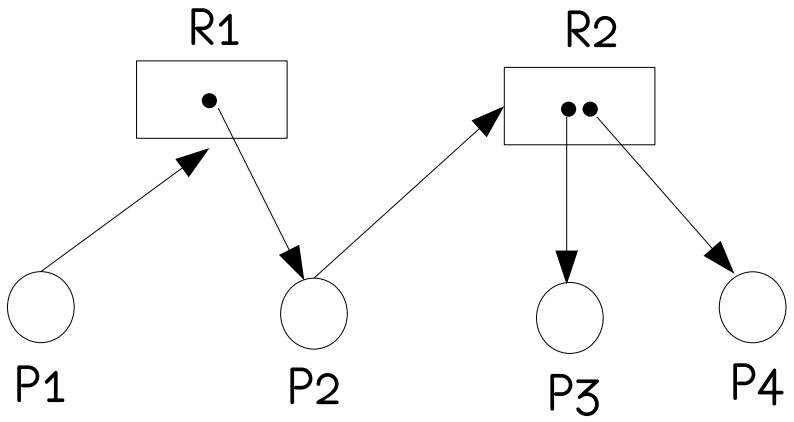
# Grafo di assegnazione delle risorse

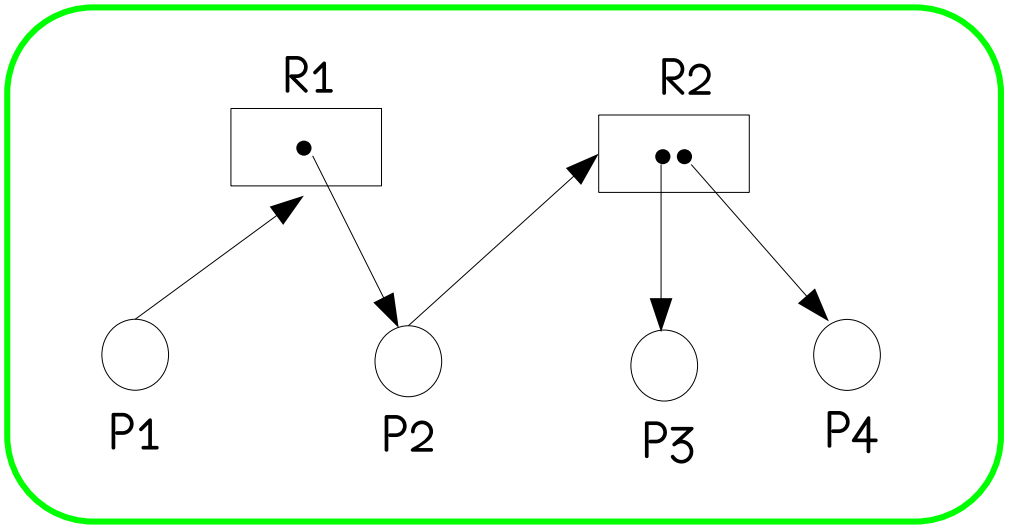
- Rappresentazione delle assegnazioni che permette di rilevare situazioni di deadlock
- È un grafo G = <V, E> tale per cui:
  - V è l'insieme dei vertici ed è partizionato in due sottoinsiemi P ed R, $P \cap R = \Phi$:
    - $P = \{P1, \ldots, Pn\}$ è l'insieme di tutti i processi del sistema
    - $R = \{R1, \ldots, Rm\}$ è l'insieme di tutte le classi di risorse del sistema
  - E è l'insieme degli archi:
    - Un'arco direzionato da Ri a Pj, $Ri \rightarrow Pj$, indica che una risorsa di classe Ri è stata assegnata al processo Pj (arco di assegnazione)
    - Un'arco direzionato da Pj a Ri, $Pj \rightarrow Ri$, indica che il processo Pj ha richiesto ed è in attesa di una risorsa di tipo Ri (arco di richiesta)

# Rappresentazione grafica del grafo di assegnazione delle risorse

- Ogni processo Pi è rappresentato da un cerchietto
- Ogni classe di risorsa Ri 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)

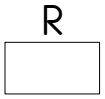


# Notazione grafica

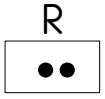




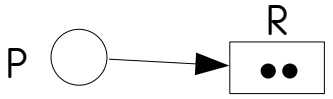
Processo P



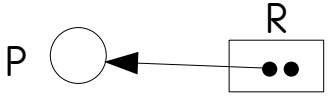

Classe di risorsa R




Classe di risorsa R con due istanze

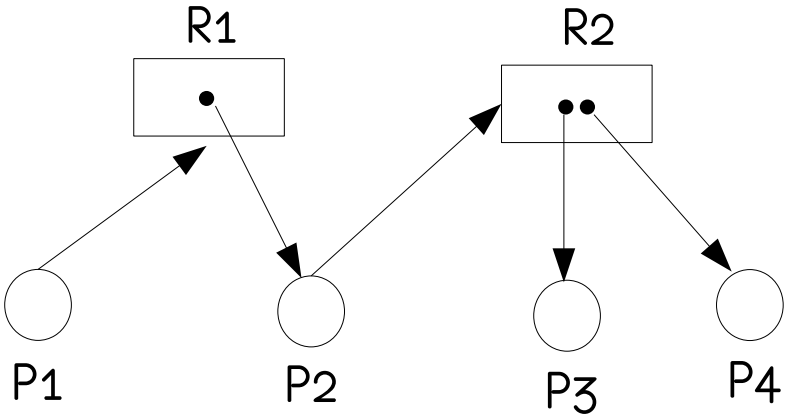



Arco di richiesta




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?

→ informazioni di tipo statico

Chi detiene quale risorsa?

Chi richiede quale risorsa?

→ 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

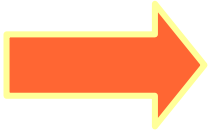
Quante classi di risorse abbiamo?

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 ...

