cucina ...

rilascia risorse

Quante classi di risorse abbiamo?

3

Quante istanze per ogni classe?

1

Quanti processi?

3

Chi detiene quale risorsa?

Chi richiede quale risorsa?

I 3 cuochi

Cuoco1

(1) P(olio);
(2) P(aceto);
(3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

(1) P(sale);
(2) P(olio);
(3) P(aceto);

... cucina ...

rilascia risorse

Cuoco3

(1) P(aceto);
(2) P(sale);
(3) P(olio);

... cucina ...

rilascia risorse

Quante classi di risorse abbiamo?

Quante istanze per ogni classe?

Quanti processi?

Chi detiene quale risorsa?

Chi richiede quale risorsa?



[a] Quando un processo P esegue la richiesta di una risorsa aggiungo un arco di richiesta

[b] Quando P ottiene la risorsa, cancello tale arco e ne inserisco uno di assegnazione

[c] Quando P rilascia la risorsa l'arco di assegnazione viene rimosso

I 3 cuochi

Cuoco1

(1) P(olio);
(2) P(aceto);
(3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

(1) P(sale);
(2) P(olio);
(3) P(aceto);

... cucina ...

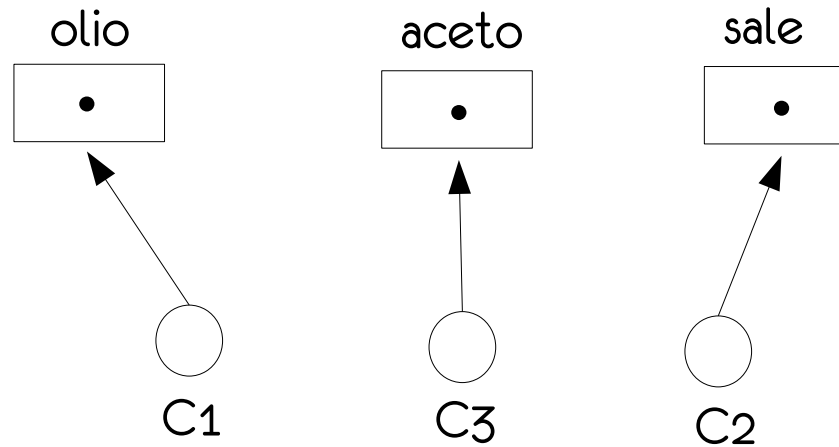
rilascia risorse

Cuoco3

(1) P(aceto);
(2) P(sale);
(3) P(olio);

... cucina ...

rilascia risorse



I 3 cuochi

Cuoco1

(1) P(olio);
(2) P(aceto);
(3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

(1) P(sale);
(2) P(olio);
(3) P(aceto);

... cucina ...

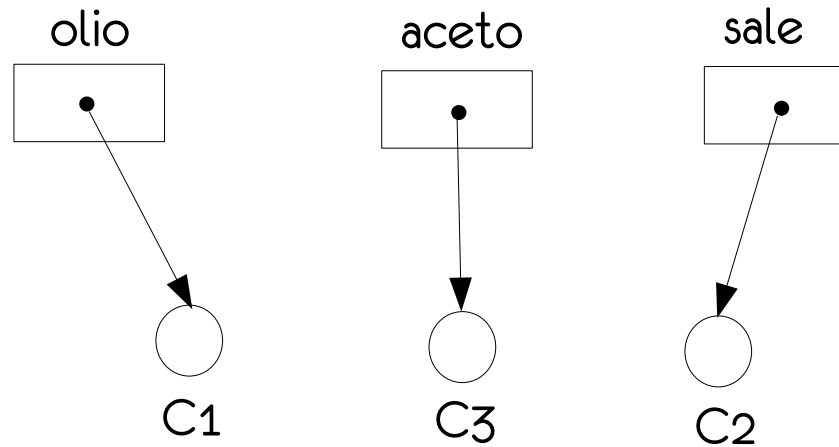
rilascia risorse

Cuoco3

(1) P(aceto);
(2) P(sale);
(3) P(olio);

... cucina ...

rilascia risorse



I 3 cuochi

Cuoco1

(1) P(olio);
(2) P(aceto);
(3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

(1) P(sale);
(2) P(olio);
(3) P(aceto);

... cucina ...

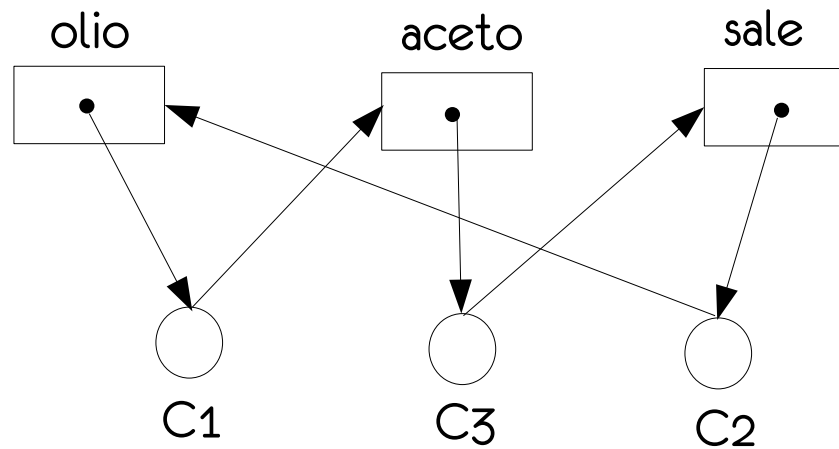
rilascia risorse

Cuoco3

(1) P(aceto);
(2) P(sale);
(3) P(olio);

... cucina ...

rilascia risorse



I 3 cuochi

Cuoco1

(1) P(olio);
(2) P(aceto);
(3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

(1) P(sale);
(2) P(olio);
(3) P(aceto);

... cucina ...

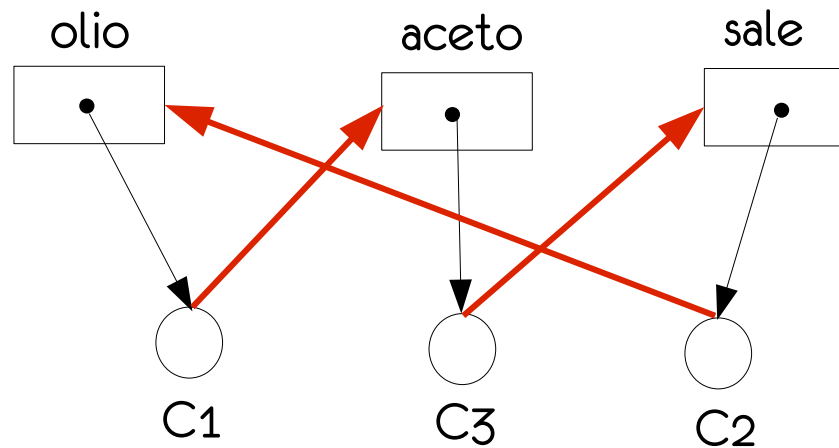
rilascia risorse

Cuoco3

(1) P(aceto);
(2) P(sale);
(3) P(olio);

... cucina ...

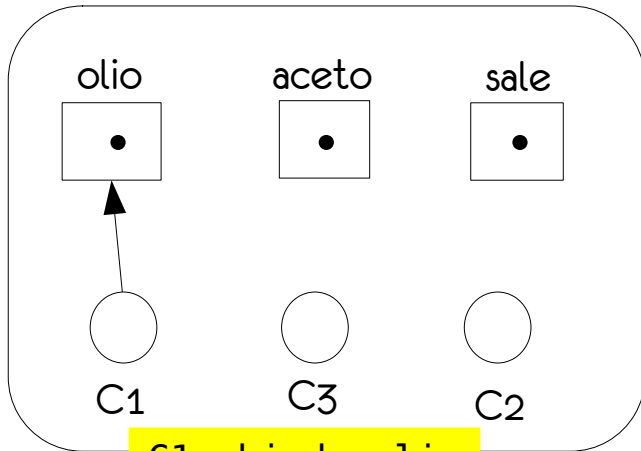
rilascia risorse



C1 aspetta C3, che aspetta C2, che aspetta C1: **deadlock**

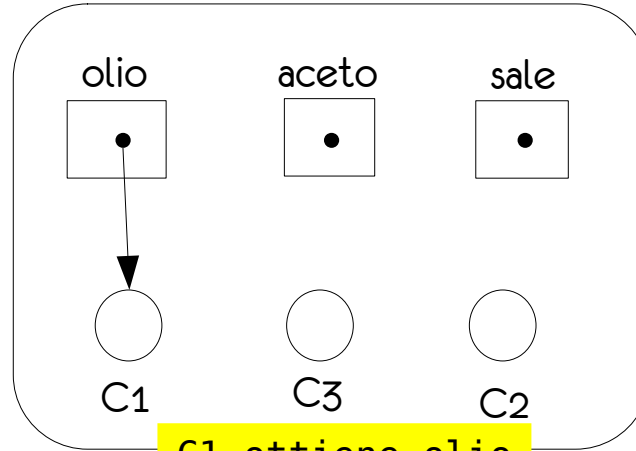
I 3 cuochi

1



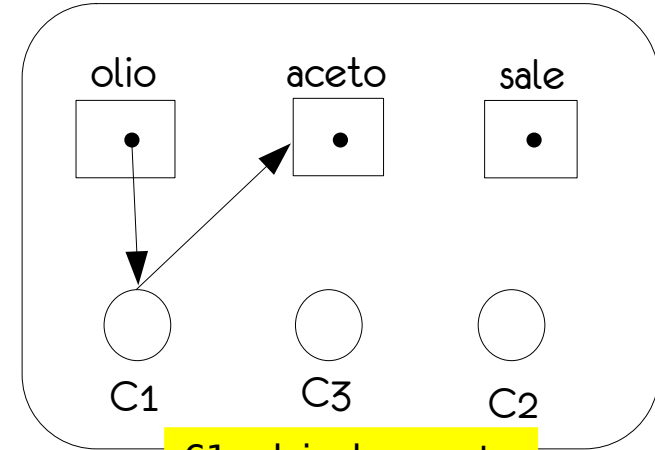
C1 chiede olio

2

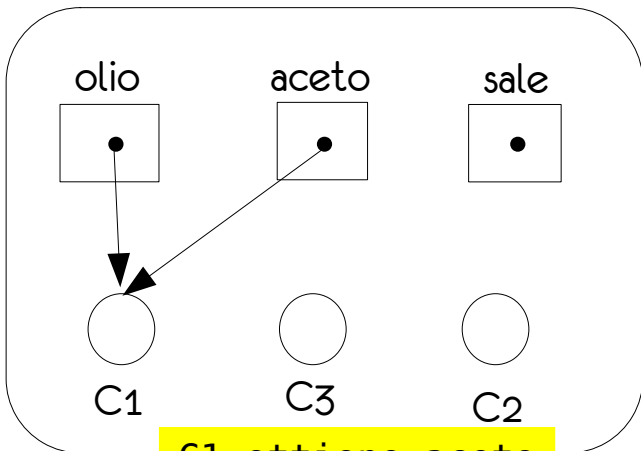


C1 ottiene olio

3

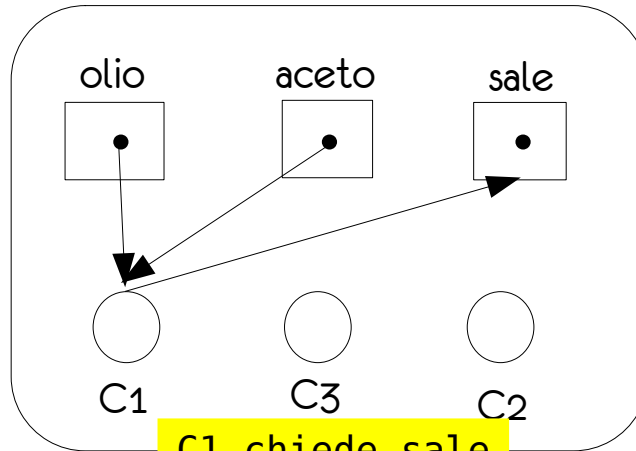


C1 chiede aceto



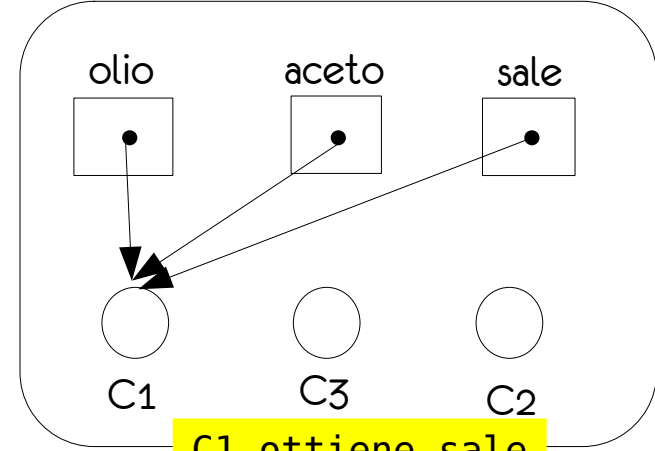
C1 ottiene aceto

4



C1 chiede sale

5



C1 ottiene sale

6

eccetera ... esecuzione senza deadlock

Uso del grafo di assegnazione

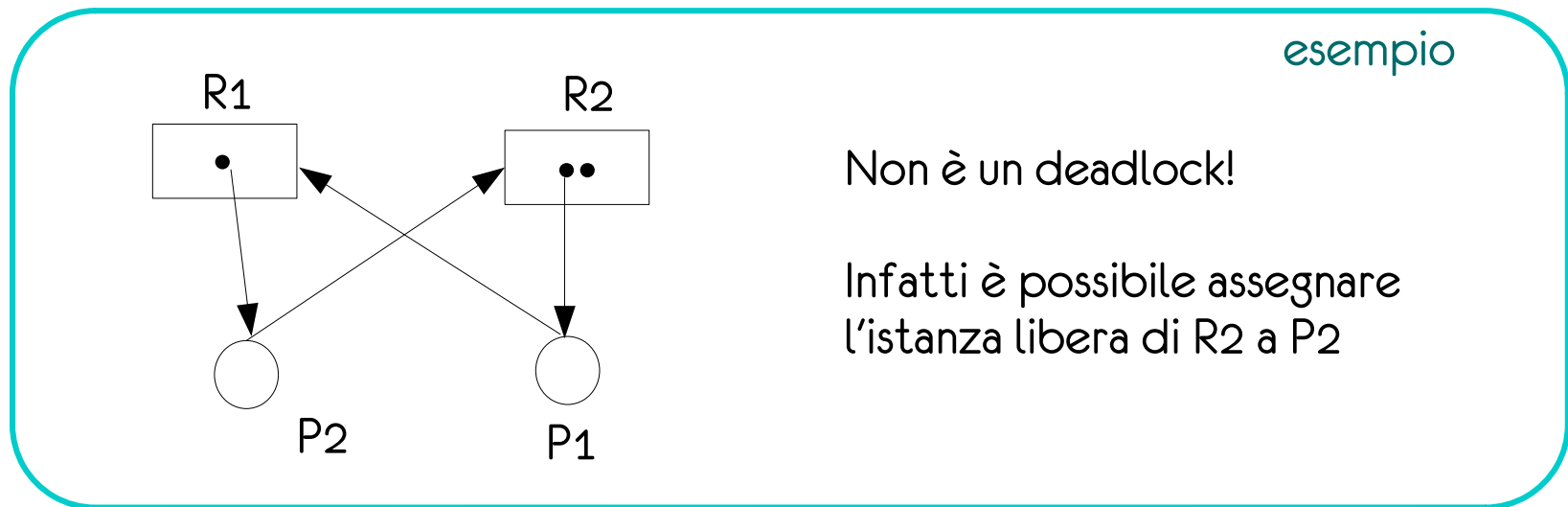
- Se il grafo **non** contiene cicli **non** c'è deadlock
- La presenza di un ciclo è condizione **necessaria** ma **non sufficiente** per avere deadlock

Uso del grafo di assegnazione

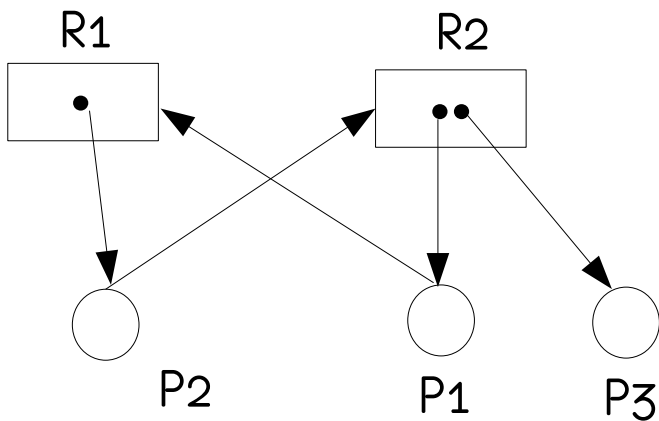
- Se il grafo **non** contiene cicli **non** c'è deadlock
- La presenza di un ciclo è condizione **necessaria ma non sufficiente** per avere deadlock:
 - Se il grafo contiene un ciclo che comprende risorse aventi tutte una sola istanza, allora c'è deadlock
 - Se il ciclo comprende risorse aventi più di una istanza non è detto che vi sia deadlock: **ciclo come risorsa necessaria ma non sufficiente**. Basta che una delle richieste sia soddisfacibile per rompere il ciclo

Uso del grafo di assegnazione

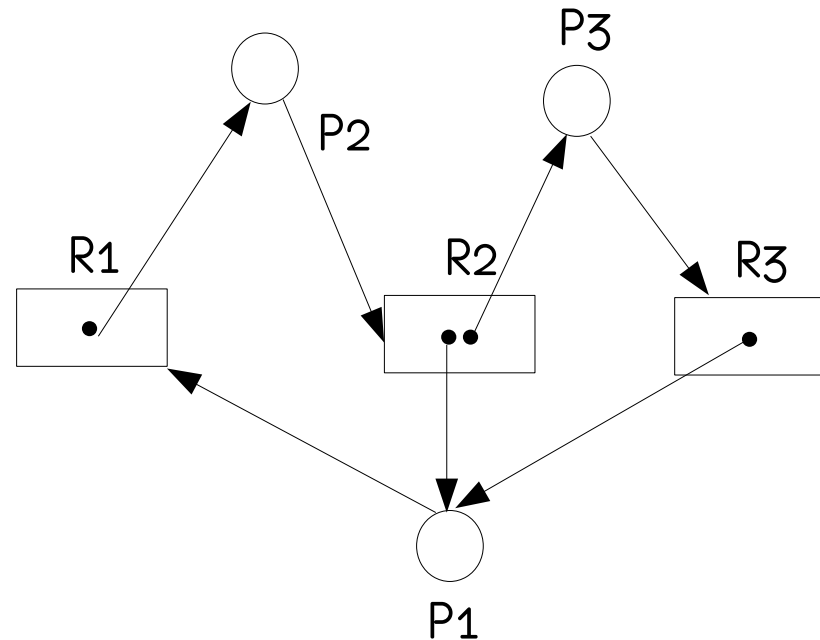
- Se il ciclo comprende risorse aventi più di una istanza non è detto che vi sia deadlock: **ciclo come risorsa necessaria ma non sufficiente**. Basta che una delle richieste sia soddisfacibile per rompere il ciclo



Altri esempi



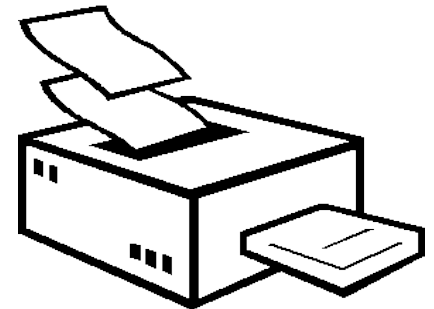
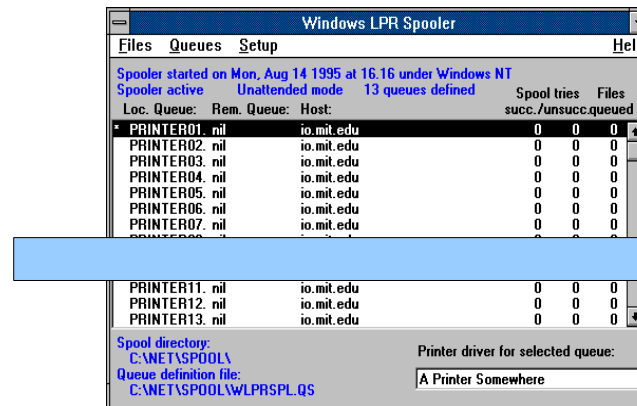
Deadlock? No
Appena P3 libera R2
P2 riparte



Deadlock? Si
P1 aspetta R1 che è di P2
P2 aspetta R2, le cui istanze
sono di P1 e P3 e
P3 aspetta R3 che è di P1

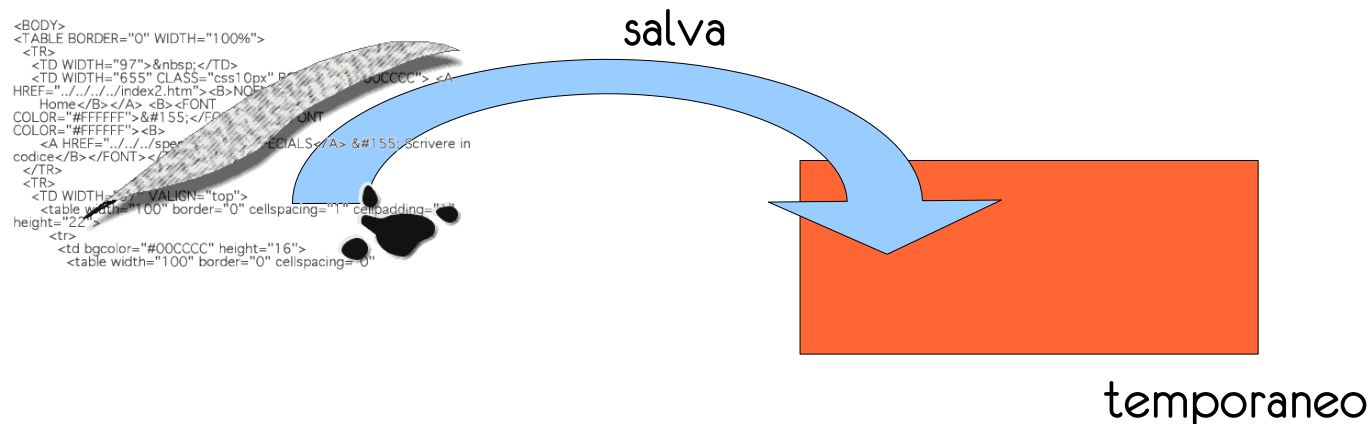
Un caso reale: spooling

- Un processo deve stampare un documento: la stampante gestisce una pagina per volta, quindi il processo dovrebbe attendere la terminazione della stampa della pagina corrente prima dell'invio alla stampante della successiva
- Problema: lentezza!!
- Per migliorare l'esecuzione del processo utente si introduce uno spooler



Spooler

- Uno spooler è un programma che funge da intermediario fra i processi di stampa e la stampante (o altri generi di device)
- Un processo di stampa può terminare subito dopo aver inviato il proprio documento allo spooler
- L'invio può avvenire in modi diversi:
 - caso (1): il documento viene salvato in un file temporaneo poi elaborato dallo spooler



Scenario

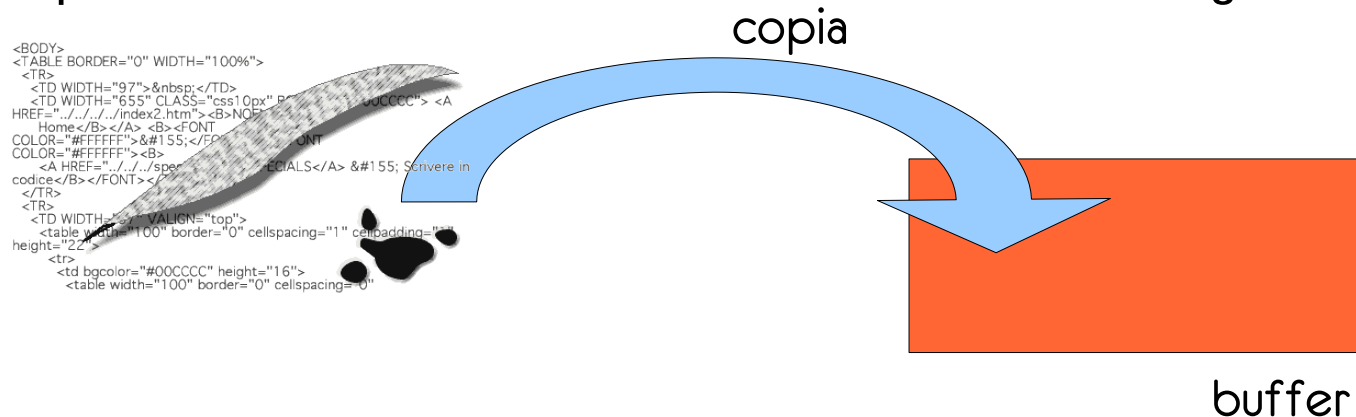
- Molti sistemi di spooling gestiscono solo documenti completamente codificati
- Quindi un processo di stampa deve per es.:
 - convertire il documento in un formato di stampa
 - salvare il risultato in un file temporaneo
- **Risorsa condivisa: memoria**
- Cosa succede se si esaurisce la memoria e nessun processo di stampa ha terminato il proprio lavoro?

Scenario

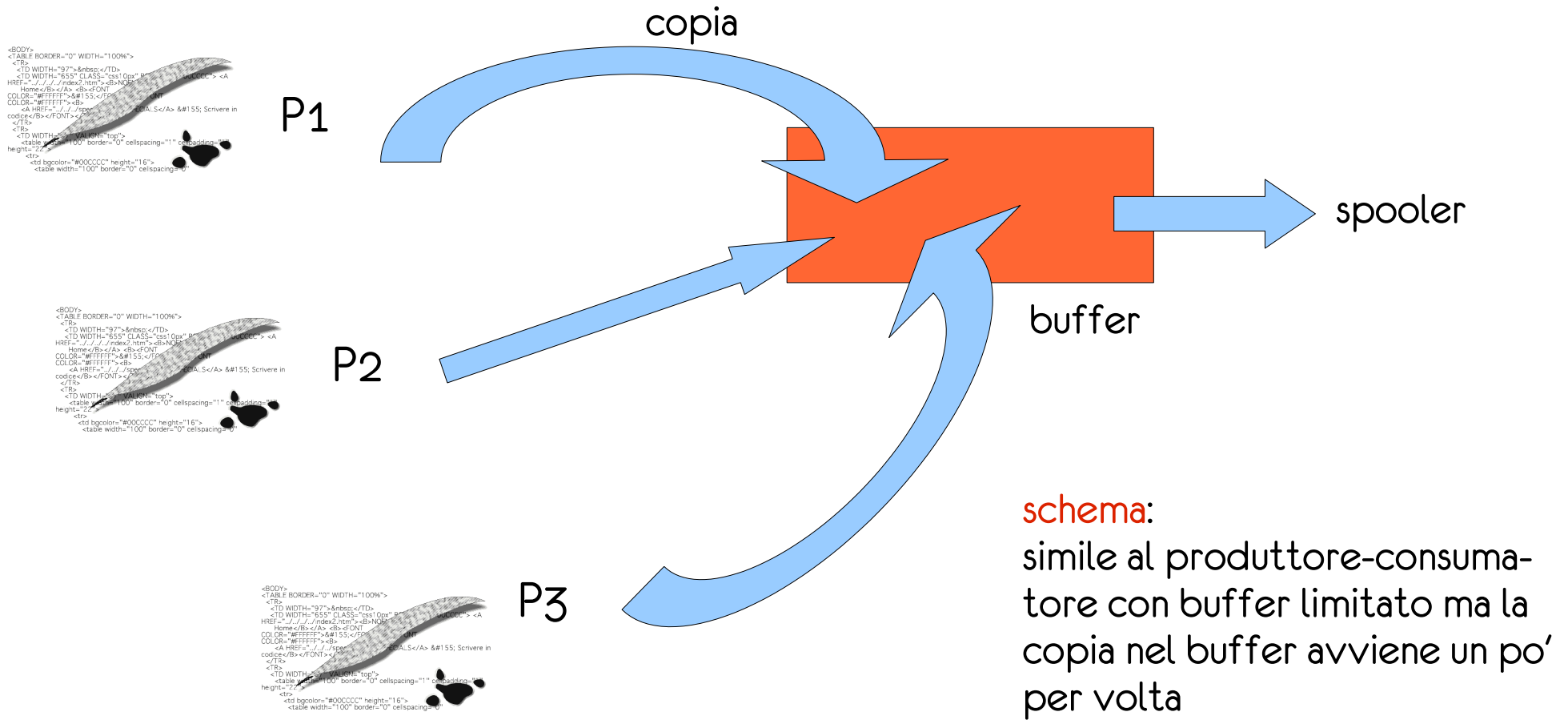
- Molti sistemi di spooling gestiscono solo documenti prodotti in modo completo
- Quindi un processo di stampa deve per es.:
 - convertire il documento in un formato di stampa
 - salvare il risultato in un file temporaneo
- **Risorsa condivisa: memoria**
- Cosa succede se si esaurisce la memoria e nessun processo di stampa ha terminato il proprio lavoro? Deadlock!
 - I processi di stampa **acquisiscono la memoria un po' per volta**, quindi attendono la memoria mancante
 - lo spooler potrebbe liberare la memoria processando un documento ma non lo fa perché **nessun documento è completo**

Spooler

- Consideriamo un'implementazione alternativa dello spooler che diventa ...
- ... un programma che funge da intermediario fra i processi di stampa e la stampante, legge da un'area di memoria predefinita e di dimensione limitata (buffer) i documenti da stampare e interagisce con la stampante.
- Un processo di stampa può terminare dopo aver inviato il proprio documento allo spooler
- L'invio avviene in questo modo, caso (2): il documento viene copiato nel buffer man mano che viene generato

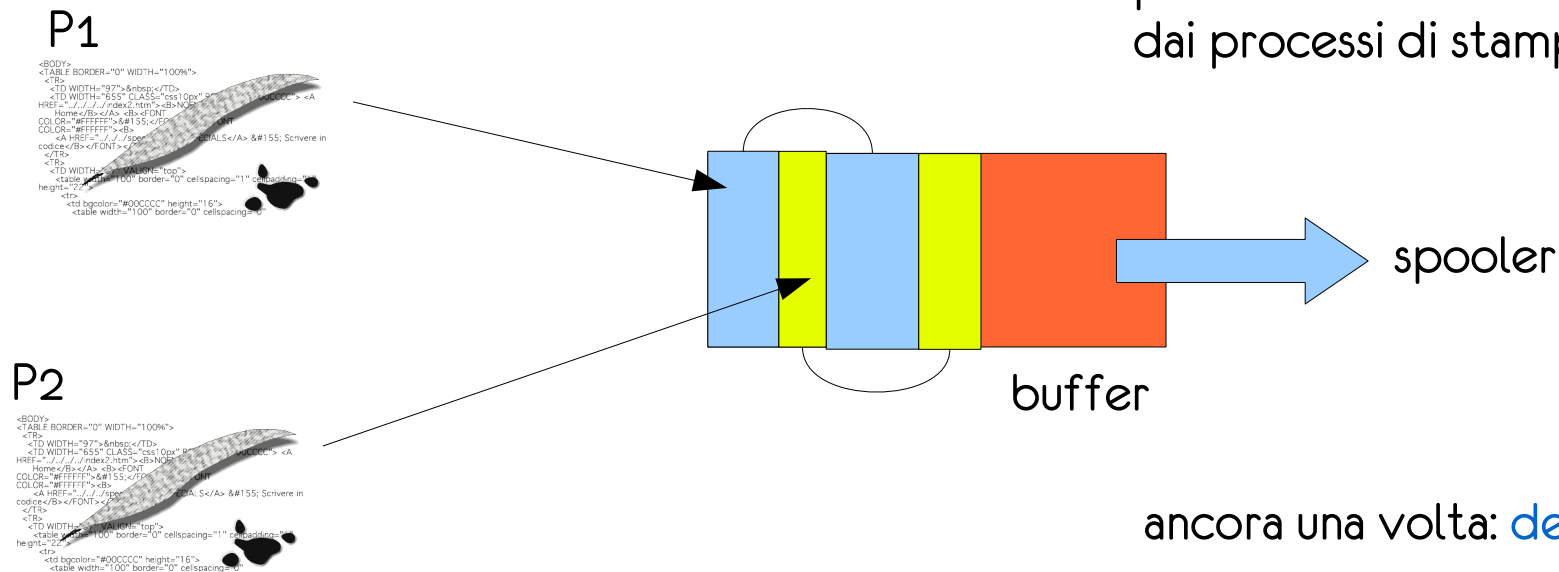


Scenario



Scenario

Problema: cosa succede se la porzione di buffer libera non basta a contenere nessuna delle porzioni di documento prodotte dai processi di stampa?