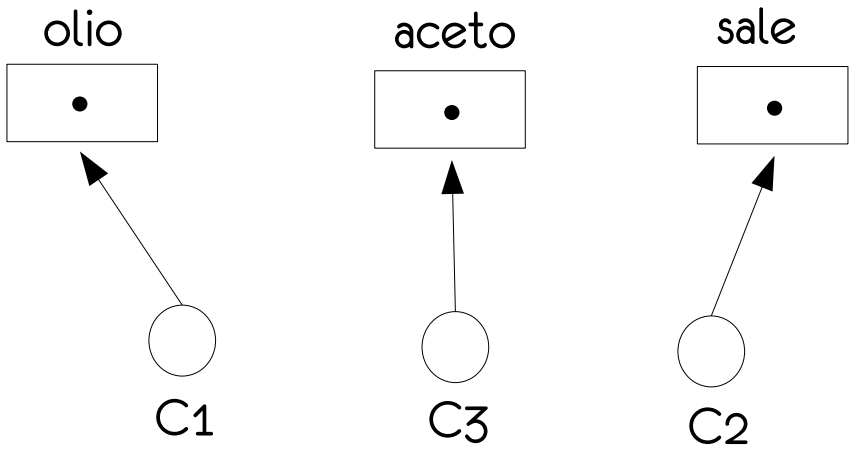
rilascia risorse

**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 ...

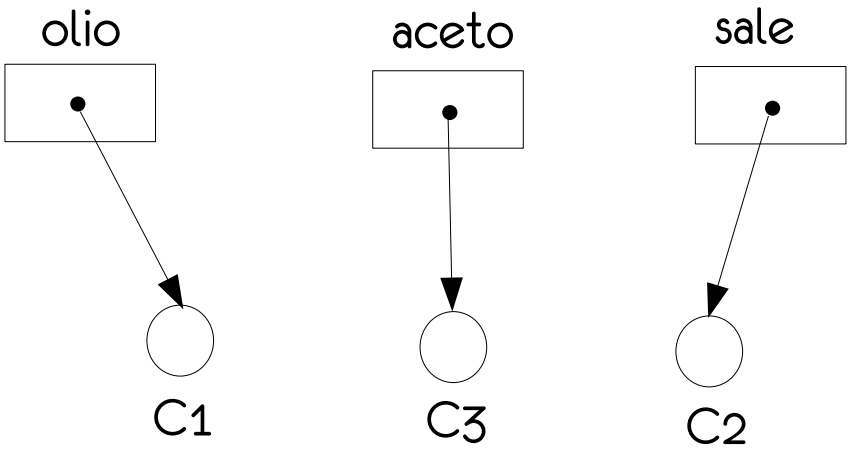
rilascia risorse

**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 ...

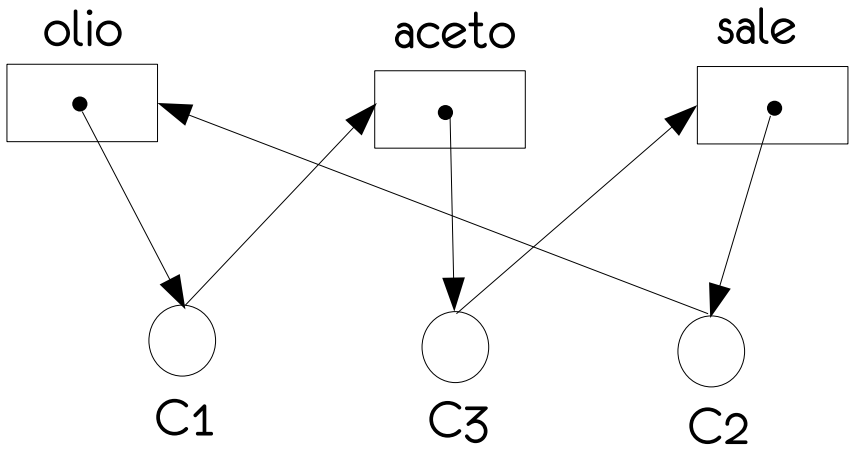
rilascia risorse

**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 ...

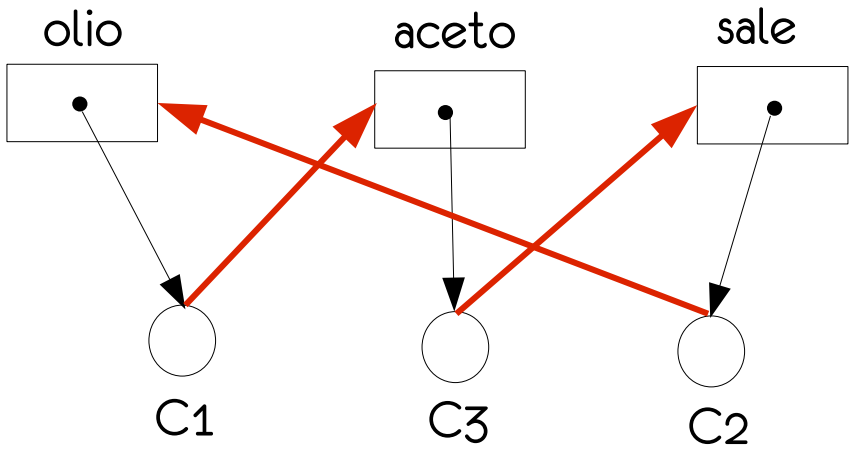
rilascia risorse

**Cuoco3**

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

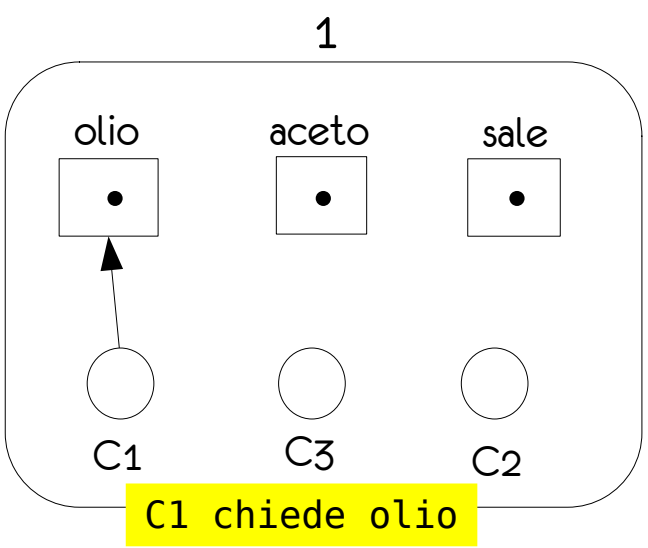
... cucina ...