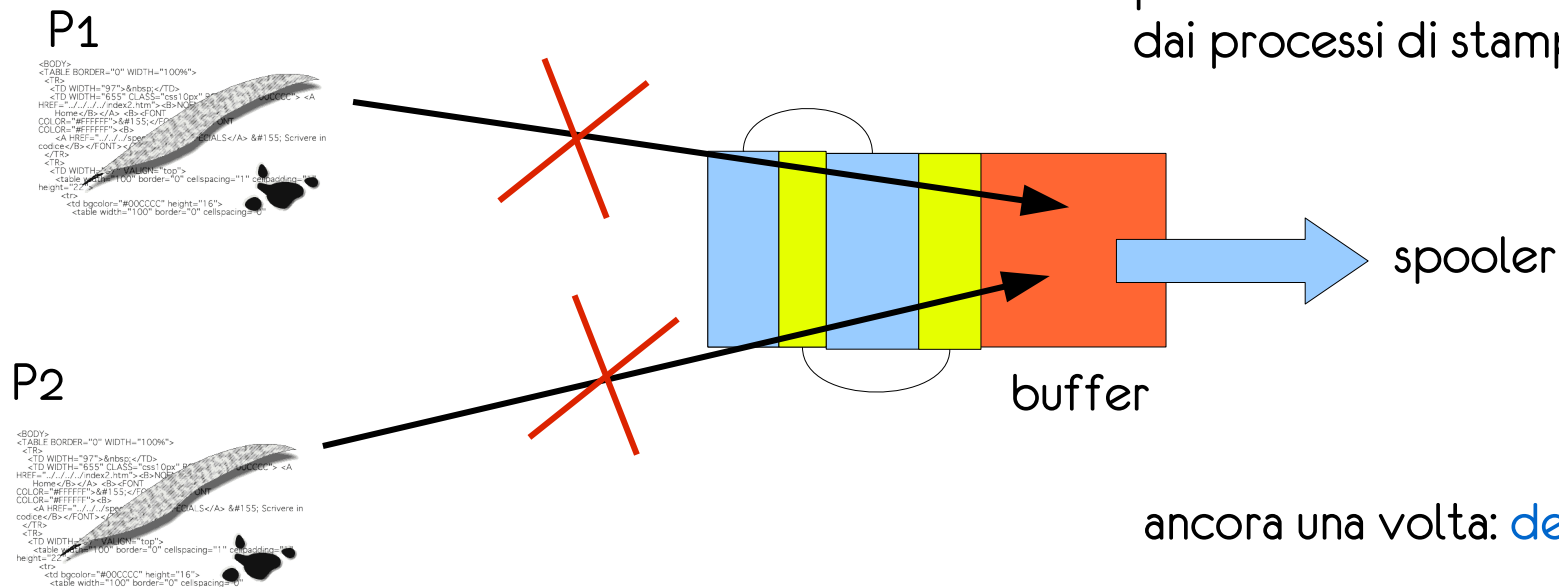
ancora una volta: **deadlock**

Nota

La discontinuità delle aree di memoria contenenti l'output di un processo di stampa non è un problema. Quando studieremo la memoria vedremo che si tratta della norma: i file sono memorizzati in aree discontinue, il SO ha strutture adeguate a mantenere/ricostruire la struttura logica e sequenziale dei file

Scenario

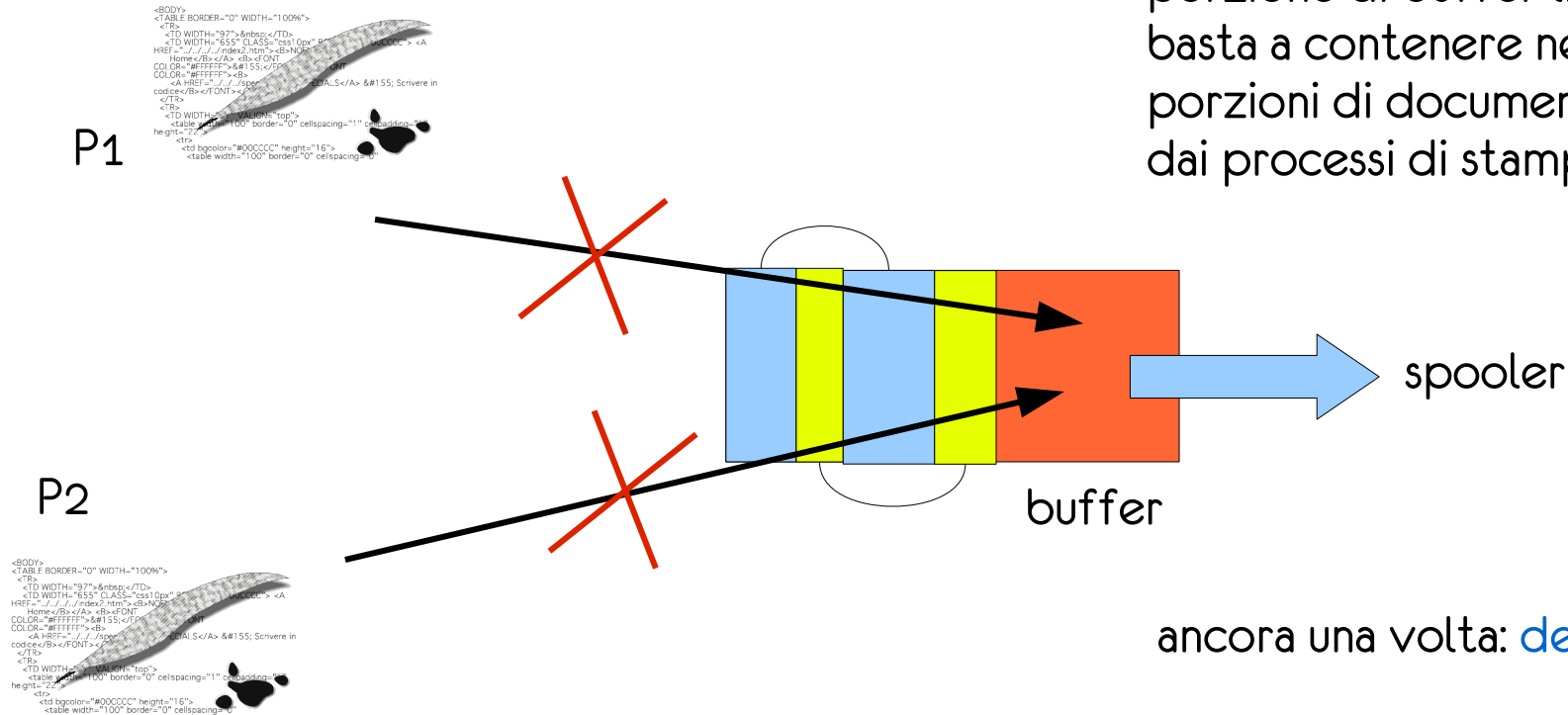
Problema: cosa succede se la porzione di buffer libera non basta a contenere nessuna delle porzioni di documento prodotte dai processi di stampa?



Il deadlock è causato da: (1) attesa circolare fra i processi di stampa, (2) possesso e attesa di porzioni di memoria, (3) mutua esclusione nell'uso di un'area di memoria specifica, (4) no prelazione

Scenario

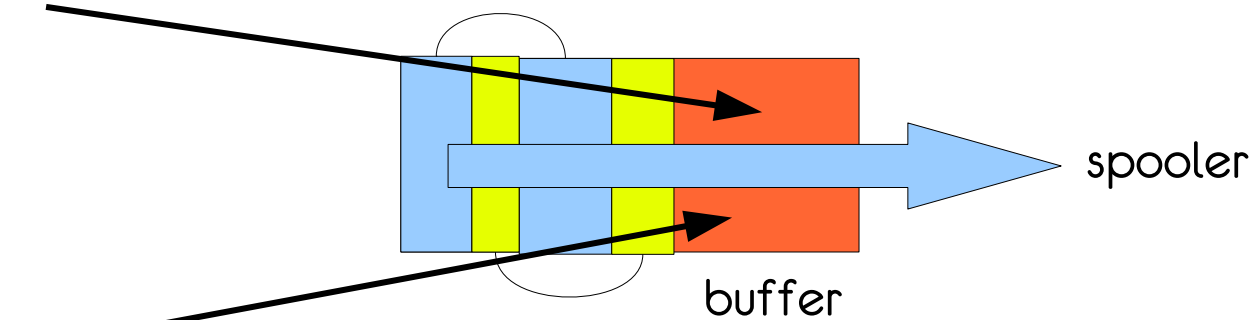
Problema: cosa succede se la porzione di buffer libera non basta a contenere nessuna delle porzioni di documento prodotte dai processi di stampa?



P1

[illegible]

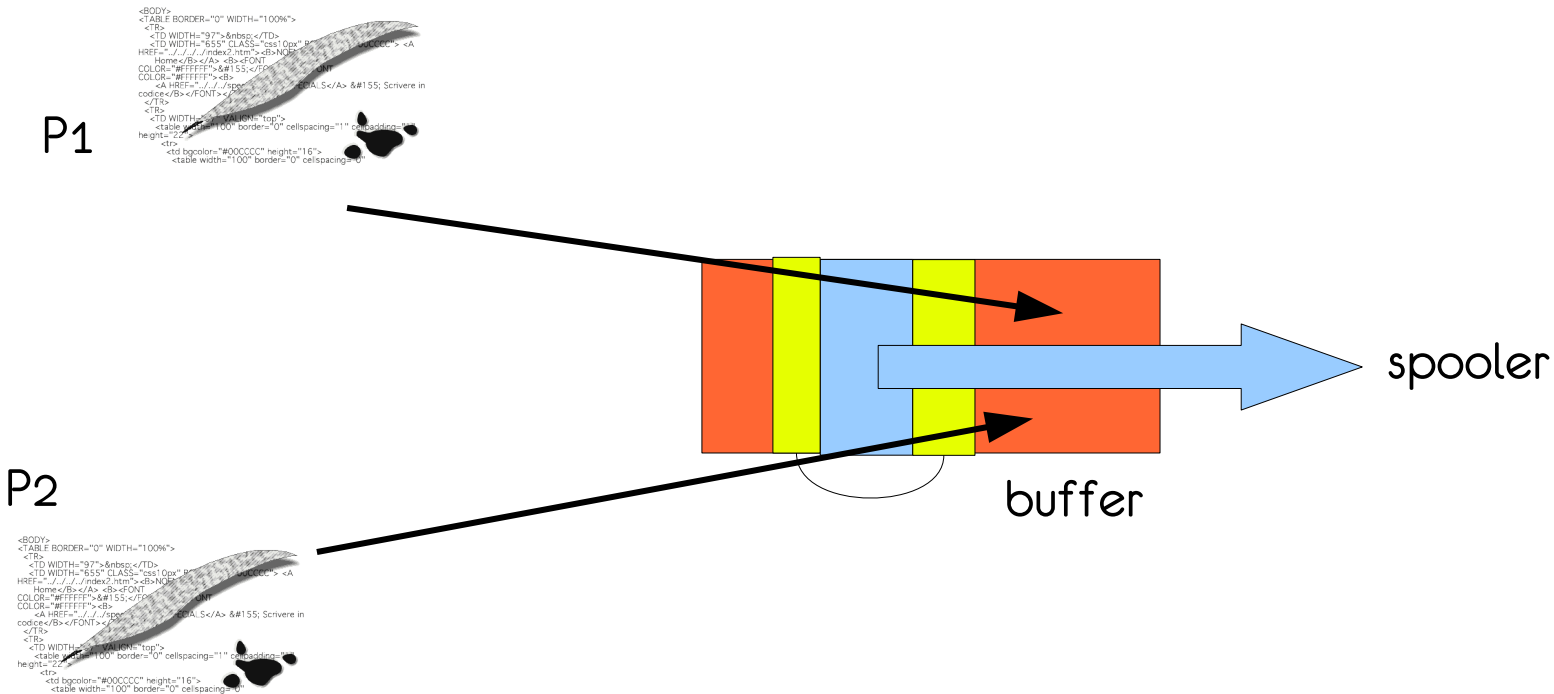
P2

[illegible]

Soluzione: **streaming**

lo spooler sottrae a un processo di stampa la memoria che ha acquisito e usato svuotandola, cioè riversandone il contenuto sulla stampante: **prelazione della risorsa memoria**

Scenario



Soluzione: **streaming**

lo spooler sottrae a un processo di stampa la memoria che ha acquisito e usato svuotandola, cioè riversandone il contenuto sulla stampante

Che fare col deadlock?

- Rilevare il deadlock:



- è una capacità fondamentale se non abbiamo metodi che a priori ne evitano il generarsi

- 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

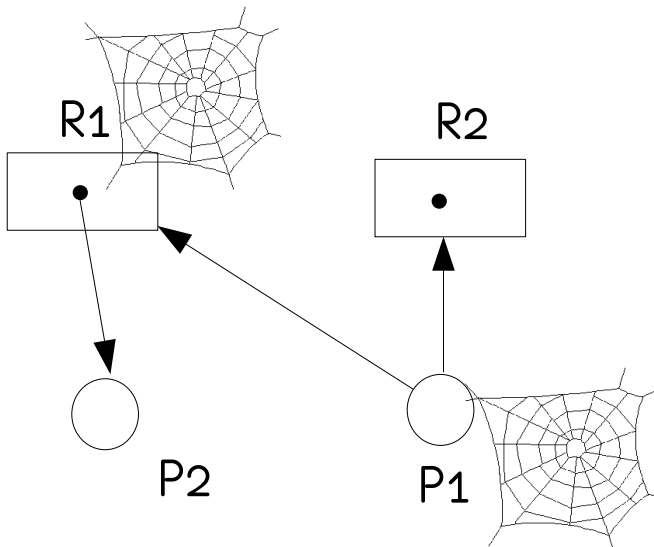


Prevenzione

- **Rendere impossibile una delle 4 condizioni necessarie al deadlock**
 - **Mutua Esclusione**: la richiesta di usare le risorse in ME può essere rilasciata solo per alcuni tipi di risorse, es. file aperti in lettura, altre sono intrinsecamente ME, es. CD writer (due processi non possono scrivere sullo stesso CD contemporaneamente)
 - **Possesso e attesa**:
 - **possibile strategia**: se un processo ha bisogno di più risorse non può accumularle un po' per volta, o le ottiene tutte insieme o non ne prende nessuna. Nota: Occorre evitare starvation.
 - **Consentire la prelazione**:
 - **possibile strategia**: un processo che ha N risorse e ne richiede un'altra o la ottiene subito o (se occorre attendere) rilascia tutte le risorse in suo possesso
 - **Attesa circolare**:
 - **possibile strategia**: imporre un ordinamento delle risorse e dei processi

Prima strategia di Havender

- Protocollo di richiesta delle risorse: **tutte le risorse necessarie ad un processo devono essere richieste insieme**
 - **se sono tutte disponibili**, il sistema le assegna e il processo prosegue
 - **se anche solo una non è disponibile** il processo non ne acquisisce nessuna e si mette in attesa
- **Vantaggio**: previene il deadlock
- **Svantaggio**: spreco di risorse, ad esempio:



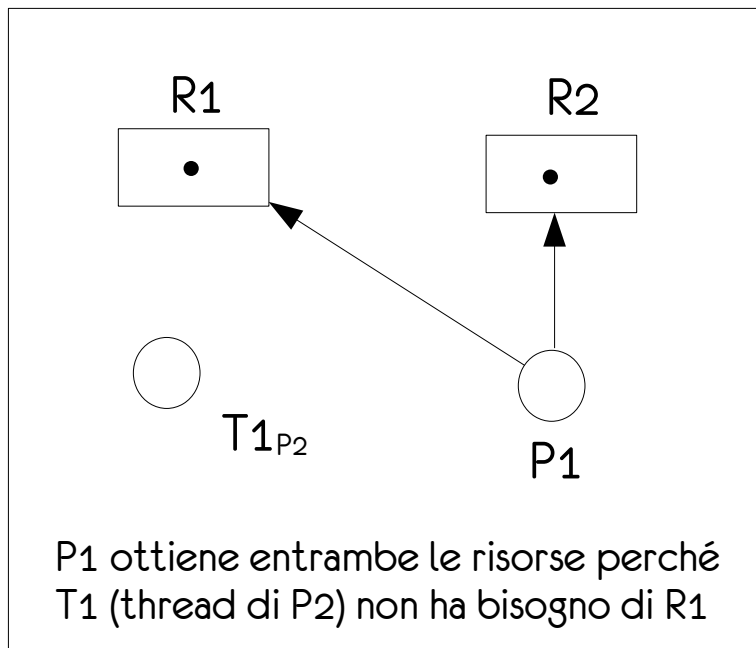
P1 richiede sia R1 che R2 ma l'unica istanza di R1 è assegnata a P2

P2 ha dovuto richiedere R1 ma l'userà solo al termine del proprio lavoro (3 ore dopo!!)