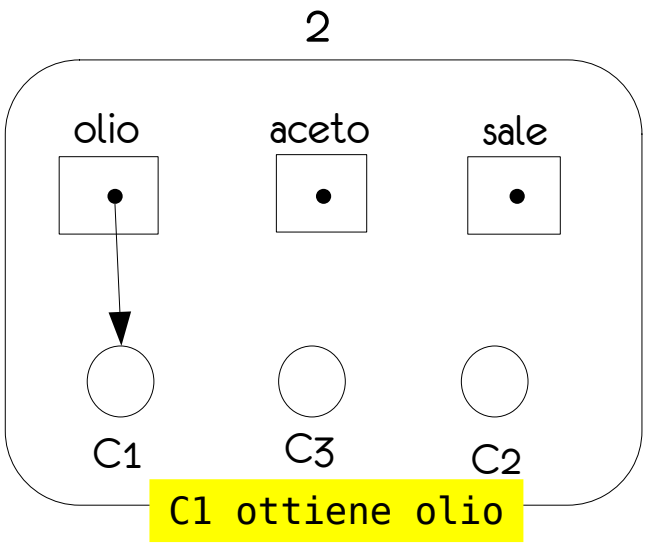
rilascia risorse

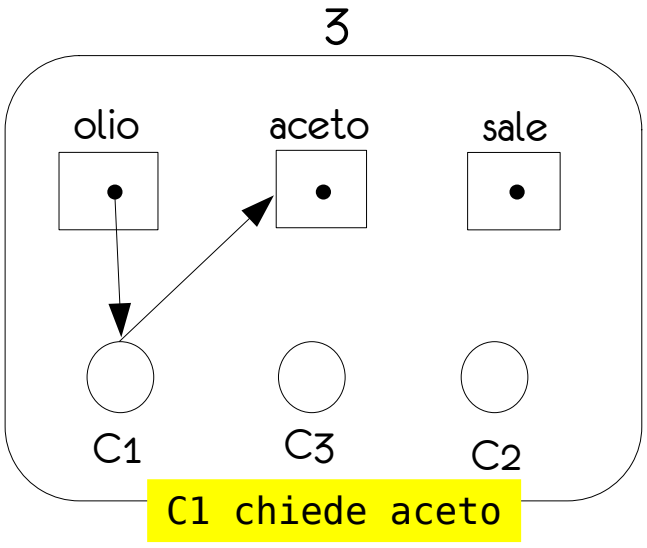


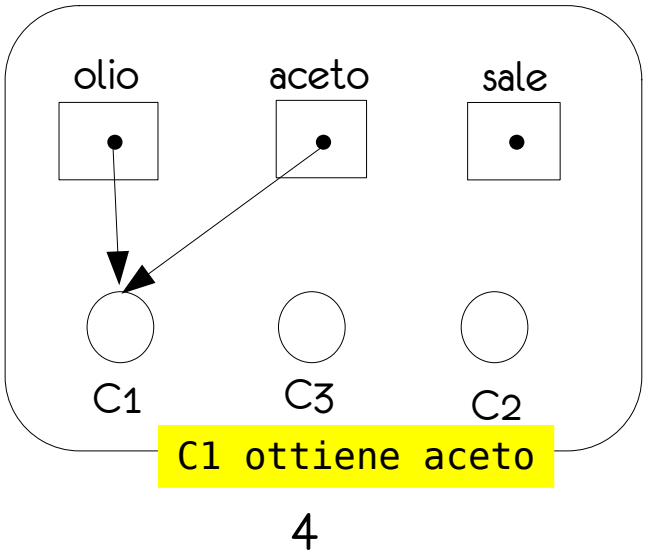

C1 aspetta C3, che asptta C2, che aspetta C1: deadlock

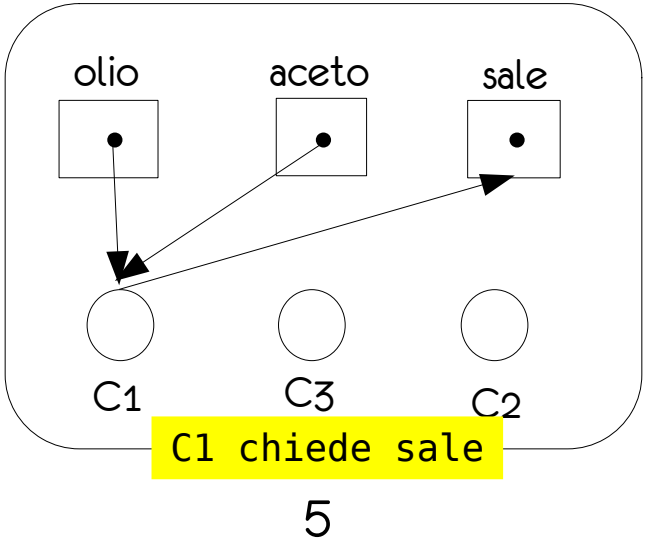
# I 3 cuochi

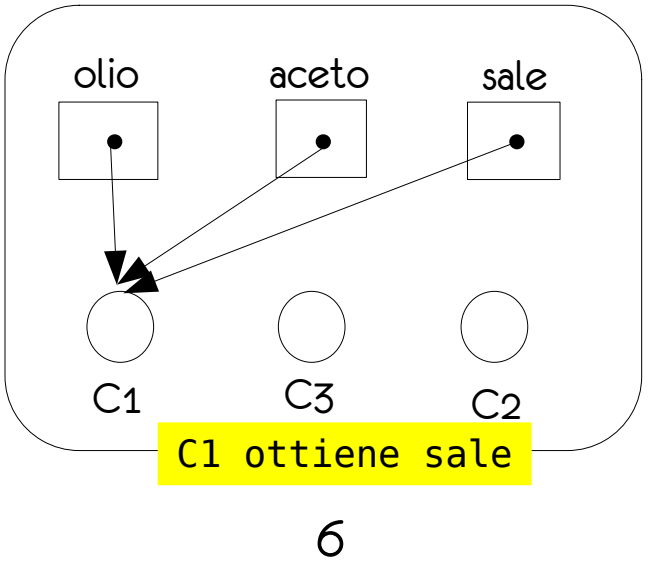



eccetera ... esecuzione senza deadlock

# 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

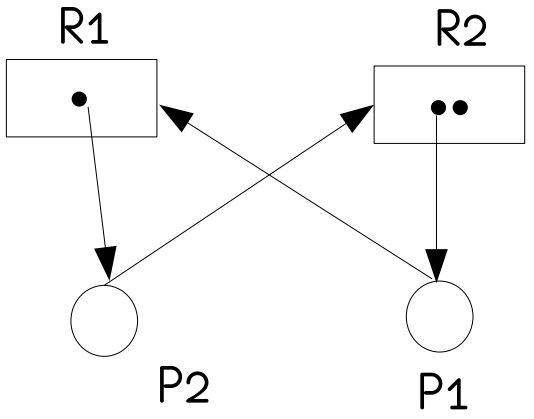
# Uso del grafo di assegnazione

- 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

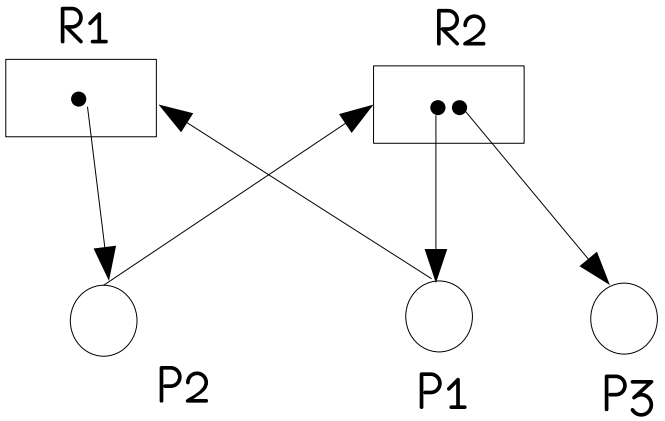
esempio



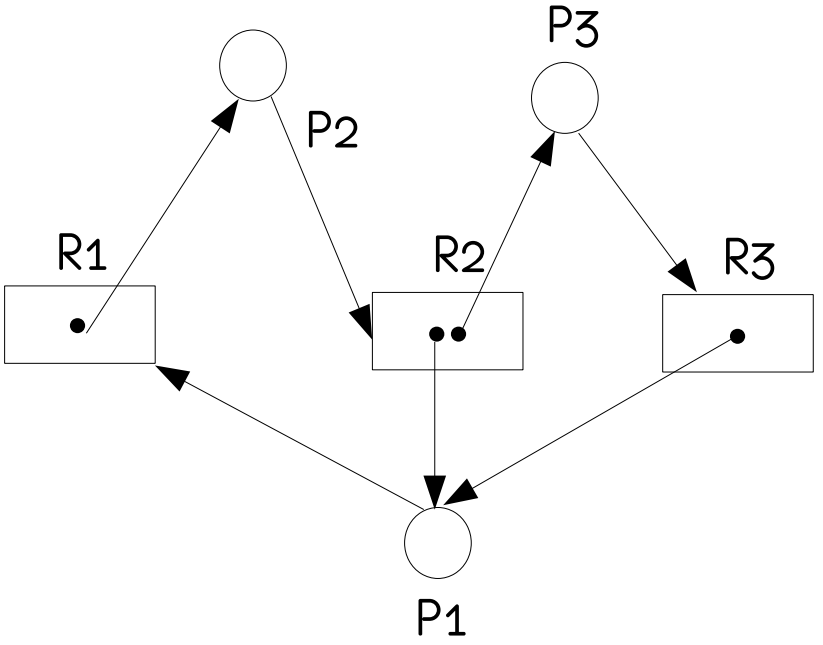

Non è un deadlock!

Infatti è possibile assegnare l'istanza libera di R2 a P2

# Altri esempi



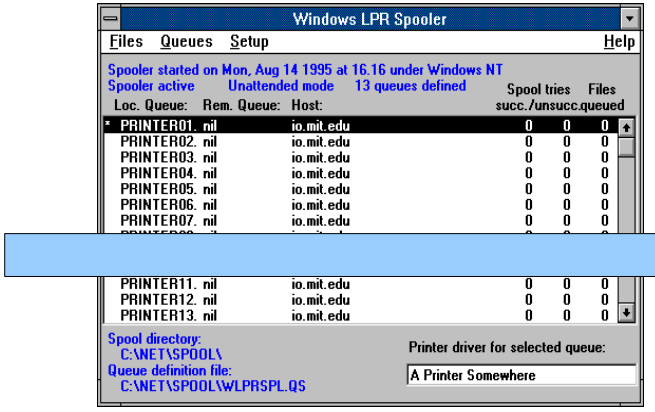

Deadlock? No
Appena P3 libera R2
P2 riparte

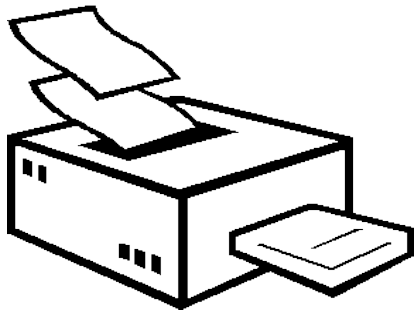



Deadlock? Si
P1 aspetta R1 che è di P2
P2 aspetta R2, le cui istanze
sono di P1 e P3 e
P3 aspetta R3 che è di P1

# Un caso reale: spooling

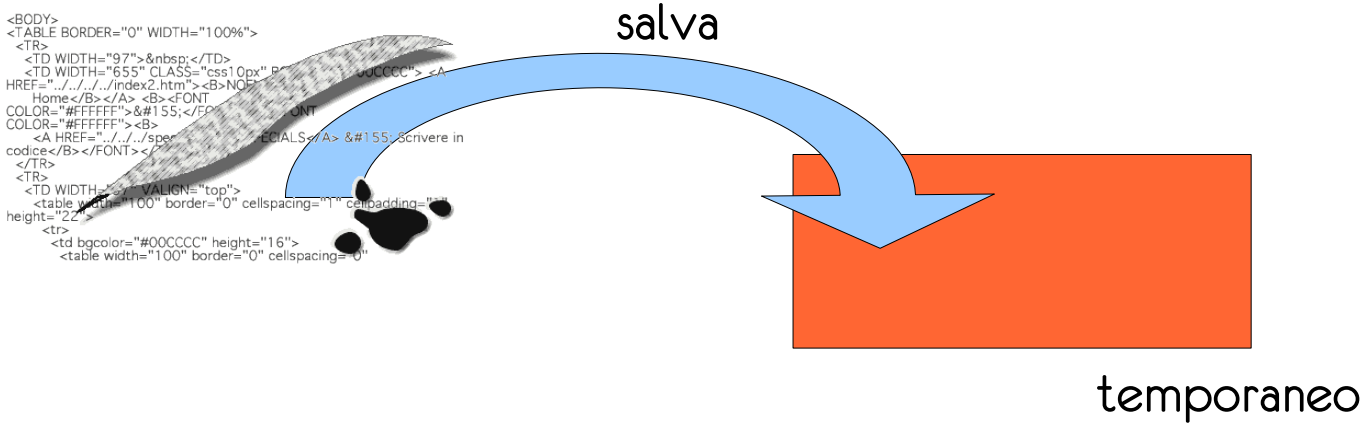
- Un processo deve stampare un documento:
  la stampante gestisce una pagina per volta, quindi il processo dovrebbe attendere la terminazione della stampa della pagina corrente prima dell'invio alla stampante della successiva
- Problema: lentezza!!
- Per migliorare l'esecuzione del processo utente si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

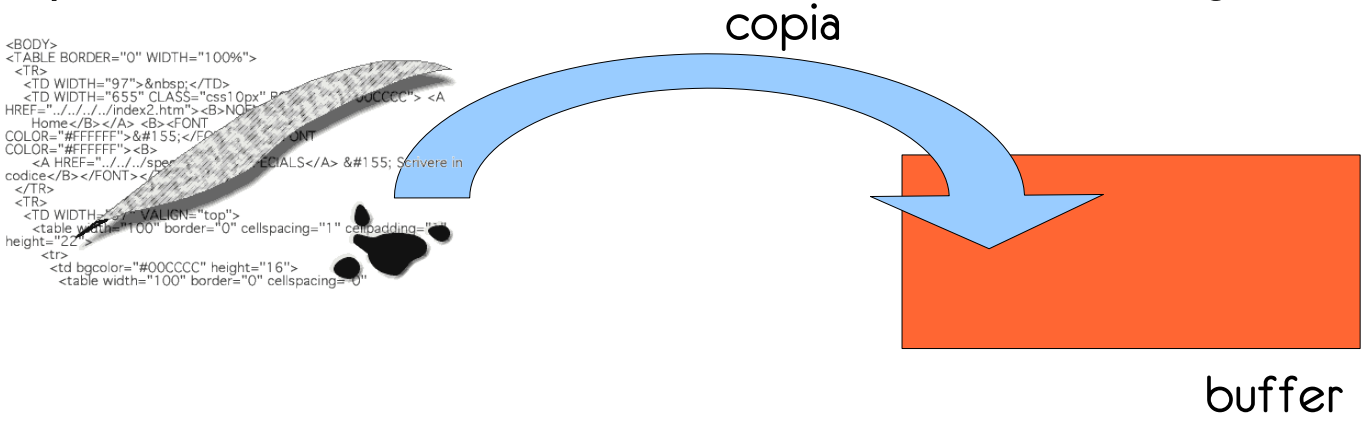
- Molti sistemi di spooling gestiscono solo documenti completamente codificati
- Quindi un processo di stampa deve per es.:
  - convertire il documento in un formato di stampa
  - salvare il risultato in un file temporaneo
- Risorsa condivisa: memoria
- Cosa succede se si esaurisce la memoria e nessun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

# Scenario

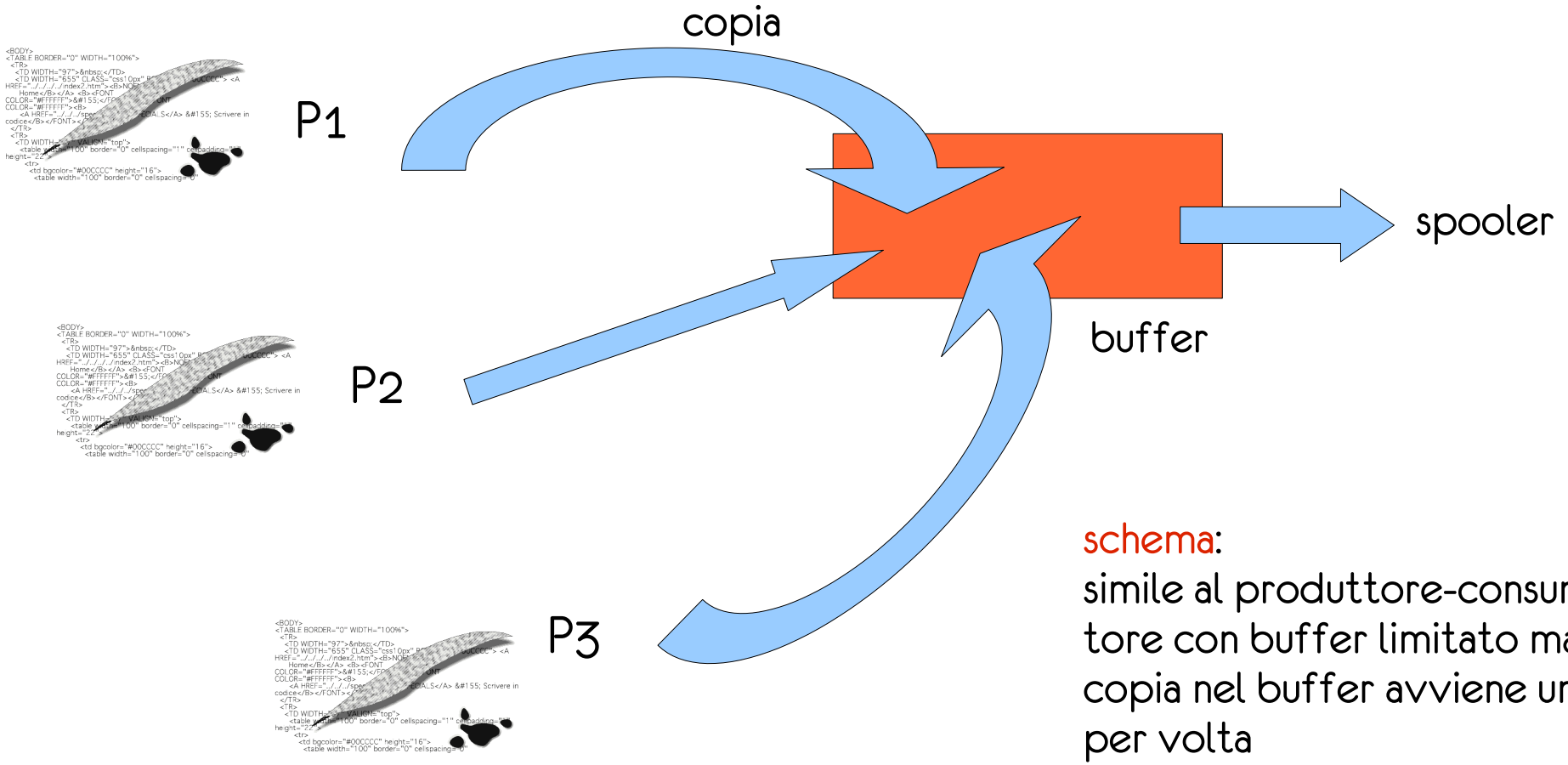
- Molti sistemi di spooling gestiscono solo documenti prodotti in modo completo
- Quindi un processo di stampa deve per es.:
  - convertire il documento in un formato di stampa
  - salvare il risultato in un file temporaneo
- Risorsa condivisa: memoria
- Cosa succede se si esaurisce la memoria e nessun 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



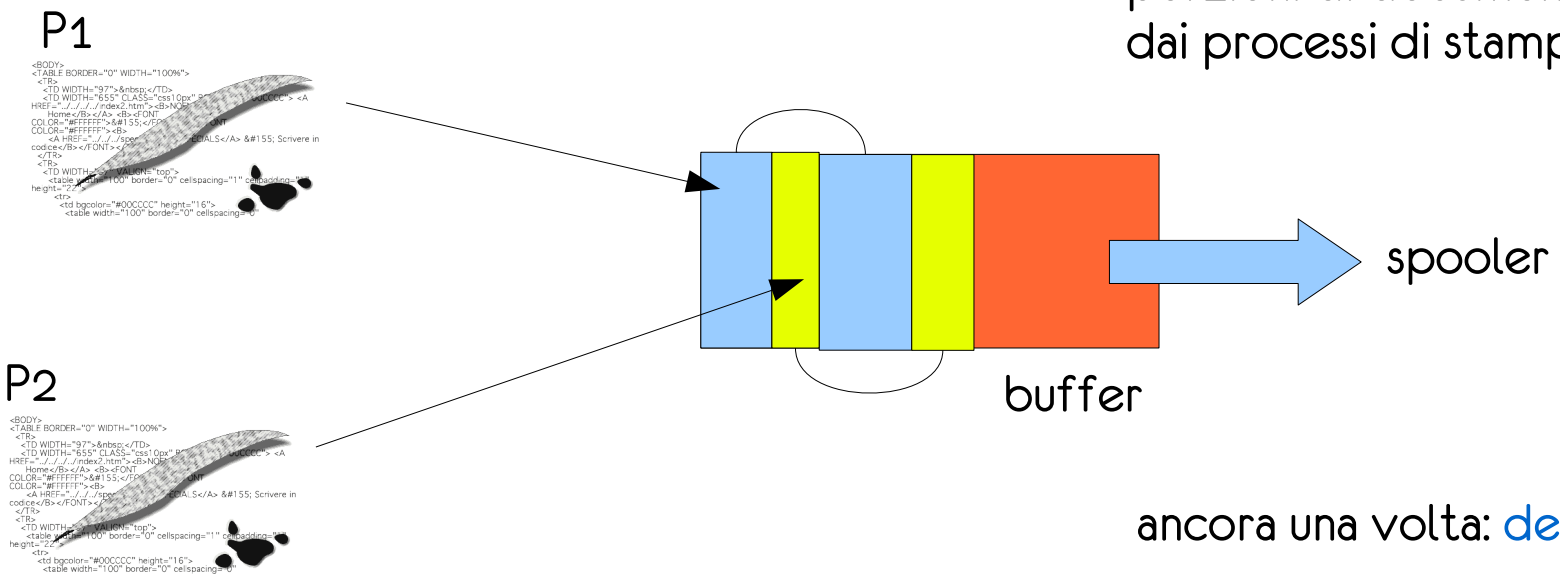
# Scenario




schema:
simile al produttore-consumatore con buffer limitato ma la copia nel buffer avviene un po' per volta

# 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

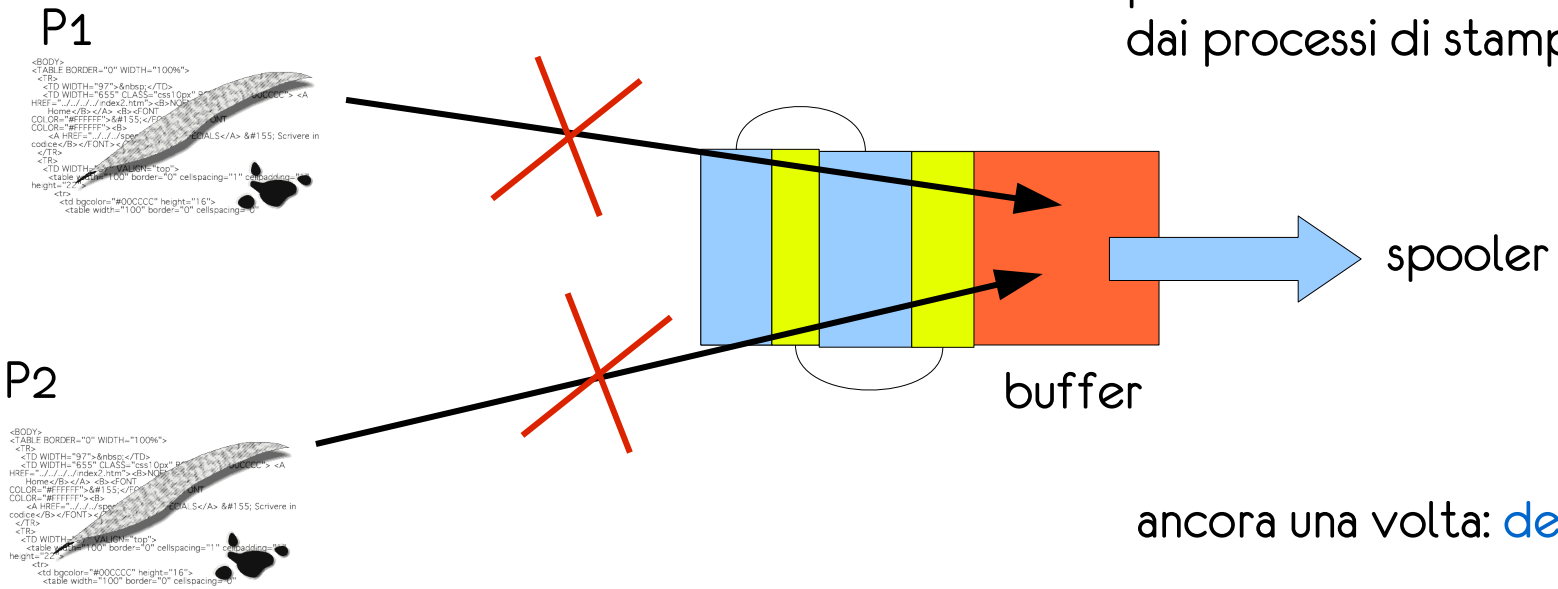
---

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?