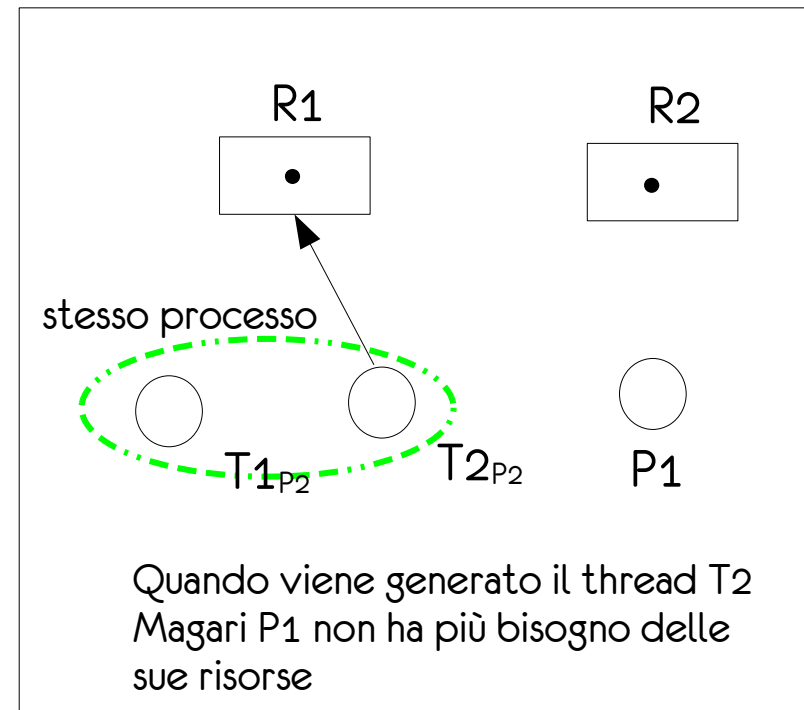
R2 è libera, P1 potrebbe usare sia R1 che R2 prima che a P2 occorra effettivamente R1

Possibile miglioramento

- Questa strategia non funziona per **processi heavyweight** però se all'interno del processo riusciamo a distinguere più **thread** di esecuzione, ciascuno dei quali ha bisogno di un sottoinsieme delle risorse ed è generato solo quando occorre ... la strategia può risultare efficace



istante 1



istante 2

Nota: in generale un processo che richiede molte risorse può essere soggetto a starvation

Seconda strategia di Havender

- **Consentire la prelazione delle risorse**
- Se la prima strategia di Havender non è applicata, un processo potrebbe accumulare risorse via via.
- Supponiamo che a un certo punto il processo effettui una richiesta non esaudibile perché le risorse sono esaurite:
 - il processo non può eseguire il proprio compito ma ...
 - ... se *non rilascia le risorse accumulate* neanche gli altri processi potranno lavorare!!

Seconda strategia di Havender

- **Consentire la prelazione delle risorse**
- Se la prima strategia di Havender non è applicata, un processo potrebbe accumulare risorse via via.
- Supponiamo che a un certo punto il processo effettui una richiesta non esaudibile perché le risorse sono esaurite:
 - il processo non può eseguire il proprio compito ma ...
 - ... se *non rilascia le risorse accumulate* neanche gli altri processi potranno lavorare!!
- **Seconda strategia di Havender:** quando un processo richiede una risorsa che gli viene negata, rilascia tutte le risorse accumulate fino a quel momento
- eventualmente il processo effettua subito dopo una nuova richiesta di tutte le risorse che ha appena perso + quella che non è riuscito ad ottenere

Critica

- È una tecnica **costosa**: perdere delle risorse può significare perdere un lavoro già compiuto in parte (es. se mi viene tolta della memoria perdo i dati eventualmente già inseriti in essa)
- Vale la pena solo se il sistema è tale per cui verrà applicata di rado
- Il suo uso in congiunzione a un **criterio di priorità** che predilige l'assegnazione di risorse a processi che ne richiedono poche, può causare la starvation di quei processi che hanno bisogno di molte risorse
- Inoltre **non tutte le risorse sono preemptible**: per esempio interrompere una stampa non è ragionevole

Attesa circolare

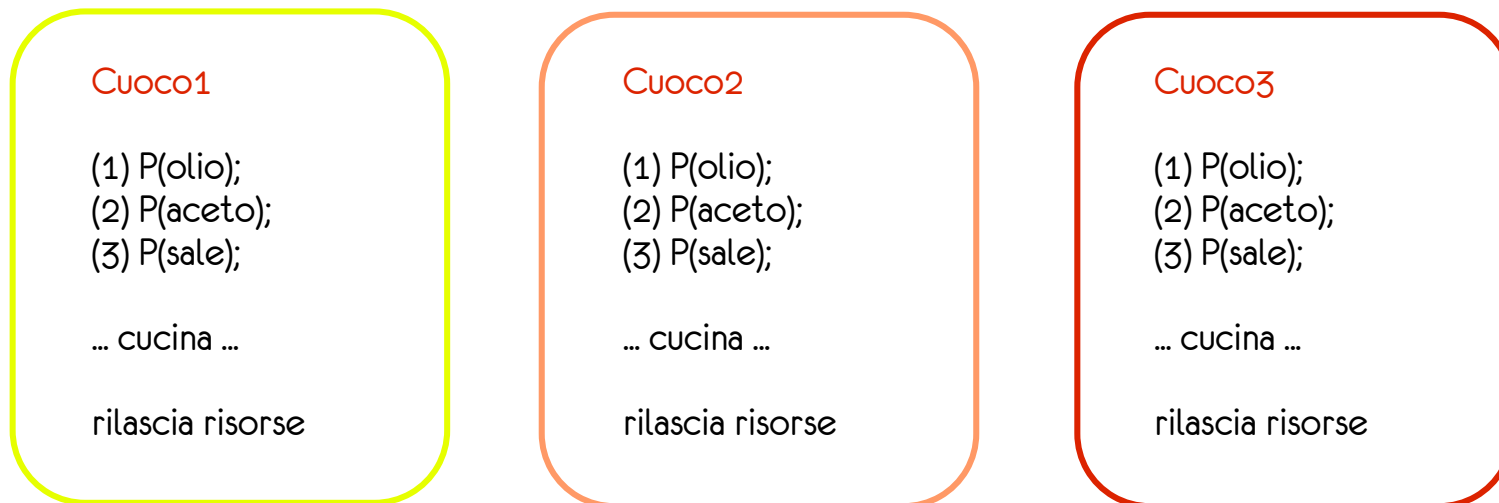
- L'ultima strategia di Havender comporta l'**avoidance dell'attesa circolare**
- Ogni **risorsa** ha assegnato un **numero**, utilizzato per quella risorsa soltanto
- Sulla base di tali numeri le risorse risultano ordinabili in ordine strettamente crescente (**$R1 < R2 < \dots < Rn$**)

Attesa circolare

- L'ultima strategia di Havender comporta l'**avoidance dell'attesa circolare**
- Ogni **risorsa** ha assegnato un **numero**, utilizzato per quella risorsa soltanto
- Sulla base di tali numeri le risorse risultano ordinabili in ordine strettamente crescente (**$R1 < R2 < \dots < Rn$**)
- Un processo che abbia bisogno di M risorse le deve **richiedere in ordine crescente**, per esempio:
 - P ha bisogno di R4, R7 ed R2 allora richiede nell'ordine R2, quindi R4 e infine R7
- Non si può avere deadlock perché l'ordinamento delle richieste impedisce l'attesa circolare
- È stata usata in alcuni sistemi operativi ma **non è molto flessibile: chi scrive programmi per il sistema deve essere consapevole dell'ordinamento imposto alle risorse**

Esempio

- Proviamo ad applicare la terza strategia di Havender ai tre cuochi
- Numeriamo le risorse: $\text{olio} \leftarrow 1$, $\text{aceto} \leftarrow 2$, $\text{sale} \leftarrow 3$
- Tutti i processi che usano queste risorse le devono **richiedere rispettando l'ordinamento**: i tre cuochi diventeranno uguali, nella loro prima parte del programma



- A questo punto i tre cuochi competono per la risorsa olio, che verrà assegnata ad uno solo di loro, che potrà richiedere la risorsa aceto senza entrare in competizione con gli altri, ancora in attesa dell'olio. In breve uno dei processi otterrà tutte le risorse necessarie e non si avrà deadlock

Deadlock avoidance

- Non sempre è possibile inibire a priori una delle condizioni necessarie affinché si abbia il deadlock (applicando le strategie di Havender)
- **questo non significa che non si possa evitare il deadlock**
- i metodi che consentono di fare ciò richiedono alcune informazioni, per esempio che i processi dichiarino quante risorse di un certo tipo hanno bisogno
- L'algoritmo di **deadlock avoidance** esamina lo stato di allocazione delle risorse e garantisce che in futuro non si formeranno attese circolari

Deadlock avoidance

- In certi contesti non è possibile inibire a priori una delle condizioni necessarie affinché si abbia il deadlock (applicando le strategie di Havender)
- **Questo non significa che non si possa evitare il deadlock**
- I metodi che consentono di fare ciò **richiedono alcune informazioni**, per esempio che i processi dichiarino quante risorse di un certo tipo hanno bisogno

Deadlock avoidance

- Occorre introdurre due nozioni nuove:
 - <1> **stato (del sistema) sicuro**: si dice che il sistema è in uno stato sicuro (o safe) se il SO può garantire che ciascun processo completerà la propria esecuzione in un tempo finito

Stato di allocazione delle risorse

- Lo **stato di allocazione delle risorse** cattura il numero di risorse libere, di risorse allocate e se disponibile il numero di risorse ancora richiedibili
- Visualizzabile tramite una **tabella**
- Es. se ho **10 istanze di una classe di risorse R**, tre processi **P1, P2 e P3**, inoltre **P1 ha allocate 3 risorse e ne desidera ancora 2**, P2 ne ha allocate 4 e ne desidera 1 e P3 ne ha allocata 1 e ne desidera ancora 4:

allocati ← A R → ancora da richiedere

	A	R
P1	3	2
P2	4	1
P3	1	4

processi ←

è una fotografia dell'allocazione delle risorse in un certo istante

NB: un processo non può procedere se non ha tutte le risorse che gli servono

free 2 → risorse libere

Stato di allocazione delle risorse

	A	R
P1	3	2
P2	4	1
P3	1	4

free 2

	A	R
P1	3	2
P2	5	0
P3	1	4

free 1

	A	R
P1	5	0
P2	4	1
P3	1	4

free 0

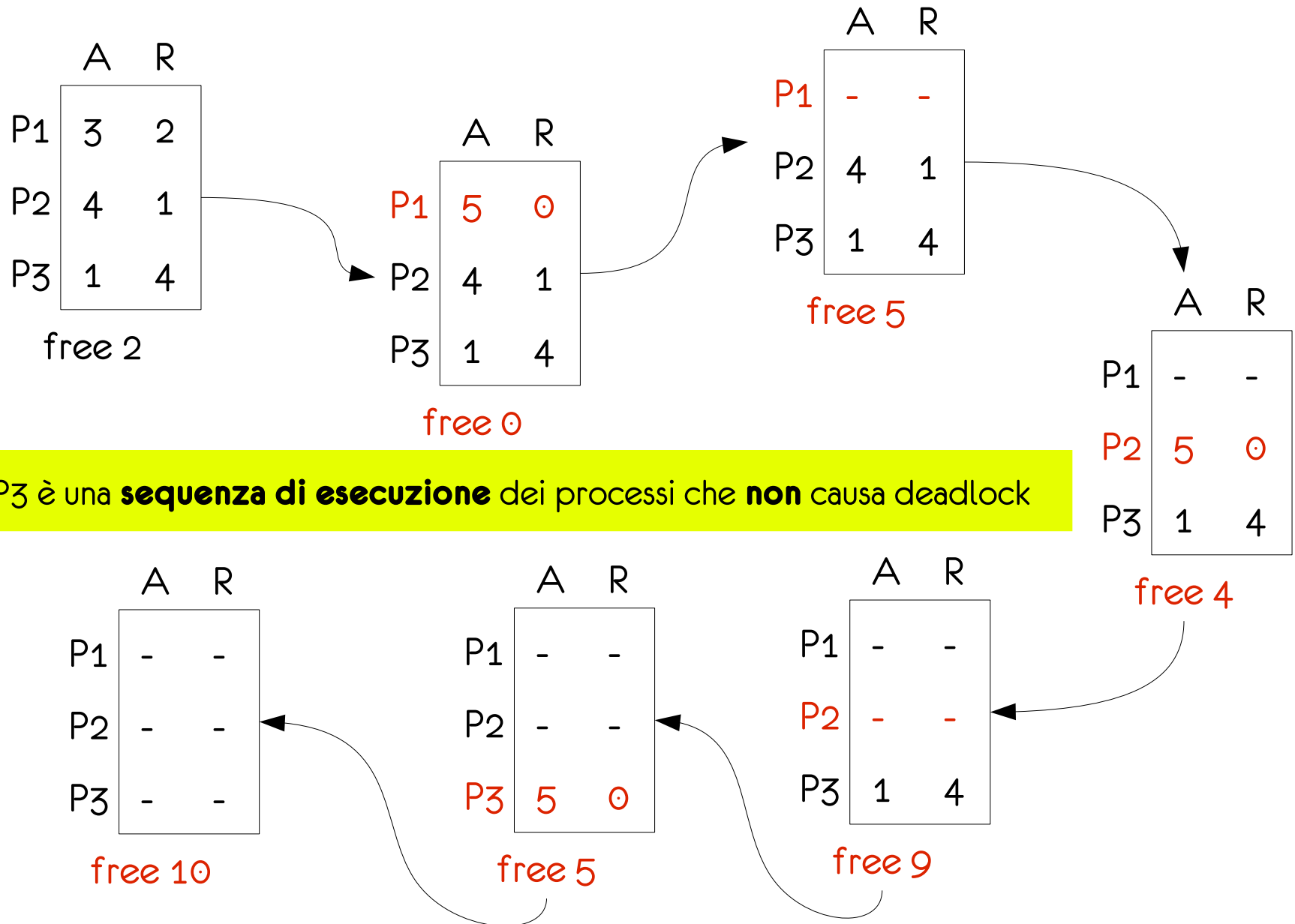
lo stato si evolve a seconda dell'esecuzione successiva. A seconda del processo che verrà eseguito si possono avere diverse evoluz.

NB: se un processo richiede n risorse, il SO può decidere di assegnargliene $m < n$

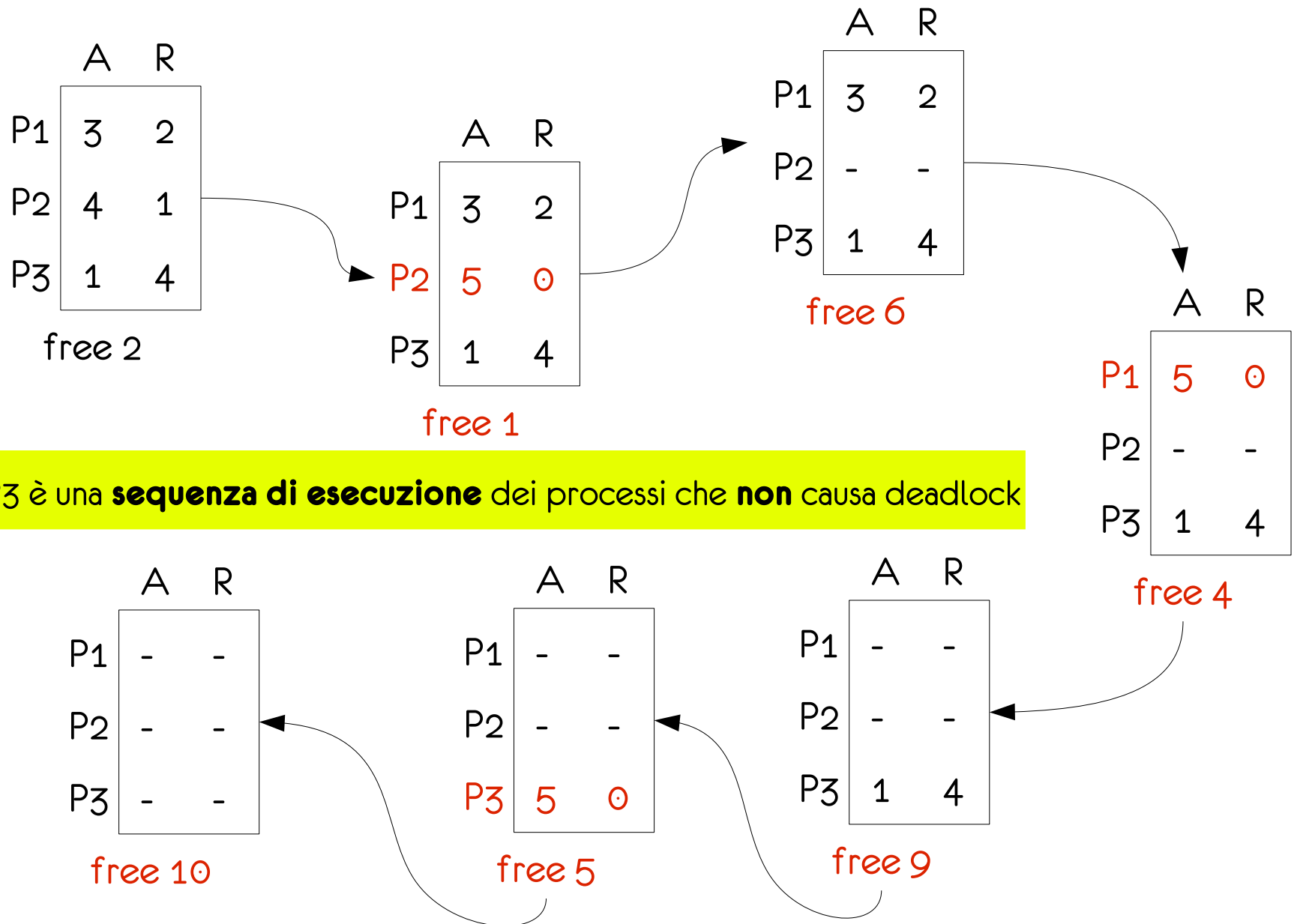
P2 ha ricevuto la risorsa mancante e può procedere

P1 ha ricevuto le risorse mancanti e può procedere

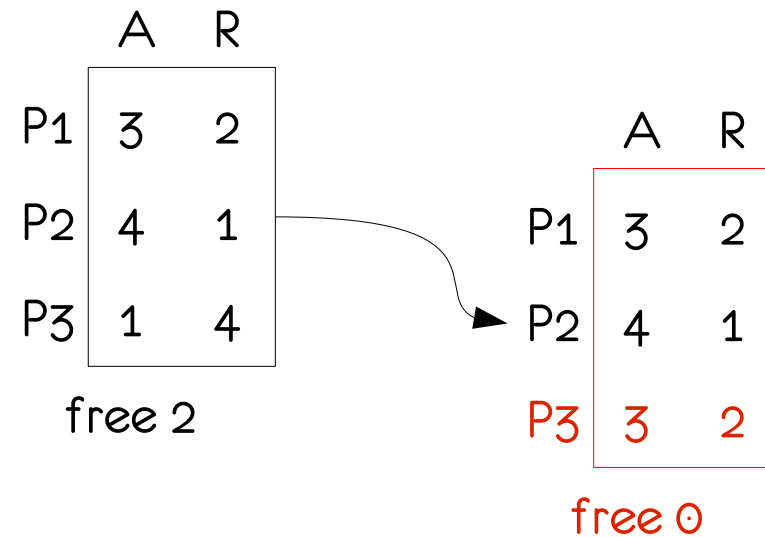
Sequenza di esecuzione



Altra sequenza di esecuzione

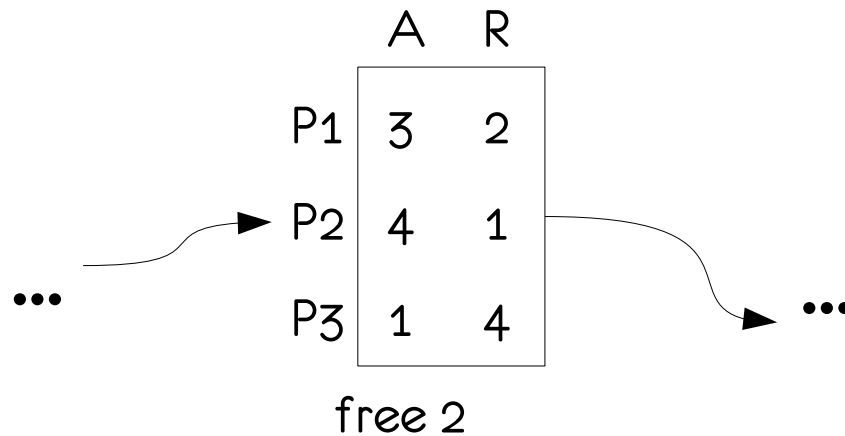


Altra sequenza di esecuzione



La scelta di soddisfare in parte la richiesta di P3 comporta invece il deadlock infatti nessun processo ha abbastanza risorse per proseguire e non ci sono più risorse libere

Stato iniziale?



Siamo partiti dallo stato indicato senza precisare che si può trattare di un qualsiasi stato di esecuzione, non necessariamente quello iniziale!!
Le risorse sono in parte già allocate quindi deve essere avvenuta una porzione di esecuzione

Deadlock avoidance

- Occorre introdurre due nozioni nuove:
 - <1> **stato (del sistema) sicuro**: si dice che il sistema è in uno stato sicuro (o safe) se il SO può garantire che ciascun processo completerà la propria esecuzione in un tempo finito
 - <2> **sequenza sicura**: una **sequenza** $\langle P^1, \dots, P^n \rangle$ di processi (parzialmente eseguiti) è detta **sequenza sicura** se le richieste che ogni P^i deve ancora fare sono soddisfacibili usando le **risorse inizialmente libere** più le **risorse usate (e liberate) dai processi P^j aventi $j < i$** (cioè dai processi che lo precedono)

Deadlock avoidance

- Finché il sistema di processi che condividono risorse rimane in uno stato sicuro il SO può evitare il verificarsi del deadlock

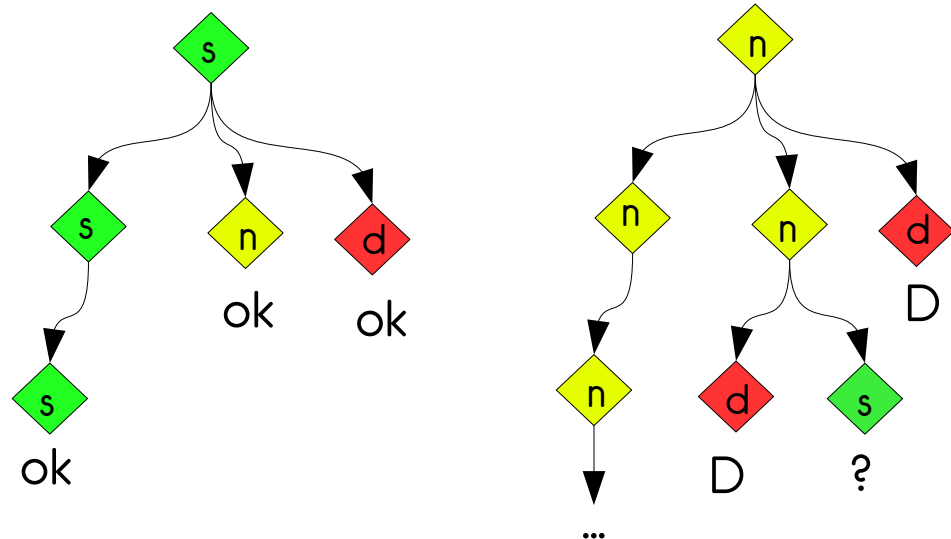
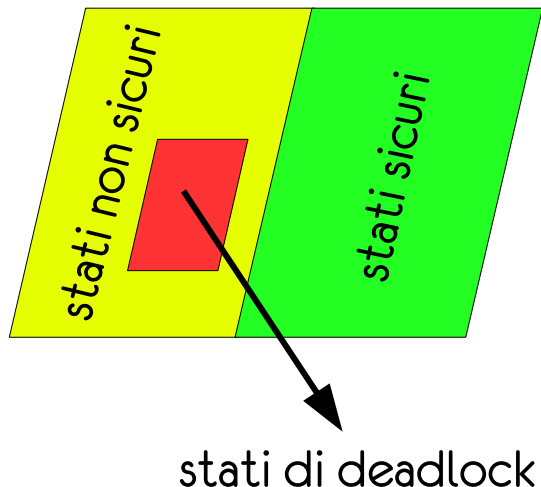
Perché funziona?

- Una sequenza sicura non presenta deadlock perché per ogni processo P^i sono veri i seguenti casi:
 - P^i ha tutte le risorse che gli servono
 - oppure a P^i basta aspettare la terminazione di qualche processo precedente per poter procedere
- può capitare che si generi una catena di attese che non ha mai termine?
 - l'unico caso è l'attesa circolare. L'attesa può risultare circolare?
 - no! Infatti alla peggio si torna indietro lungo la sequenza fino a P^1 :

per definizione di sequenza sicura tutte le richieste che il processo P^1 effettuerà da qui alla sua fine devono essere o disponibili oppure in possesso di un processo P^j con $j < 1$, che però non esiste

Stati sicuri e sequenze sicure

- Uno stato è sicuro se da esso si dirama almeno una sequenza sicura, quindi **se esiste** almeno un ordinamento dei processi che è una sequenza sicura
- Uno stato non sicuro non necessariamente è uno stato di deadlock ma può portare al deadlock
- Uno stato non sicuro può evolvere in uno stato sicuro? Se tale evoluzione esistesse per definizione lo stato sarebbe sicuro



Evoluzione degli stati

STATO
SICURO

	A	R
P1	5	5
P2	2	2
P3	2	7

free 3

...

	A	R
P1	5	5
P2	3	1
P3	2	7

free 2

...

	A	R
P1	5	5
P2	3	1
P3	4	5

free 0

DEADLOCK

	A	R
P1	5	5
P2	4	0
P3	2	7

STATO
SICURO

free 1

...

	A	R
P1	5	5
P2	-	-
P3	3	6

free 4

DEADLOCK

	A	R
P1	5	5
P2	4	0
P3	3	6

free 0

STATO
NON SICURO

Lo stato iniziale è sicuro ma a seconda delle scelte di allocazione delle risorse può trasformarsi in uno stato non sicuro e anche in un deadlock

Evoluzione degli stati

STATO
SICURO

	A	R
P1	5	5
P2	2	2
P3	2	7

free 3

...

	A	R
P1	5	5
P2	3	1
P3	2	7

free 2

...

	A	R
P1	5	5
P2	3	1
P3	4	5

free 0

DEADLOCK

	A	R
P1	5	5
P2	4	0
P3	2	7

STATO
SICURO

free 1

...

	A	R
P1	5	5
P2	-	-
P3	3	6

free 4

DEADLOCK

	A	R
P1	5	5
P2	4	0
P3	3	6

free 0

STATO
NON SICURO

Lo stato iniziale è sicuro ma a seconda delle scelte di allocazione delle risorse può trasformarsi in uno stato non sicuro e anche in un deadlock

Altro esempio: i 3 cuochi

Cuoco1

(1) P(olio);
(2) P(aceto);
(3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

(1) P(sale);
(2) P(olio);
(3) P(aceto);

... cucina ...

rilascia risorse

Cuoco3

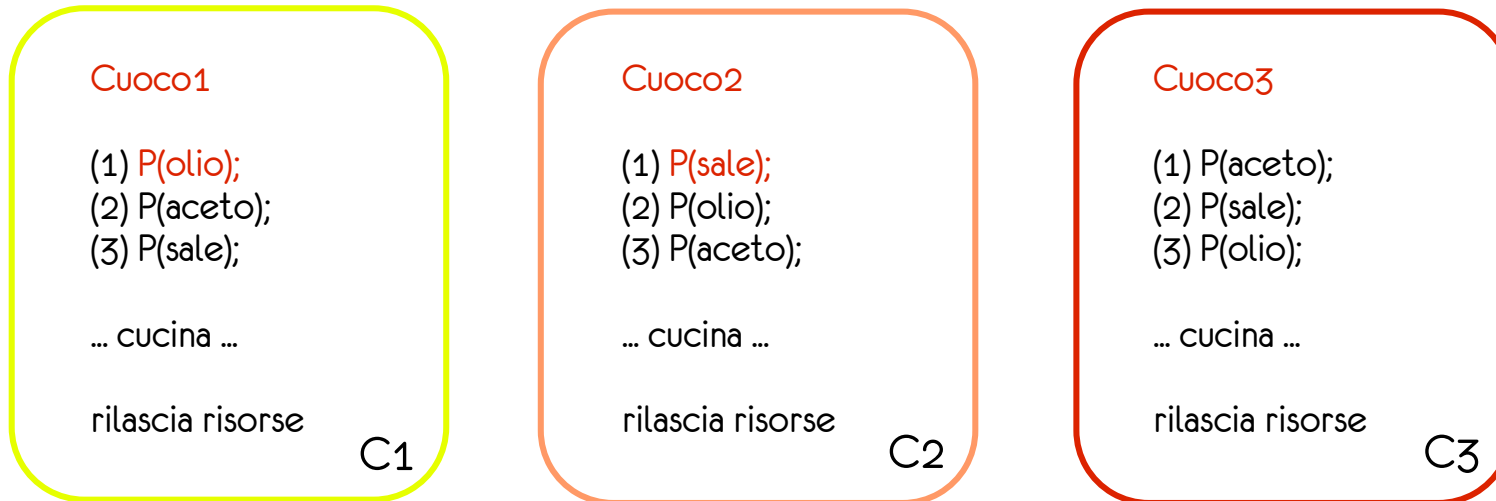
(1) P(aceto);
(2) P(sale);
(3) P(olio);

... cucina ...

rilascia risorse

- Supponiamo che Cuoco1 abbia l'olio e Cuoco2 il sale, il sistema è in uno stato sicuro?
- In questo momento non c'è deadlock ma ...
- ... esiste un ordinamento che è una sequenza sicura?

i 3 cuochi



- Tutti i possibili ordinamenti:
 - C1, C2, C3: no C1 dipende da C2 che lo segue
 - C1, C3, C2: idem
 - C2, C1, C3: no C2 dipende da C1 che lo segue
 - C2, C3, C1: idem
 - C3, C1, C2: no C1 dipende da C2
 - C3, C2, C1: no C2 dipende da C1

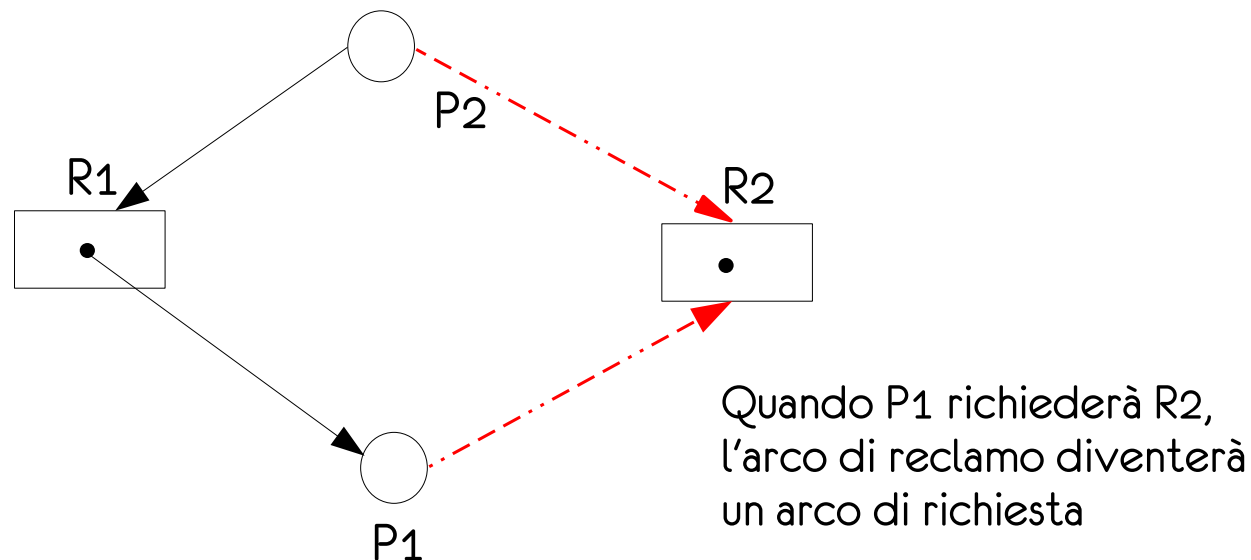
Lo stato non
è sicuro

Deadlock avoidance

- Per evitare il deadlock il SO cerca di mantenere l'esecuzione in uno stato sicuro
- **E se una scelta sbagliata portasse a uno stato non sicuro?**
- in questo caso non è più possibile riportare il sistema in uno stato sicuro e molto facilmente si genererà un deadlock