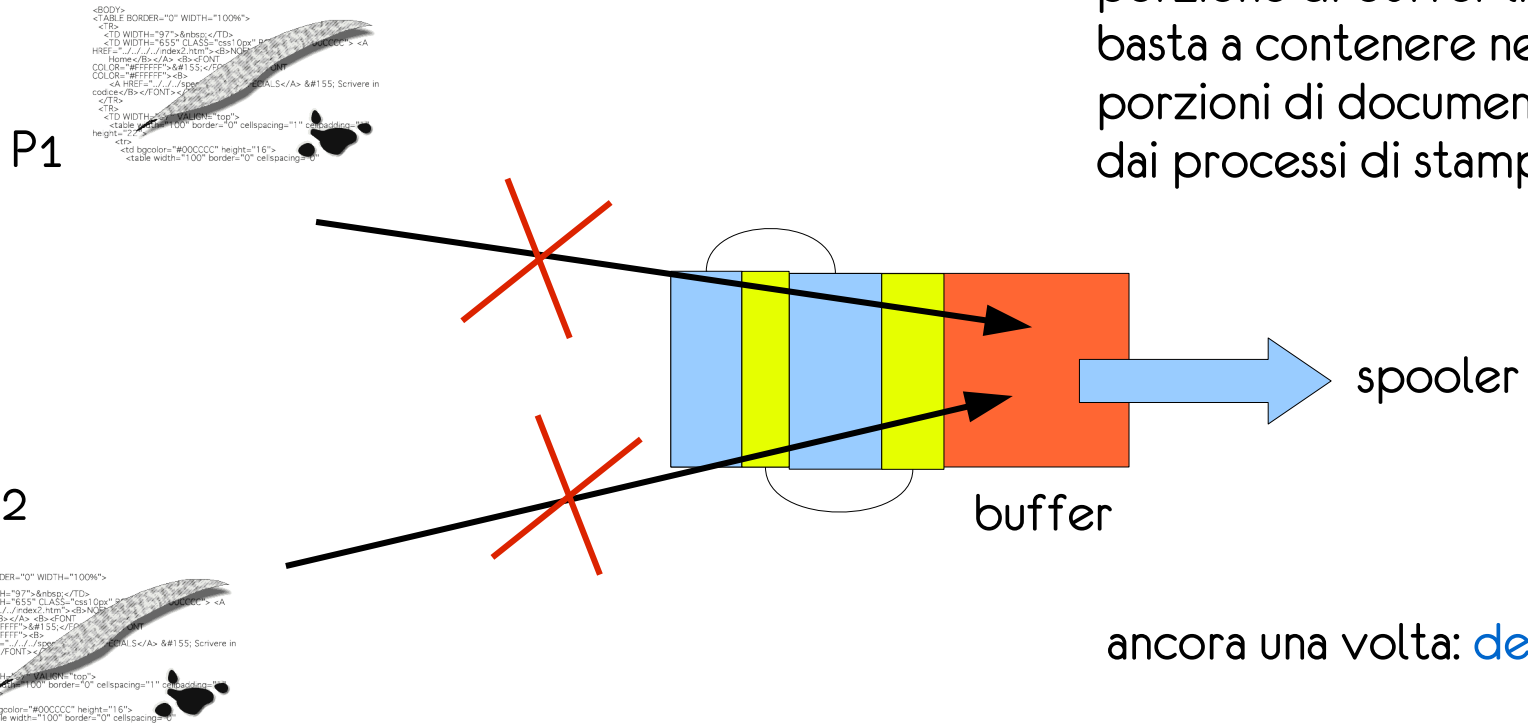



ancora una volta: deadlock

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

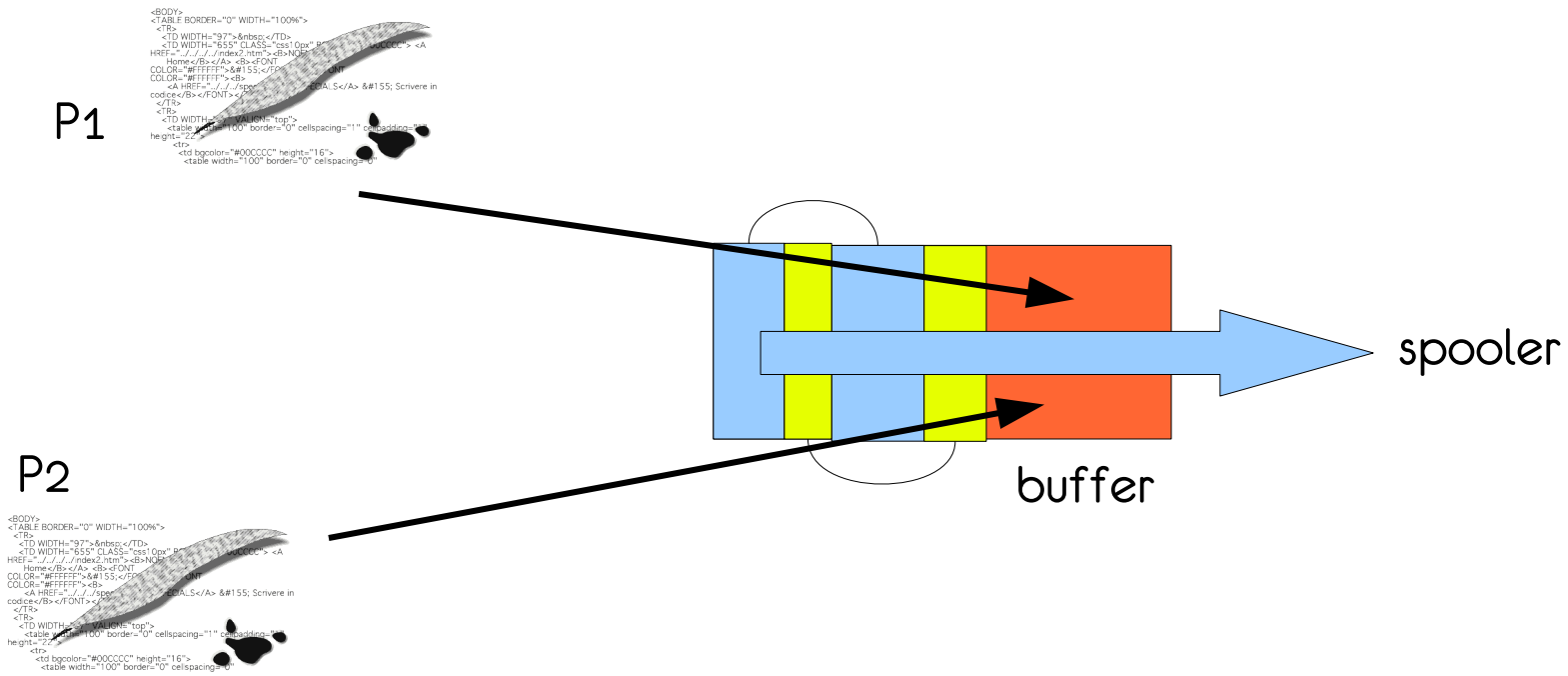
Problema: cosa succede se la porzione di buffer libera non basta a contenere nessuna delle porzioni di documento prodotte dai processi di stampa?




ancora una volta: deadlock

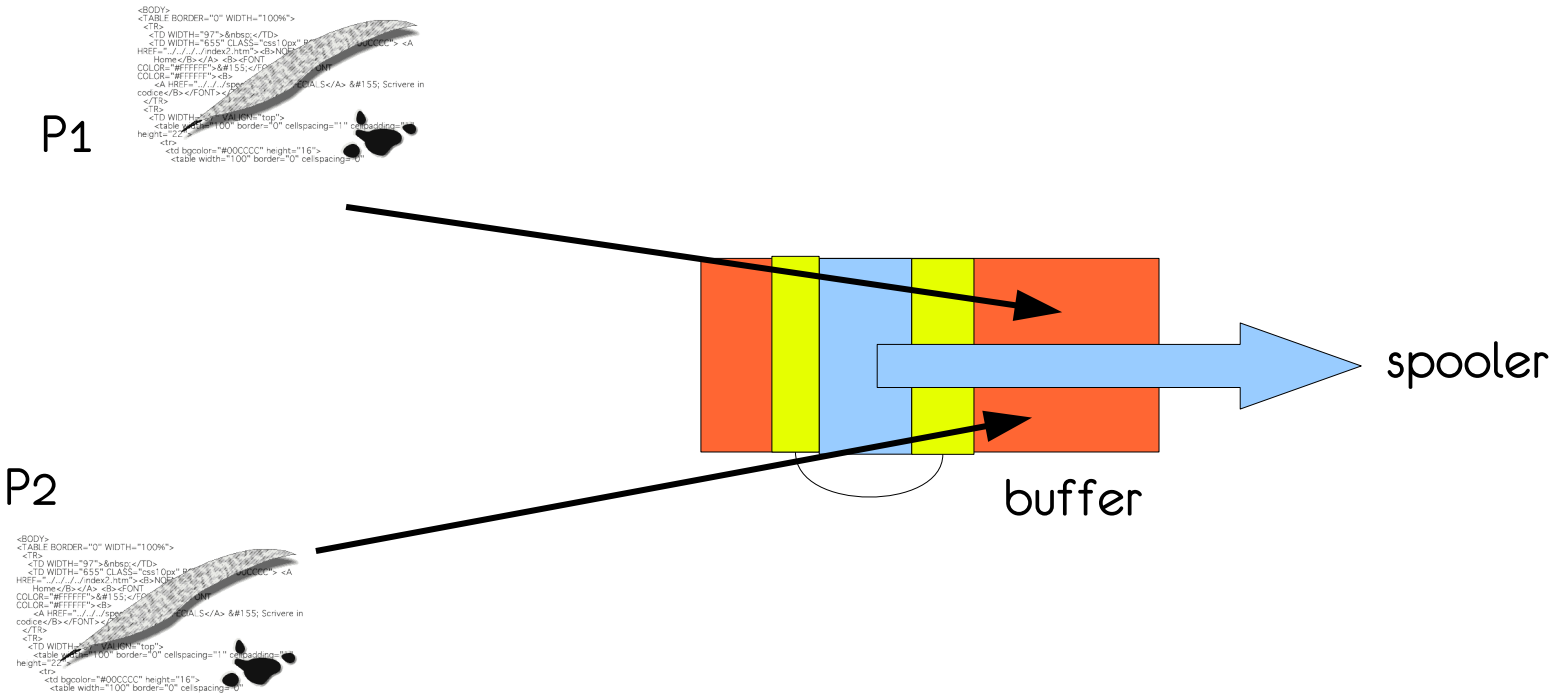
Può servire una politica del tipo: se il buffer è pieno al 75% nessun nuovo processo di stampa può iniziare a copiare un nuovo documento nel buffer?
**No**, perché il deadlock non è necessariamente causato da un nuovo processo possono bastare quelli già attivi

# 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: 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

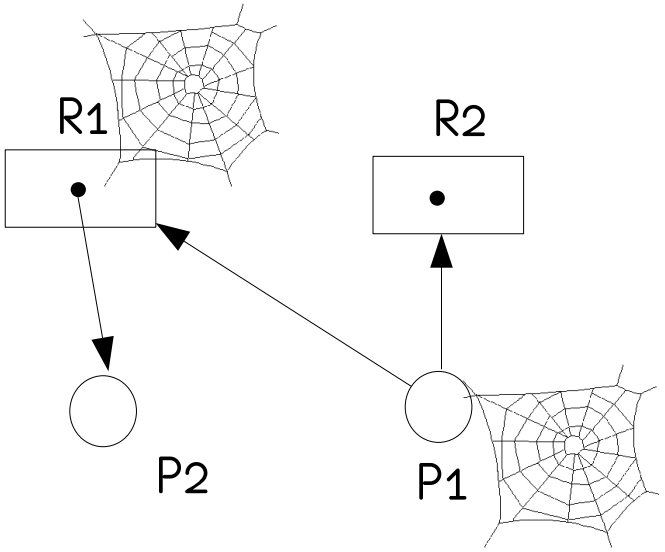
# Prevenzione

le 3 strategie di Havender

- **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 prossesso
  - 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:



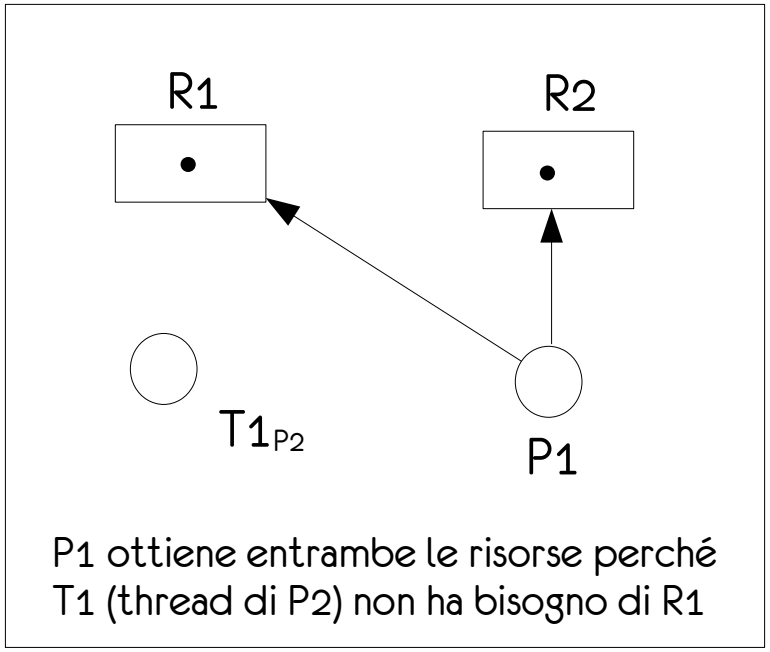

P1 richiede sia R1 che R2 ma l'unica istanza di R1 è assegnata a P2

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

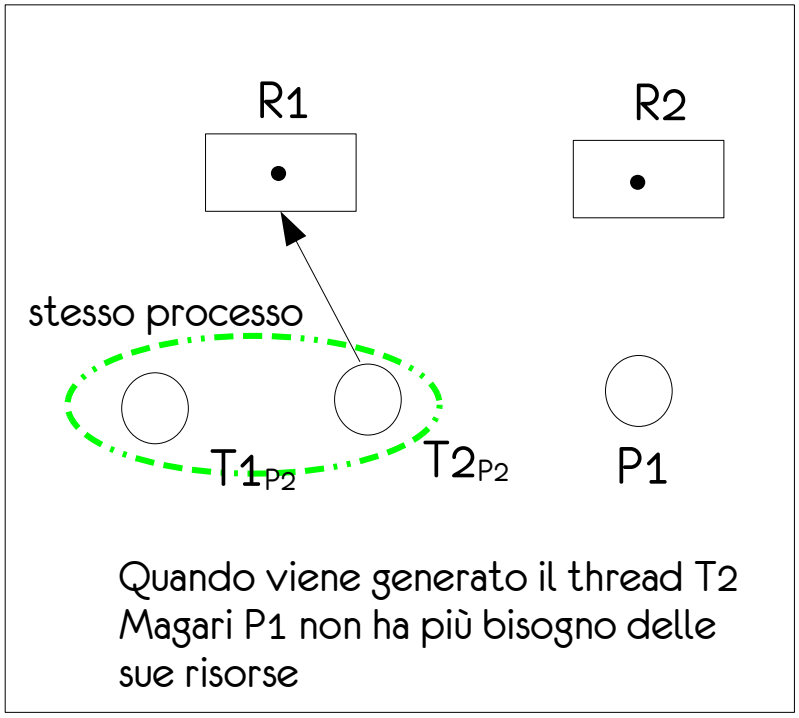
R2 è libera, P1 potrebbe usare sia R1 che R2 prima che a P2 occorra effettivamente R1

# Possibile miglioramento

- Questaa 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

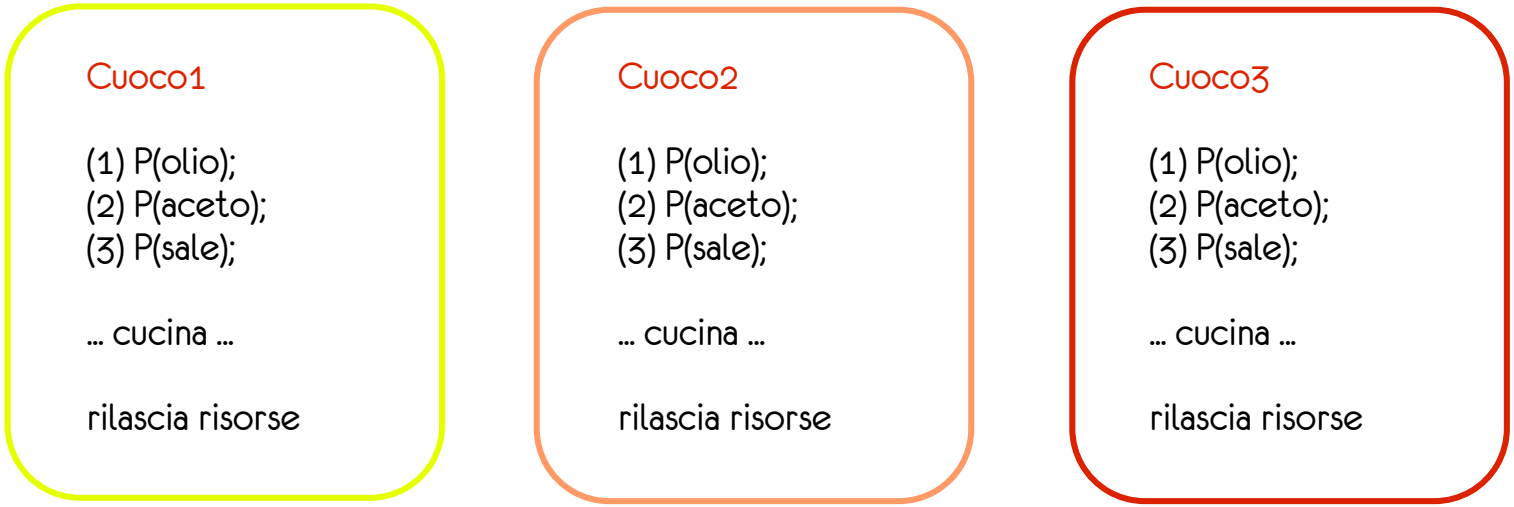
- L'ultima strategia di Havender comporta l'avoidance dell'attesa circolare
- Ogni risorsa ha assegnato un numero, utilizzato per quella risorsa soltanto
- Sulla base di tali numeri le risorse risultano ordinabili in ordine strettamente crescente (R1 < R2 < ... < Rn)

# 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 < ... < 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: olio ← 1, aceto ← 2, sale ← 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

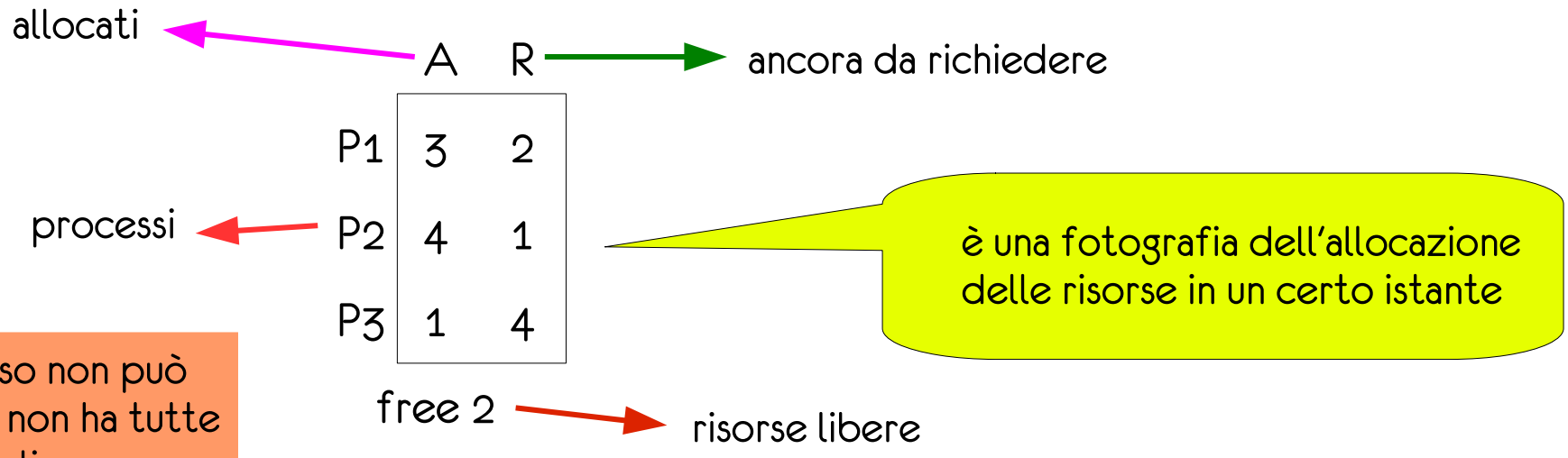
- In certi contesti non è possibile inibire a priori una delle condizioni necessarie affinché si abbia il deadlock (applicando le strategie di Havender)
- **Questo non significa che non si possa evitare il deadlock**
- I metodi che consentono di fare ciò **richiedono alcune informazioni**, per esempio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