Algoritmo di deadlock avoidance

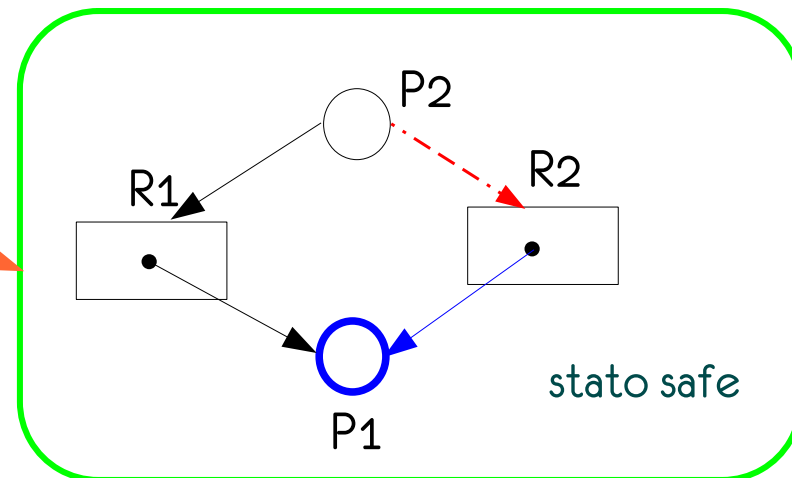
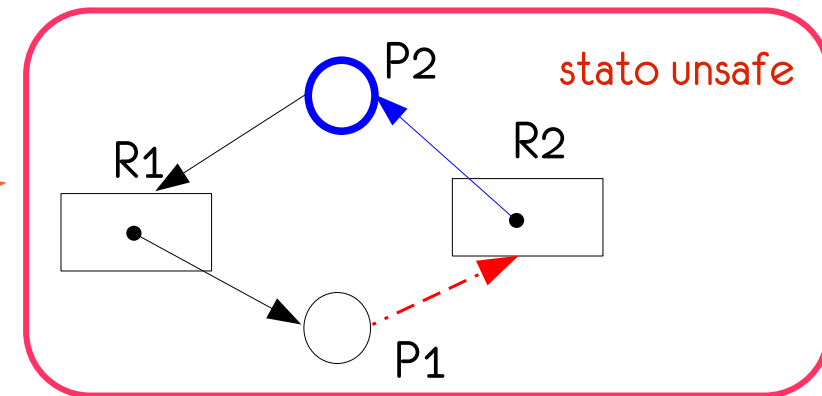
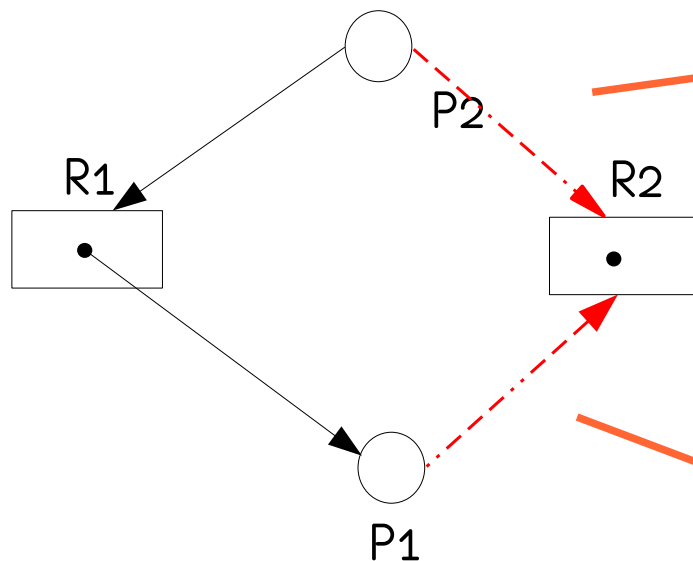
- Questo metodo funziona solo se ogni classe di risorsa ha **una istanza**
- È possibile prevenire il deadlock utilizzando una variante del grafo di assegnazione delle risorse ottenuto introducendo un terzo tipo di arco:
 - **arco di reclamo** (claim edge): $P_i \rightarrow R_j$ indica che P_i richiederà R_j *in futuro*; è rappresentato da una linea tratteggiata



Algoritmo di deadlock avoidance

- All'inizio tutti i processi inseriscono nel grafo di assegnazione un claim edge per ciascuna risorsa di cui avranno bisogno
- È possibile trasformare un arco di reclamo in un arco di richiesta **SSE** non si genera un ciclo (costituito da qualsiasi tipo di archi)

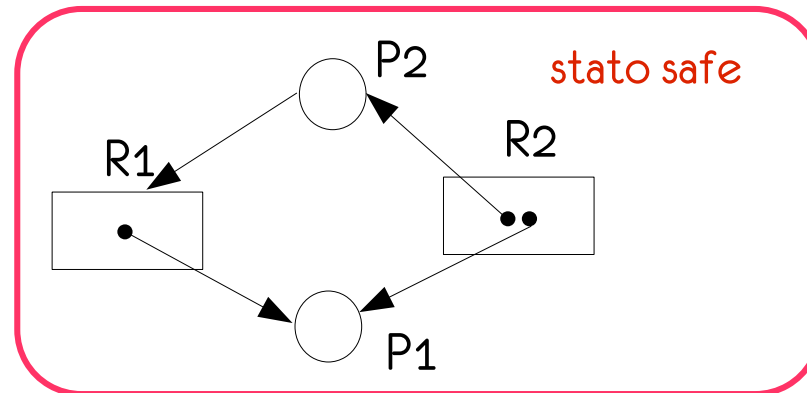
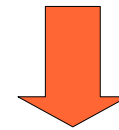
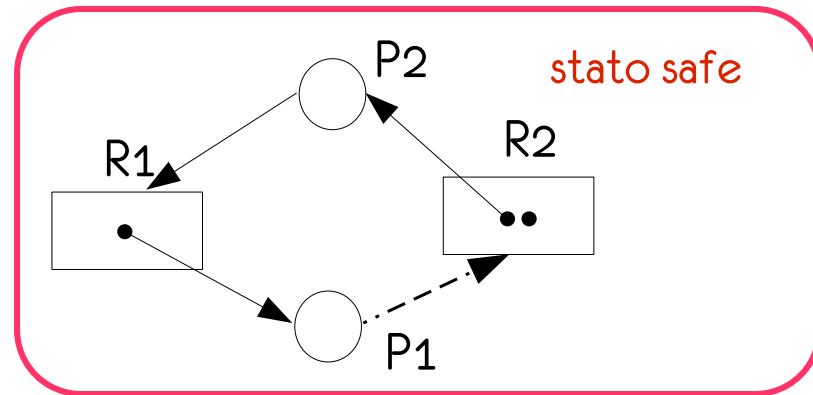
• Es.



Algoritmo di deadlock avoidance

NB: se io avessi due istanze di R2 lo stato sarebbe safe!!
La condizione non è più sufficiente

NOTA: quando un processo rilascia una risorsa l'arco di assegnazione ritorna ad essere un arco di reclamo



Avoidance: algoritmo del banchiere

- Algoritmo più generale, si applica anche quando i processi richiedono $n > 1$ risorse di un certo tipo
- **Metafora**: i processi sono visti come clienti di una banca a cui possono richiedere un prestito fino a un certo massimo
- **Informazione richiesta**: ogni nuovo processo deve dichiarare all'inizio il numero massimo di risorse (dei vari tipi) di cui avrà bisogno
- **M** = numero delle classi di risorsa gestite
- **N** = numero dei processi
- Complessità: **$O(N^2M)$**



Algoritmo del banchiere: variabili

- **disponibili[M]**: indica la disponibilità per ogni classe di risorsa
- **massimo[N][M]**: per ciascun processo indica il numero massimo di risorse di ciascun tipo che saranno richieste
- **assegnate[N][M]**: indica quante risorse di ciascuna classe sono assegnate a ogni processo
- **necessarie[N][M]**: indica quante risorse di ciascun tipo ancora mancano ai vari processi ($necessarie = massimo - assegnate$)

Algoritmo del banchiere

- L'algoritmo soddisfa una richiesta di un processo **SSE**
l'assegnazione delle risorse richieste porta ad uno stato sicuro
- È diviso in due algoritmi:
 - 1) un algoritmo per verificare che uno stato è sicuro
 - 2) un algoritmo di gestione di una richiesta (che utilizza il precedente)

- **Convenzione notazionale:**

Dati due vettori di uguale lunghezza X e Y si indica con $X < Y$ il fatto che per ogni indice i $X[i] < Y[i]$, si indica con $X \leq Y$ il fatto che per ogni indice i $X[i] \leq Y[i]$ e si indica con $Z = X + Y$ il fatto che per ogni indice i $Z[i] = X[i] + Y[i]$

Algoritmo di verifica della sicurezza

1. Siano **Lavoro** e **Fine** due array di lunghezza **M** ed **N**
2. **Lavoro = Disponibili**
3. **Fine[i] = falso**, per ogni $i \in [1, N]$
4. Cerca $i \in [1, N] \mid \text{Fine}[i] == \text{false} \wedge \text{Necessarie}[i] \leq \text{Lavoro}$
5. se l'hai trovato:
 1. **Lavoro = Lavoro + Assegnate[i]**
 2. **Fine[i] = true**
 3. **goto 4**
6. altrimenti **goto 7**
7. se $\forall i \in [1, N], \text{Fine}[i] == \text{true}$ lo stato è sicuro

Esempio con $M == 1$

STATO
SICURO?

	A	R
P1	5	5
P2	2	2
P3	2	7

disponibili = 3
necessarie = {5, 2, 7}
assegnate = {5, 2, 2}

fine = {false, false, false}
lavoro = disponibili, cioè è uguale a 3

free 3

c'è $i \in [1, N] \mid$
fine[i]==false \wedge
necessarie[i] ≤ lavoro?
Sì i == 2!!

lavoro = lavoro + assegnate[2]
fine[2] = true

è come se fossi passata
virtualmente nello stato

	A	R
P1	5	5
P2	-	-
P3	2	7

free 5

disponibili = 3
necessarie = {5, 2, 7}
assegnate = {5, 2, 2}

fine = {false, true, false}
lavoro = 5

Esempio con $M == 1$

	A	R
P1	5	5
P2	-	-
P3	2	7

disponibili = 3
necessarie = {5, 2, 7}
assegnate = {5, 2, 2}

fine = {false, true, false}
lavoro = 5

free 5

c'è $i \in [1, N] \mid$
fine[i]==false \wedge
necessarie[i] ≤ lavoro?
Sì i == 1!!

lavoro = lavoro + assegnate[1]
fine[1] = true

è come se fossi passata
virtualmente nello stato

	A	R
P1	-	-
P2	-	-
P3	2	7

free 10

disponibili = 3
necessarie = {5, 2, 7}
assegnate = {5, 2, 2}

fine = {true, true, false}
lavoro = 10

Esempio con $M == 1$

	A	R
P1	-	-
P2	-	-
P3	2	7

disponibili = 3
necessarie = {5, 2, 7}
assegnate = {5, 2, 2}

fine = {true, true, false}
lavoro = 10

LO STATO VALUTATO
È SICURO

free 10

c'è $i \in [1, N] \mid$
fine[i] == false \wedge
necessarie[i] \leq lavoro?
Sì $i == 3!!$

lavoro = lavoro + assegnate[3]
fine[3] = true

è come se fossi passata
virtualmente nello stato

	A	R
P1	-	-
P2	-	-
P3	-	-

free 12

disponibili = 3
necessarie = {5, 2, 7}
assegnate = {5, 2, 2}

fine = {true, true, true}
lavoro = 12

Algoritmo di gestione delle richieste

1. Consideriamo un processo j , sia **Richieste** un vettore di M elementi, **Richieste[i]** è il num. di risorse di classe i richieste da j in un certo istante
2. Se **Richieste > Necessarie[j] ERRORE!** Il processo viola le sue stesse dichiarazioni iniziali di necessità
3. Se invece **Richieste > Disponibili** aspetta
4. Altrimenti **simula l'esecuzione** della richiesta:
 1. **Disponibili = Disponibili - Richieste**
 2. **Assegnate[j] = Assegnate[j] + Richieste**
 3. **Necessarie[j] = Necessarie[j] - Richieste**
5. **Verifica se lo stato raggiunto è sicuro:**
 1. **Se sicuro:** si effettua l'assegnazione
 2. **Se non è sicuro:** si ripristinano i valori precedenti e si sospende il processo

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:

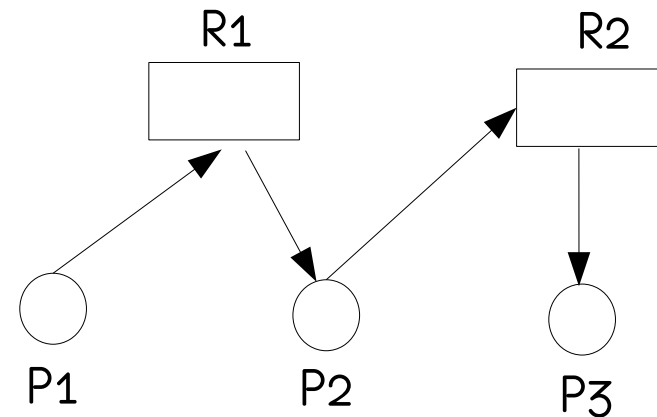
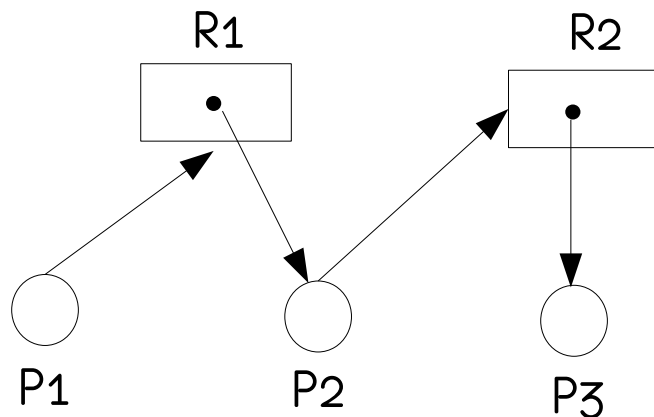
- occorre definire opportuni protocolli di assegnazione delle risorse

- 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

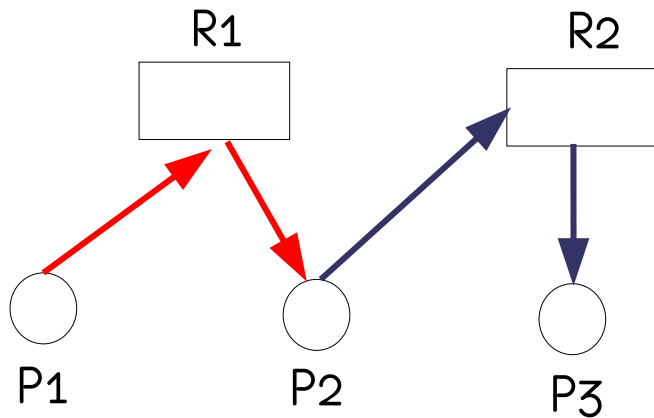
Istanza singola di risorsa

- 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

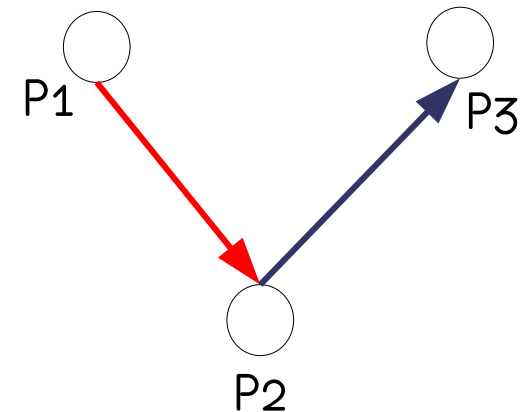


Istanza singola di risorsa

- Un grafo d'attesa è ancora più semplice infatti ha un solo tipo di vertici: i processi
- Un arco $P_i \rightarrow P_j$ indica che P_i è in attesa di una risorsa assegnata a P_j
- <passo 2> Un arco $P_i \rightarrow P_j$ corrisponde a una coppia di archi $P_i \rightarrow R_s$ e $R_s \rightarrow P_j$ nel grafo di allocazione:

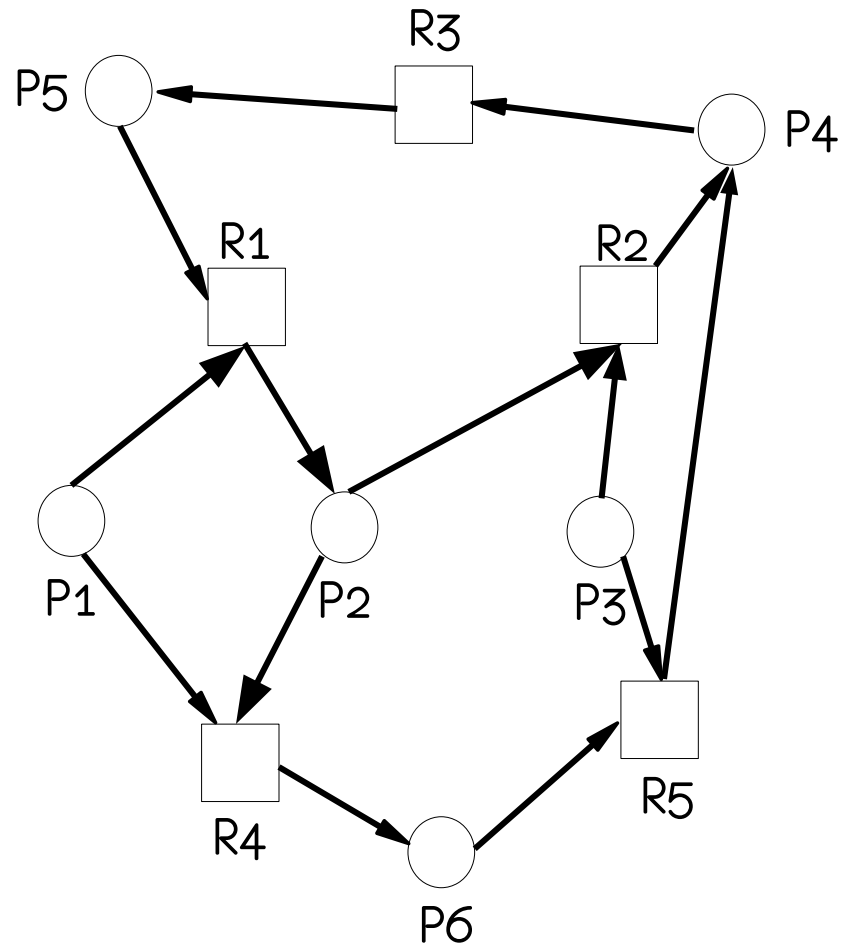


P1 aspetta la risorsa assegnata a P2
P2 aspetta la risorsa assegnata a P3

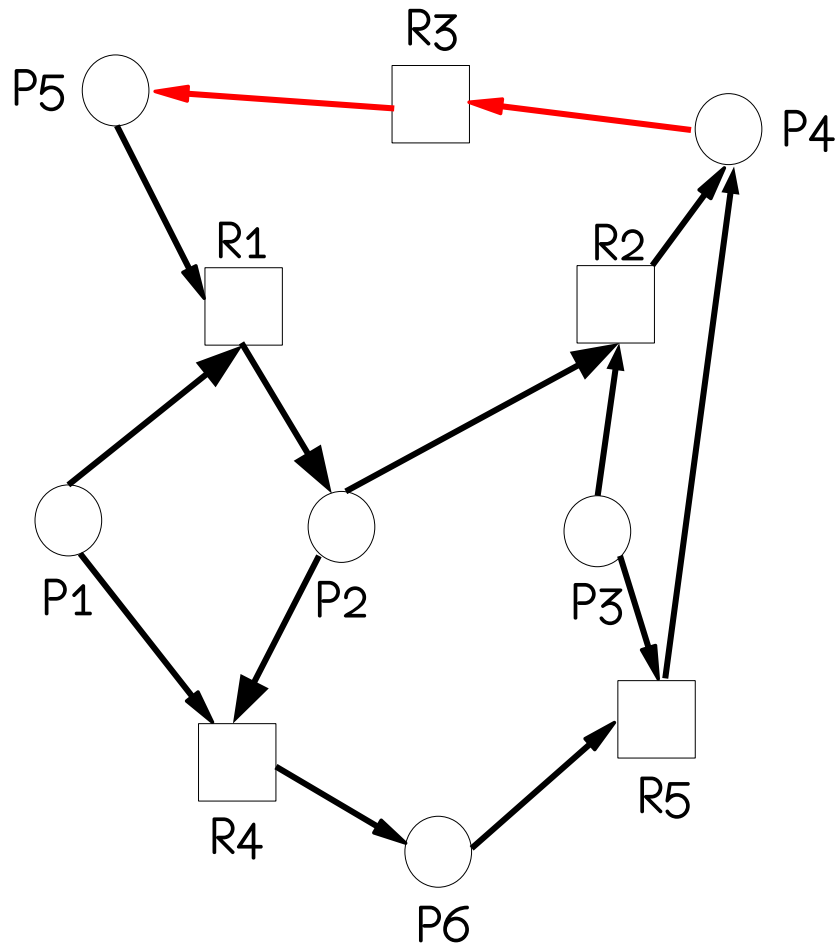


P1 aspetta P2
P2 aspetta P3

esempio 1/5



esempio 2/5



$P1 \rightarrow R1 \rightarrow P2$
 $P1 \rightarrow R4 \rightarrow P6$

$P2 \rightarrow R4 \rightarrow P6$
 $P2 \rightarrow R2 \rightarrow P4$

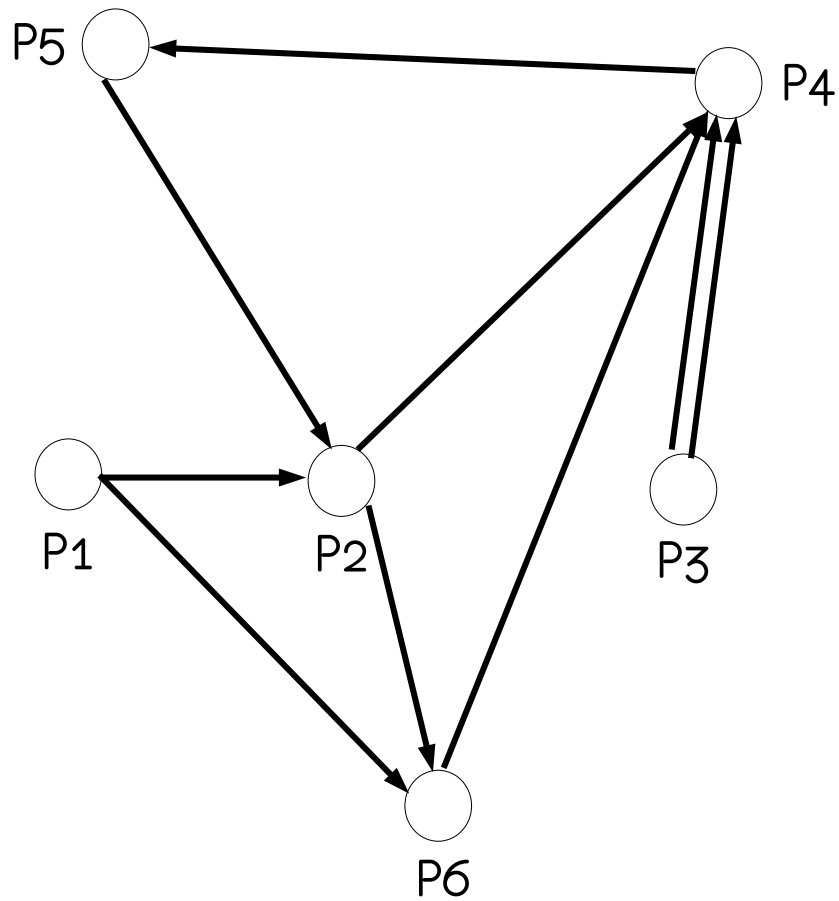
$P3 \rightarrow R2 \rightarrow P4$
 $P3 \rightarrow R5 \rightarrow P4$

$P4 \rightarrow R3 \rightarrow P5$

$P5 \rightarrow R1 \rightarrow P2$

$P6 \rightarrow R5 \rightarrow P4$

esempio 3/5



$P1 \rightarrow R1 \rightarrow P2$

$P1 \rightarrow R4 \rightarrow P6$

$P2 \rightarrow R4 \rightarrow P6$

$P2 \rightarrow R2 \rightarrow P4$

$P3 \rightarrow R2 \rightarrow P4$

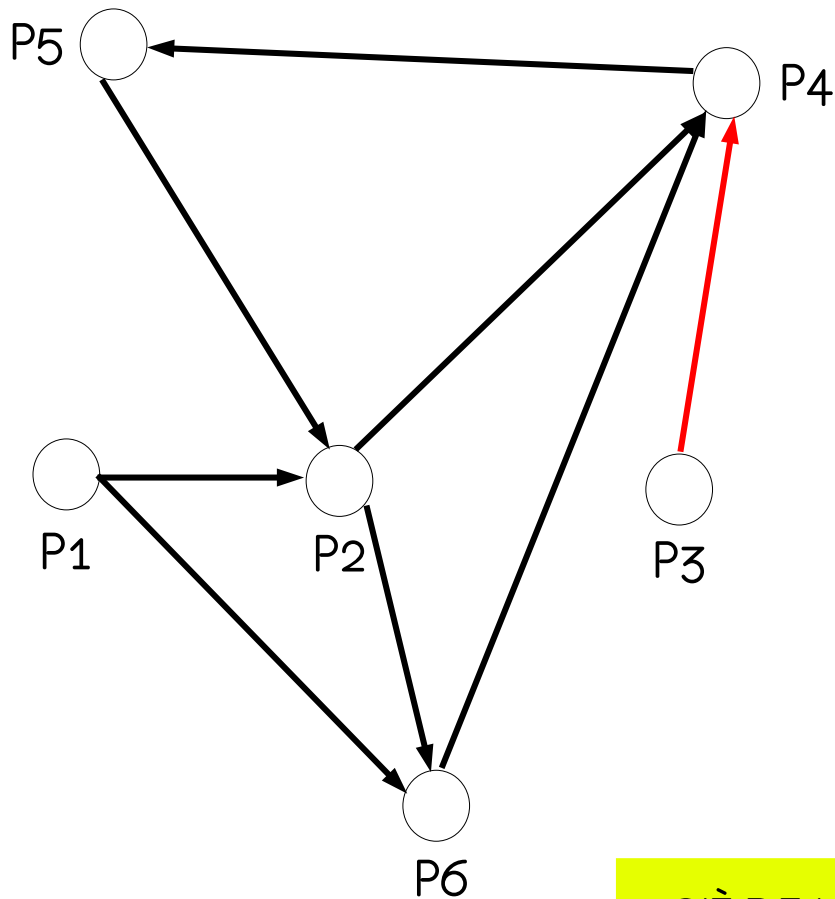
$P3 \rightarrow R5 \rightarrow P4$

$P4 \rightarrow R3 \rightarrow P5$

$P5 \rightarrow R1 \rightarrow P2$

$P6 \rightarrow R5 \rightarrow P4$

esempio 4/5



$P1 \rightarrow R1 \rightarrow P2$
 $P1 \rightarrow R4 \rightarrow P6$

$P2 \rightarrow R4 \rightarrow P6$
 $P2 \rightarrow R2 \rightarrow P4$

$P3 \rightarrow R2 \rightarrow P4$
 $P3 \rightarrow R5 \rightarrow P4$

nella visualizzazione li
fondiamo in un arco solo

$P4 \rightarrow R3 \rightarrow P5$

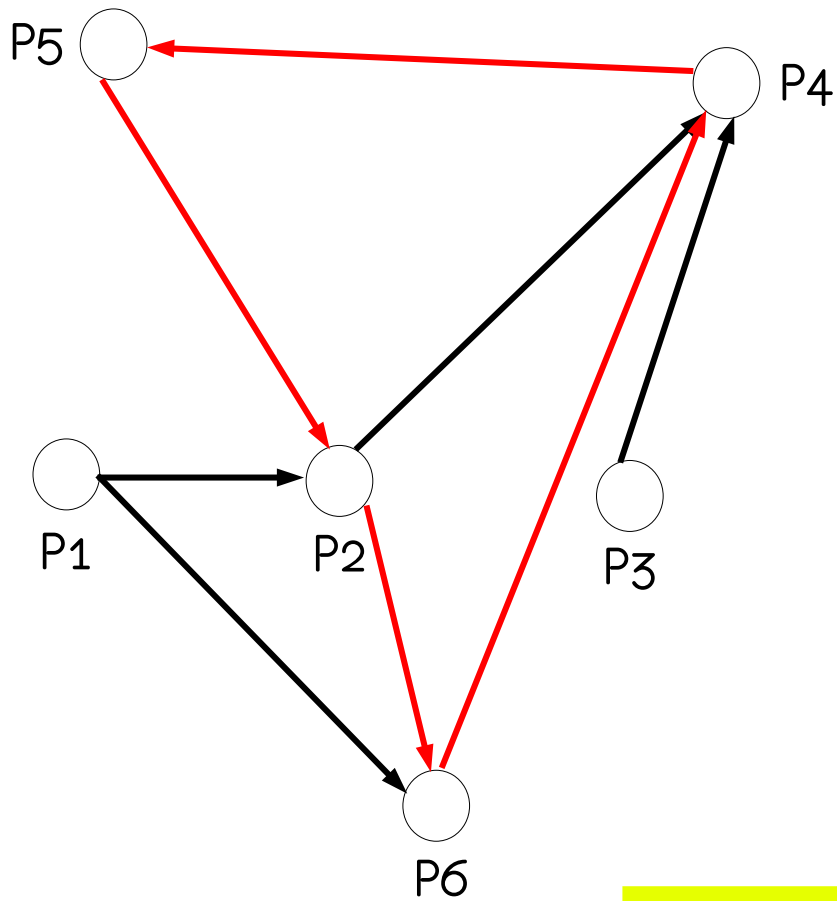
$P5 \rightarrow R1 \rightarrow P2$

$P6 \rightarrow R5 \rightarrow P4$

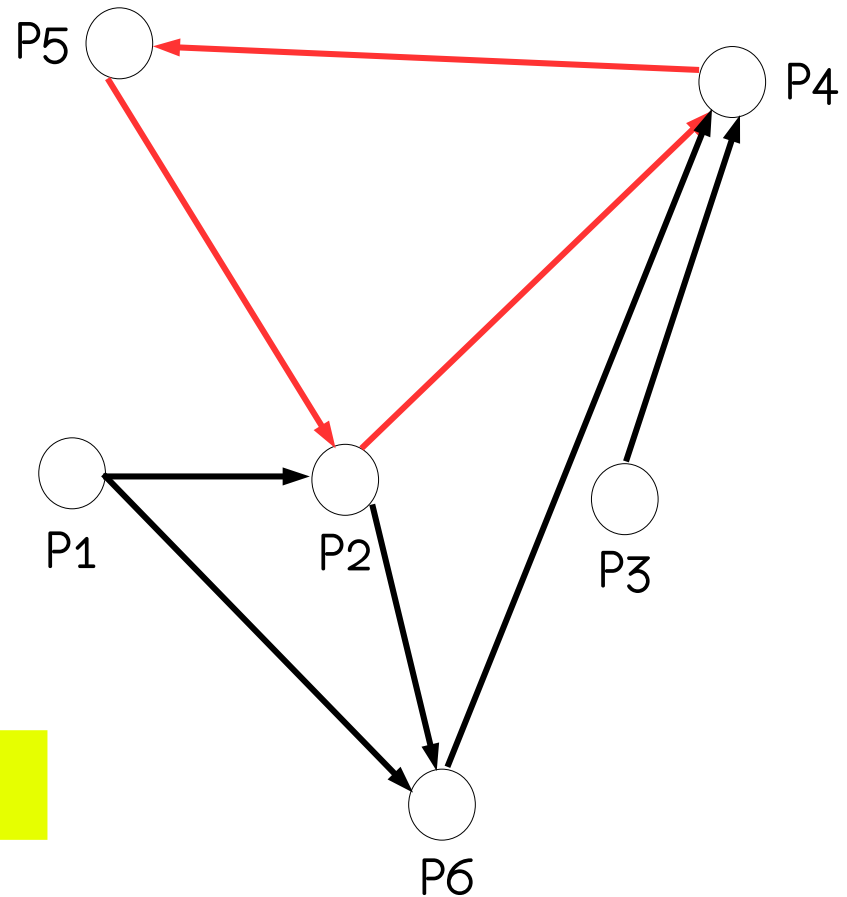
C'È DEADLOCK?
BISOGNA VERIFICARE SE CI SONO CICLI

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 P_i
- **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}

Assegnate[3][4] ==

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

Richieste[3][4] ==

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

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

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)

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  $\exists$  un indice i | Fine[i] == false  $\wedge$  Richieste[i]  $\leq$  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

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

while ($\exists i \mid \text{Fine}[i] == \text{false} \wedge \text{Richieste}[i] \leq \text{Lavoro}$)

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 ($\exists i \mid \text{Fine}[i] == \text{false} \wedge \text{Richieste}[i] \leq \text{Lavoro}$)

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 ($\exists i \mid \text{Fine}[i] == \text{false} \wedge \text{Richieste}[i] \leq \text{Lavoro}$)
LA CONDIZIONE È FALSA

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] ==

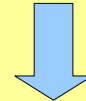
1	0	0	0
2	0	1	0
0	1	0	3

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

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

while ($\exists i \mid \text{Fine}[i] == \text{false} \wedge \text{Richieste}[i] \leq \text{Lavoro}$)

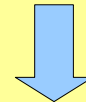
1. ...



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

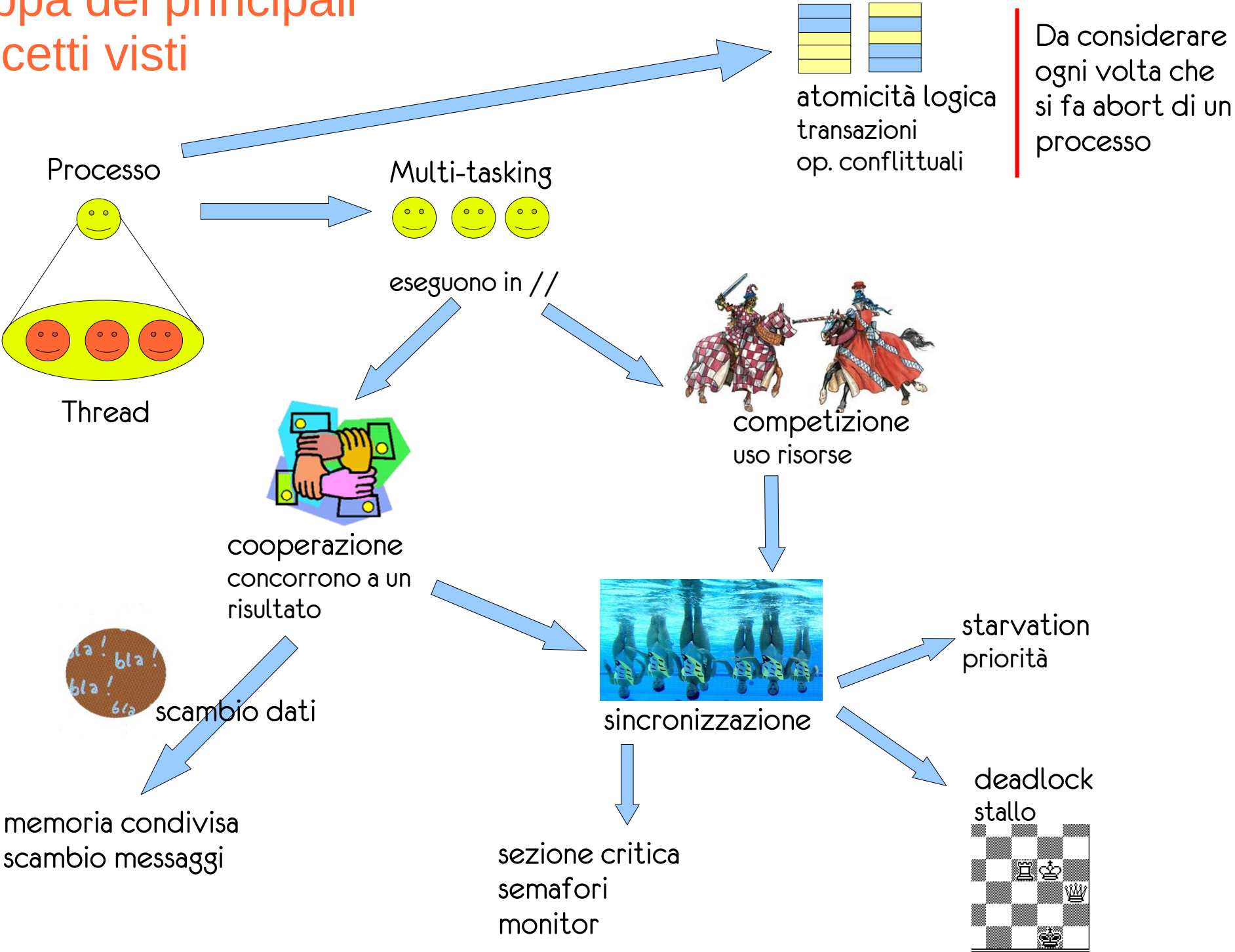
- Idea: sottrarre successivamente risorse ad alcuni processi per assegnarle ad altri
- Anche in questo caso occorre identificare una vittima e la scelta è effettuata su criteri economici, la prelazione di risorse deve essere poco costosa
- Che fare dei processi a cui sono state sottratte risorse? Come 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

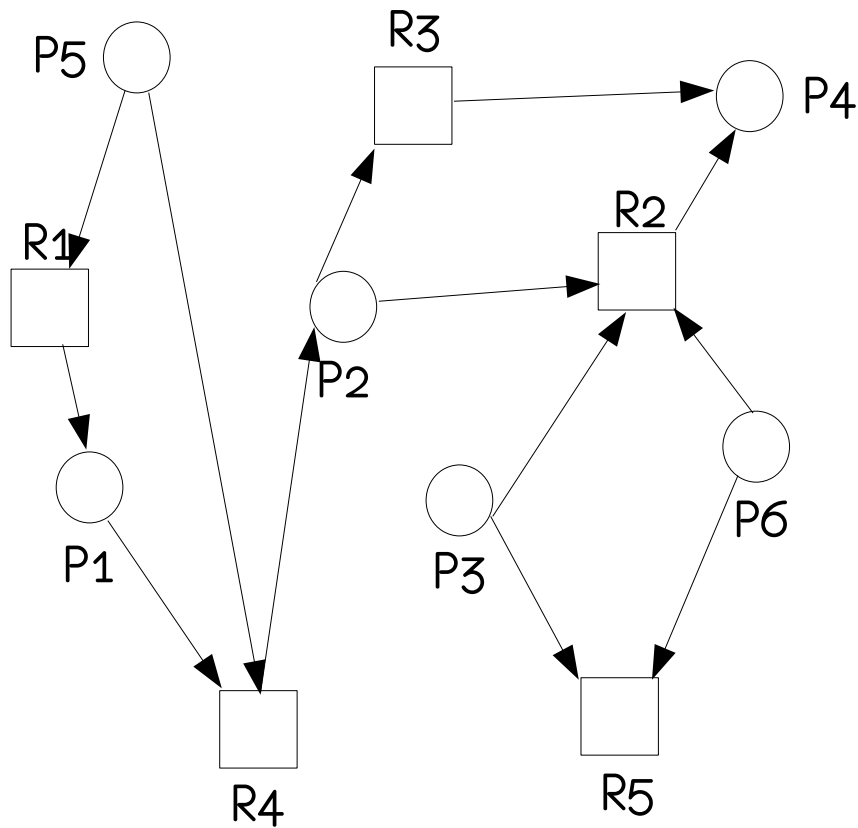


Esercizi

- 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

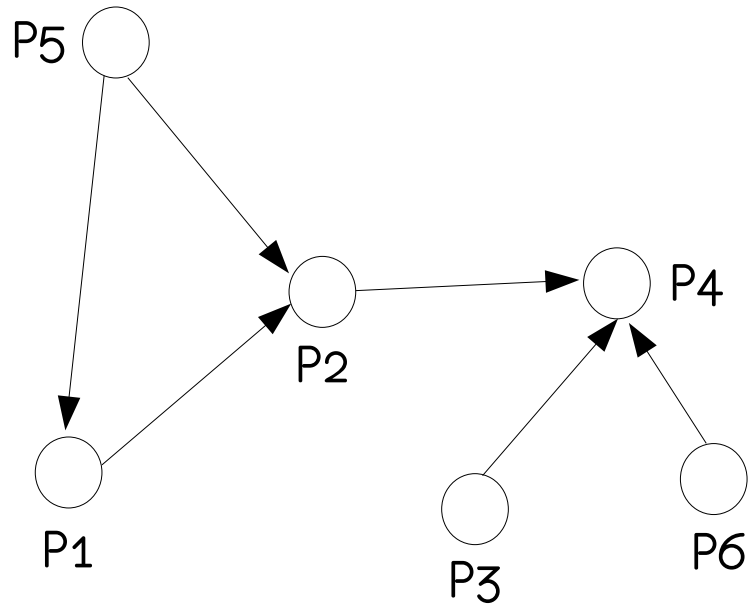
Si consideri il seguente grafo di assegnazione delle risorse, trasformarlo in un grafo di attesa e verificare se vi è deadlock o meno motivando la risposta



soluzione

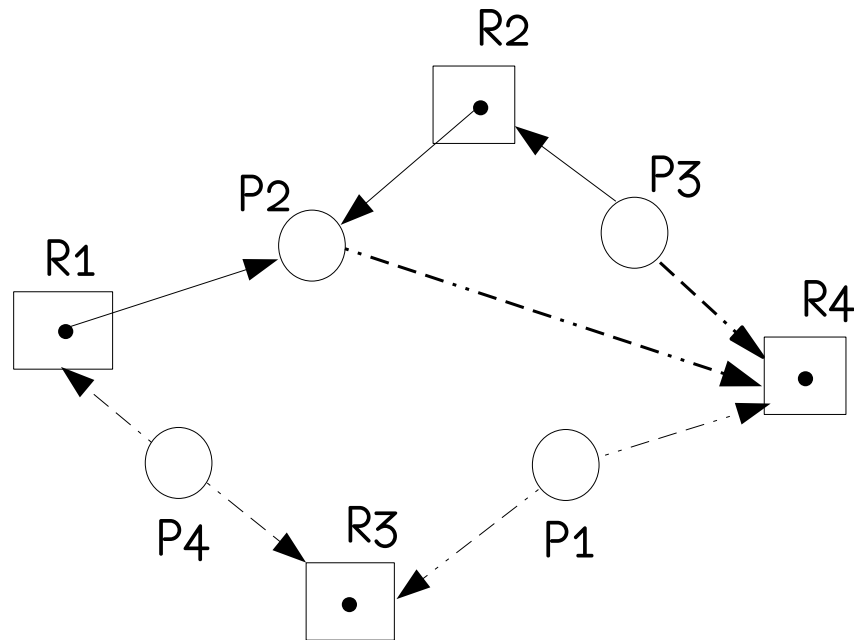
Soluzione:

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

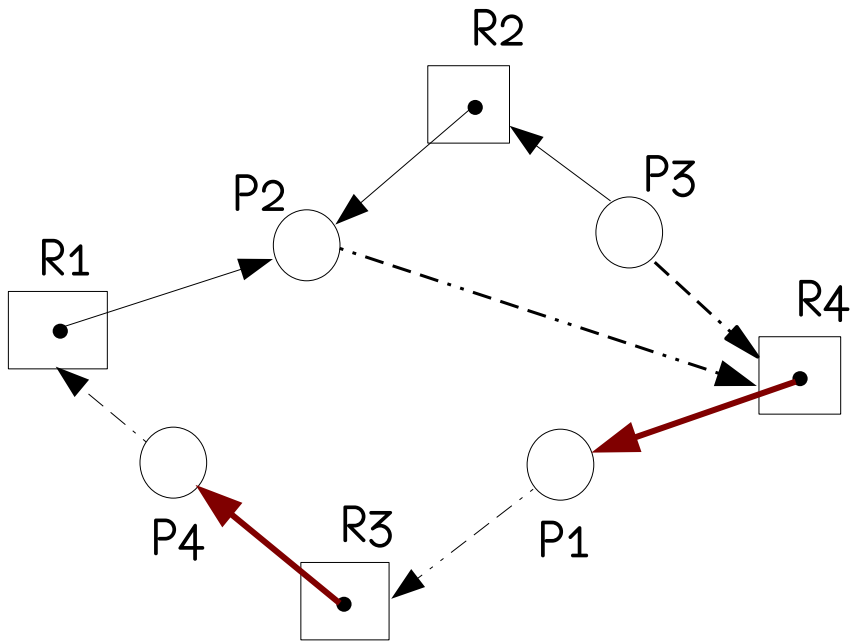


esercizio

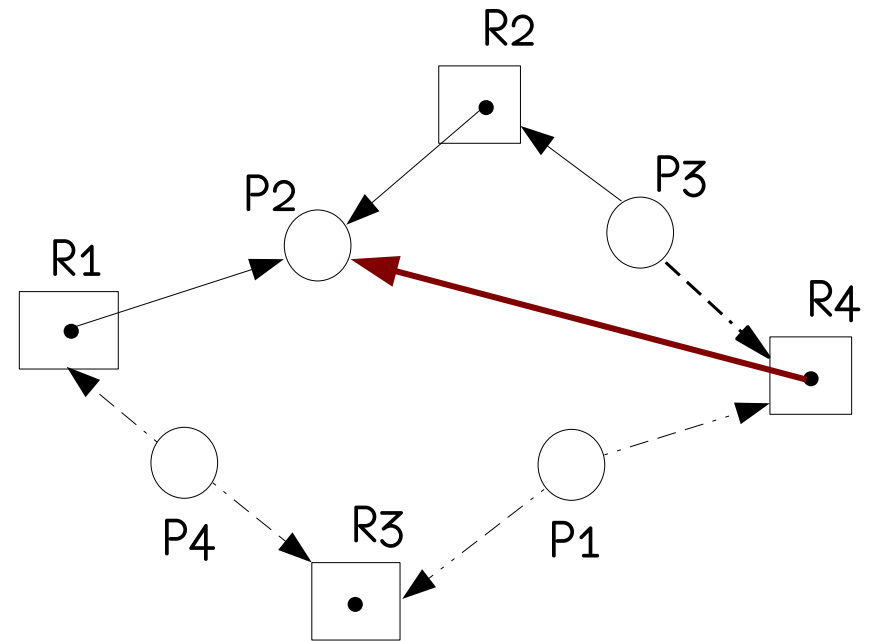
Dato il seguente grafo di assegnazione delle risorse con archi di reclamo individuare un'assegnazione che porta a uno stato non sicuro e un'assegnazione sicura



soluzione



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



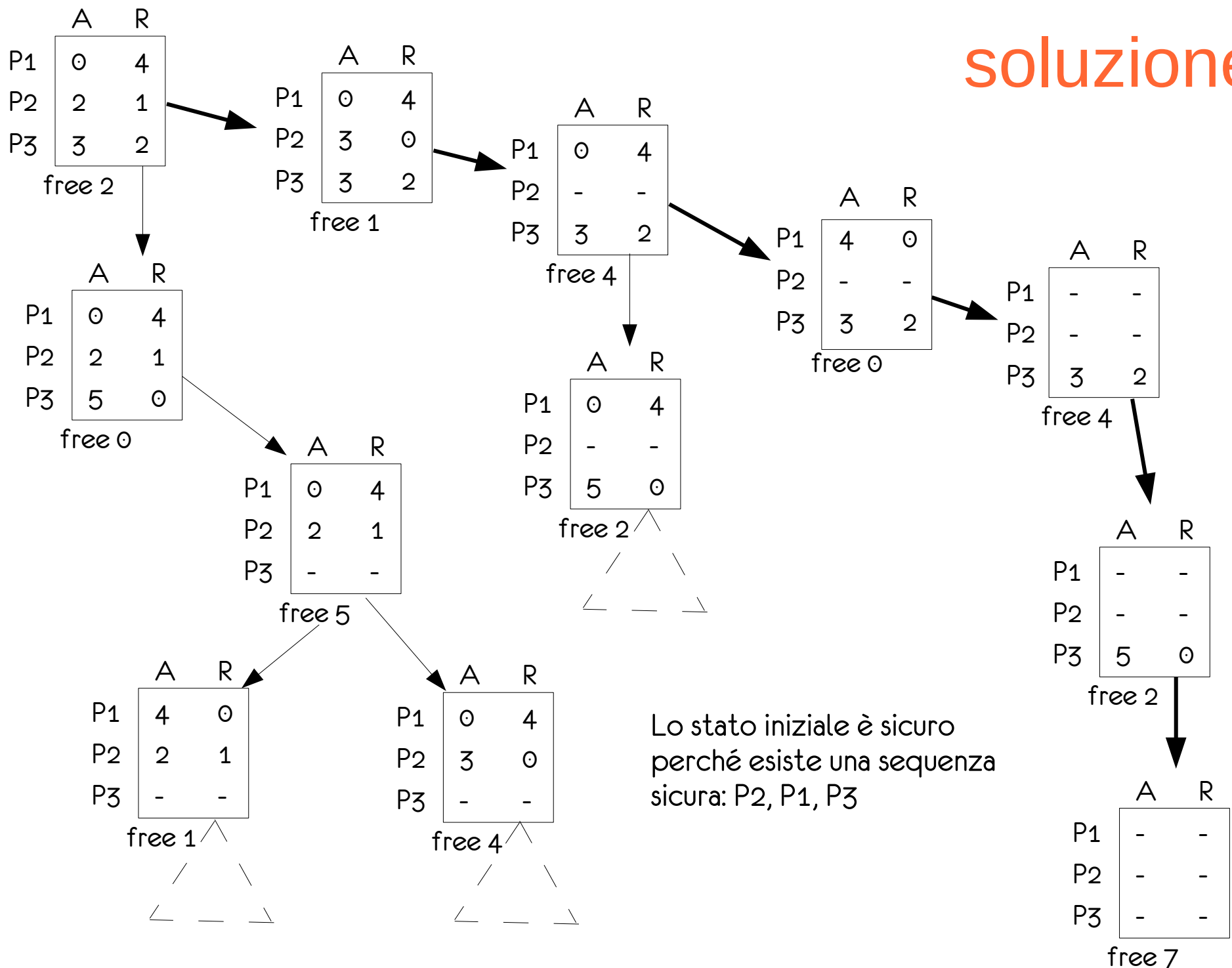
Stato sicuro: il grafo non contiene alcun ciclo

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



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 $i \mid !\text{Fine}[i] \ \&\& \ \text{richieste}[i] < \text{Lavoro}$? Sì, richieste[2]
Lavoro = Lavoro + Assegnate[2] = {0,1,2}
Fine = {false, true, false}

c'è un $i \mid !\text{Fine}[i] \ \&\& \ \text{richieste}[i] < \text{Lavoro}$? Sì, richieste[3]
Lavoro = Lavoro + Assegnate[3] = {2,1,3}
Fine = {false, true, true}

c'è un $i \mid !\text{Fine}[i] \ \&\& \ \text{richieste}[i] < \text{Lavoro}$? Sì, 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 $i \mid !\text{Fine}[i] \ \&\& \ \text{richieste}[i] < \text{Lavoro}$? Sì, 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

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)

soluzione 1/2

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