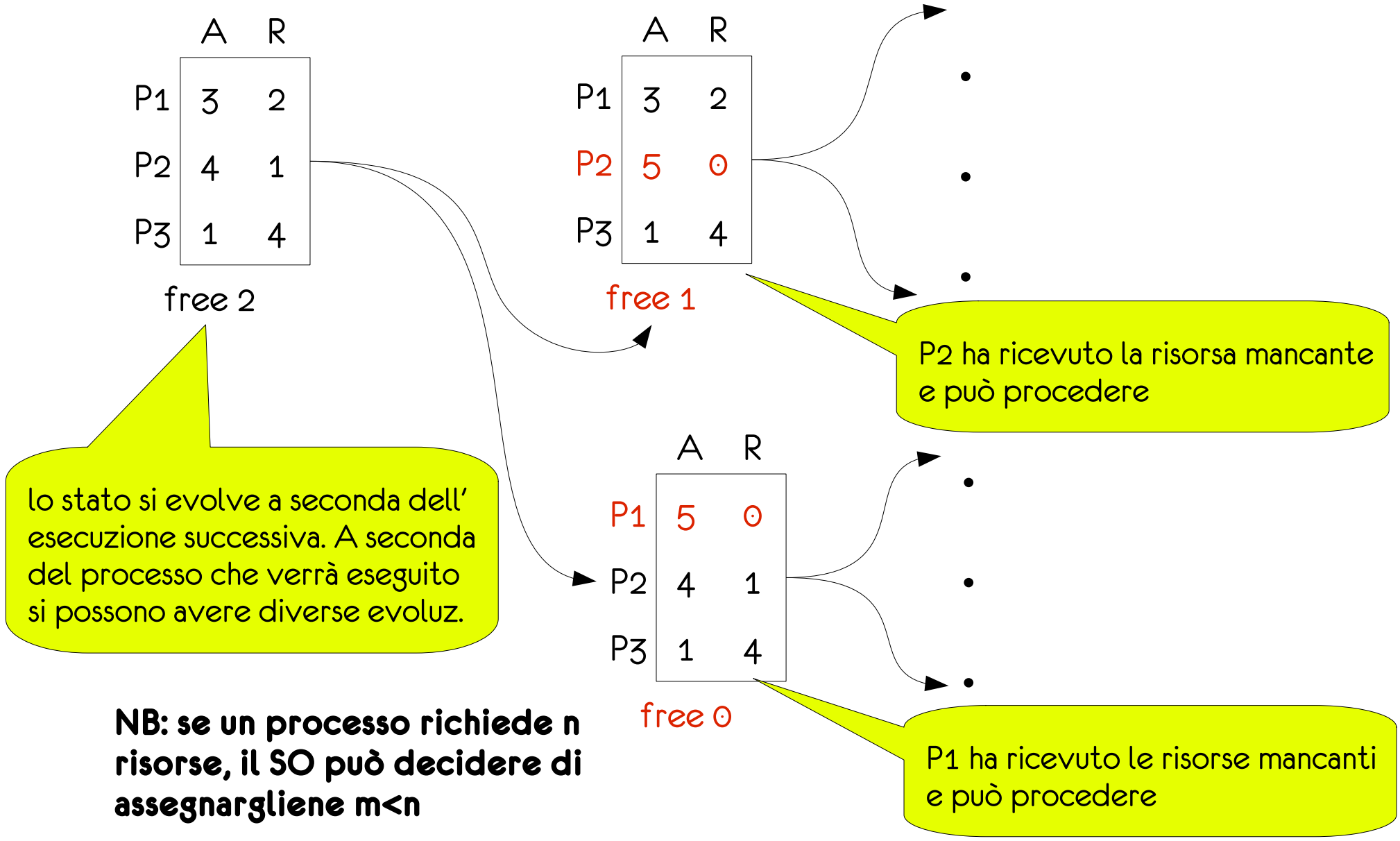
# Stato di allocazione delle risorse

- 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:



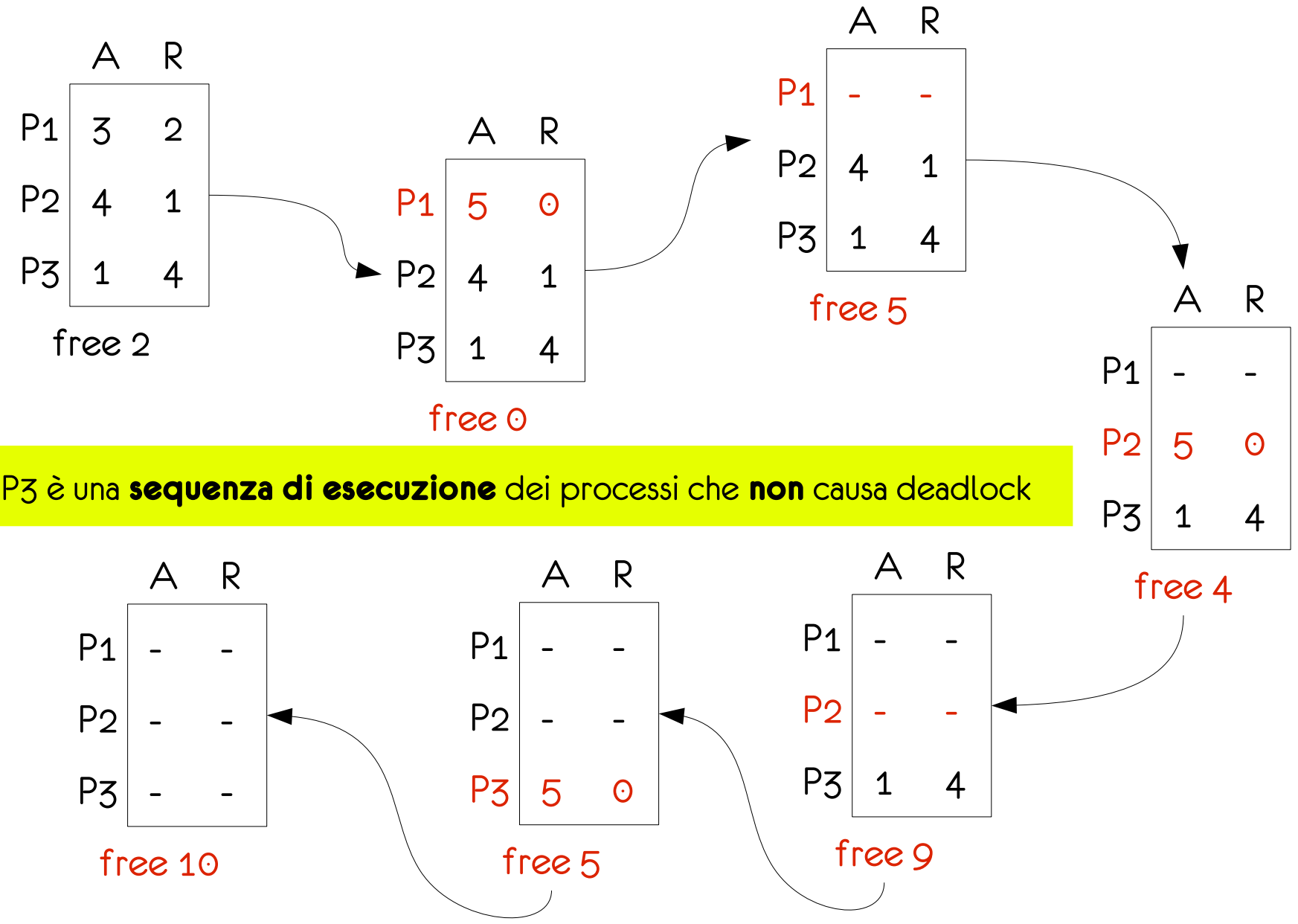

NB: un processo non può procedere se non ha tutte le risorse che gli servono

# Stato di allocazione delle risorse



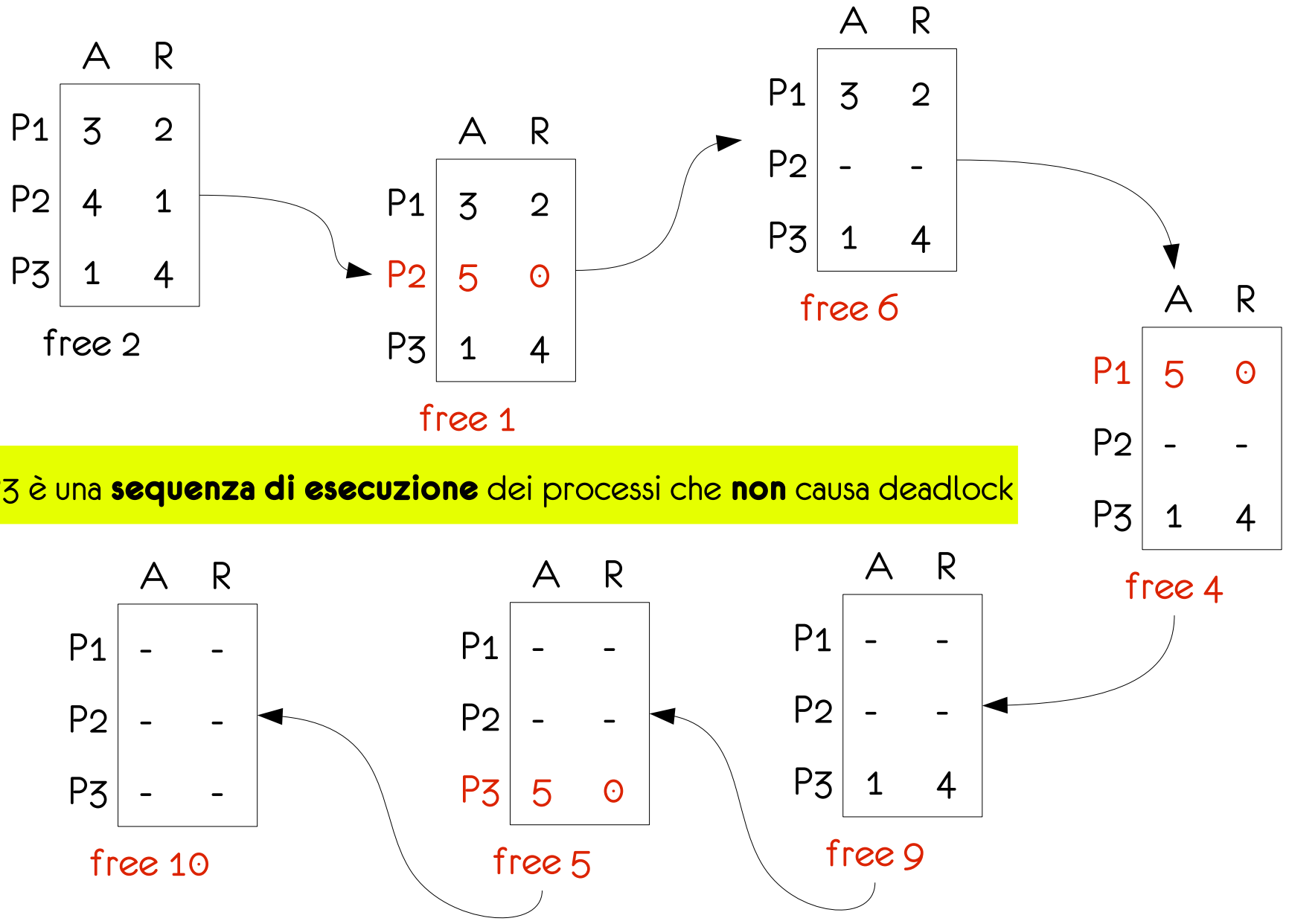

**NB: se un processo richiede n risorse, il SO può decidere di assegnargliene m<n**

# Sequenza di esecuzione



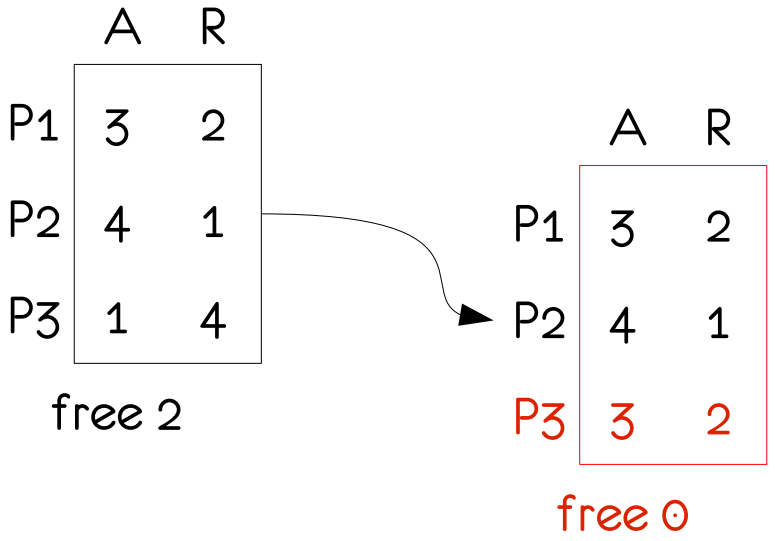

P1, P2, P3 è una **sequenza di esecuzione** dei processi che **non** causa deadlock

# Altra sequenza di esecuzione



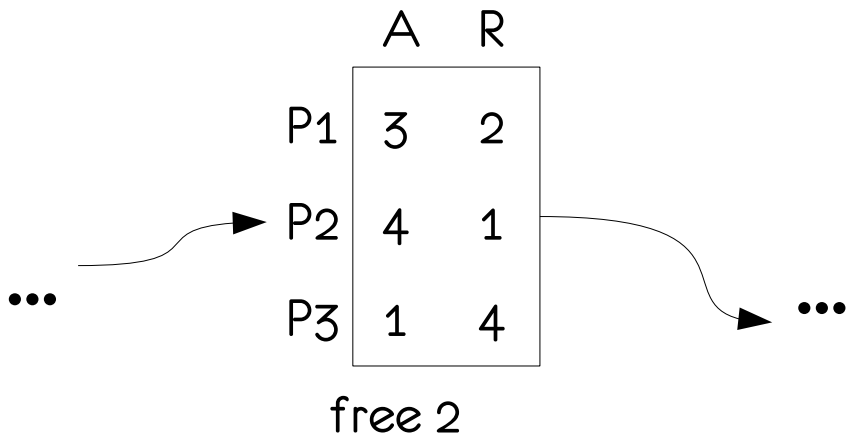

P2, P1, P3 è una **sequenza di esecuzione** dei processi che **non** causa deadlock

# 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** $<P^1, ..., P^n>$ 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

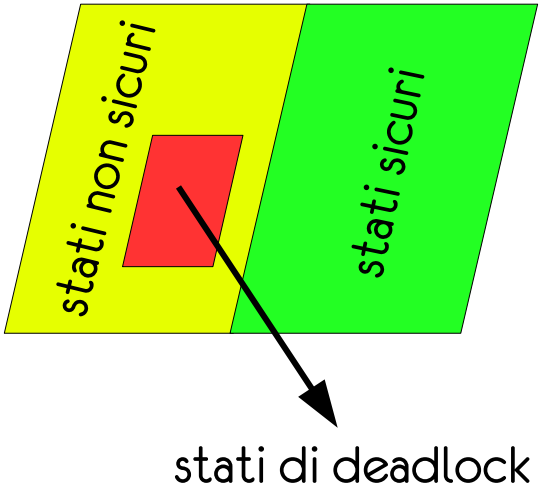
# Perché funziona?

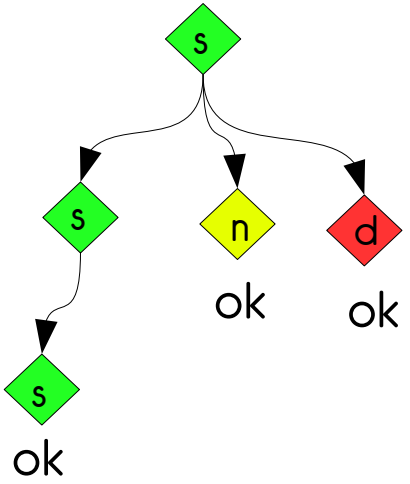
- **Una sequenza sicura non presenta deadlock perché per ogni processo $P^i$ sono veri i seguenti casi:**
  - $P^i$ ha tutte le risorse che gli servono
  - oppure a $P^i$ basta aspettare la terminazione di qualche processo precedente per poter procedere
- **può capitare che si generi una catena di attese 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 $P^1$:

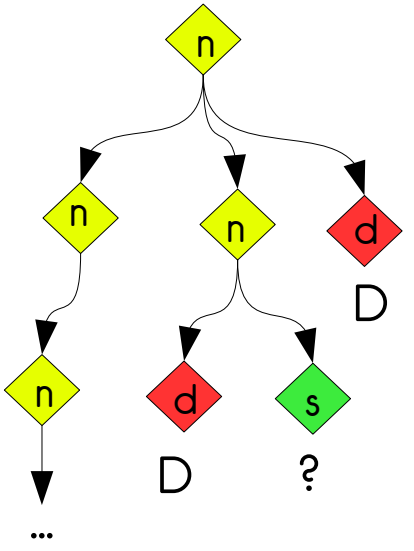
    per definizione di sequenza sicura tutte le richieste che il processo $P^1$ effettuerà da qui alla sua fine devono essere o disponibili oppure in possesso di un processo $P^j$ con $j<1$, che però non esiste

# Stati sicuri e sequenze sicure

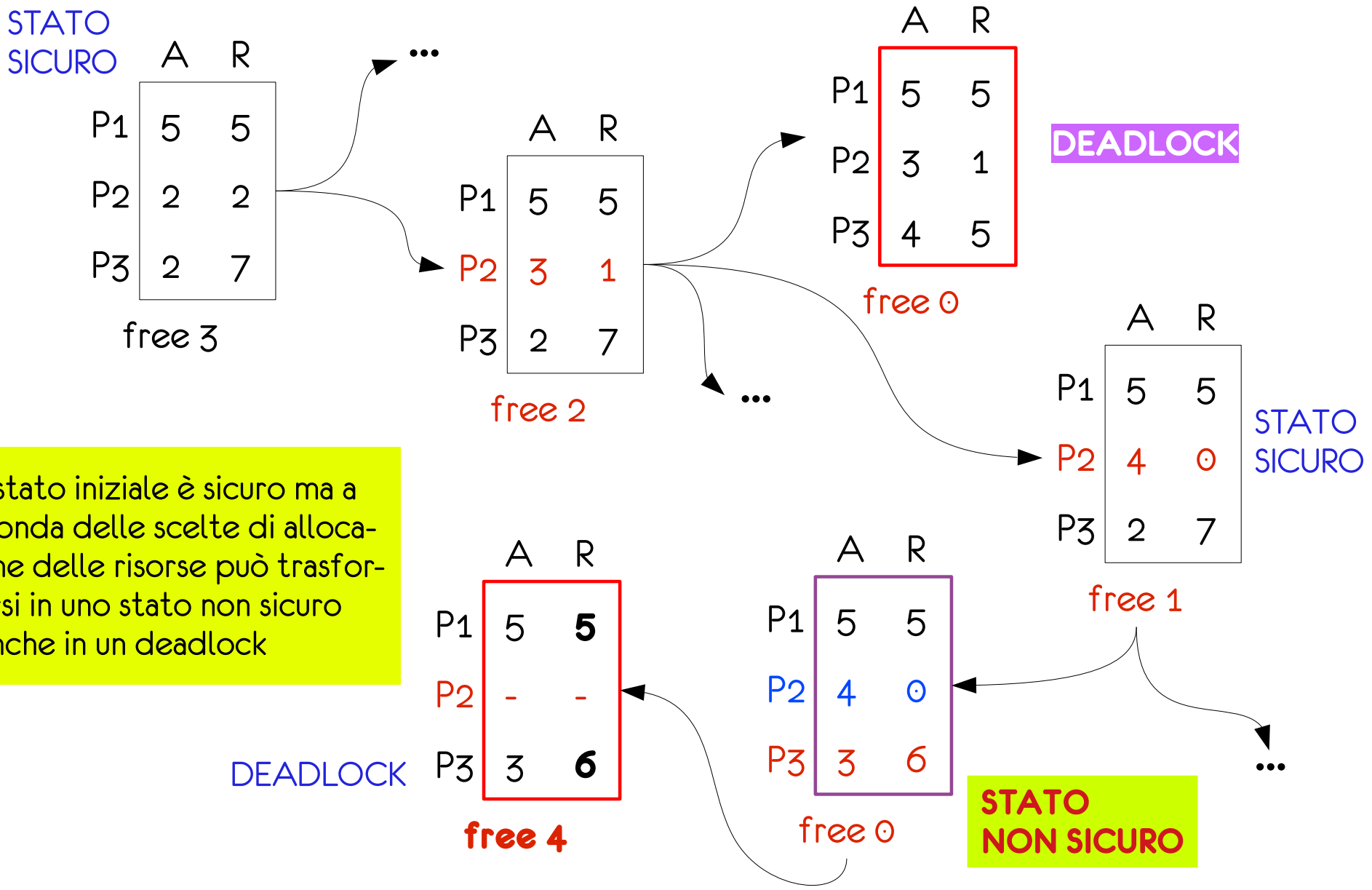
- Uno stato è sicuro se da esso si dirama almeno una sequenza sicura, quindi **se esiste** almeno un ordinamento dei processi che è una sequenza sicura
- Uno stato non sicuro non necessariamente è uno stato di deadlock ma può portare al deadlock
- Uno stato non sicuro può evolvere in uno stato sicuro?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



# Evoluzione degli stati




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