SAPIENZA
UNIVERSITÀ DI ROMA

Corso di laurea in Ingegneria dell'Informazione
Indirizzo Informatica

Reti e sistemi operativi

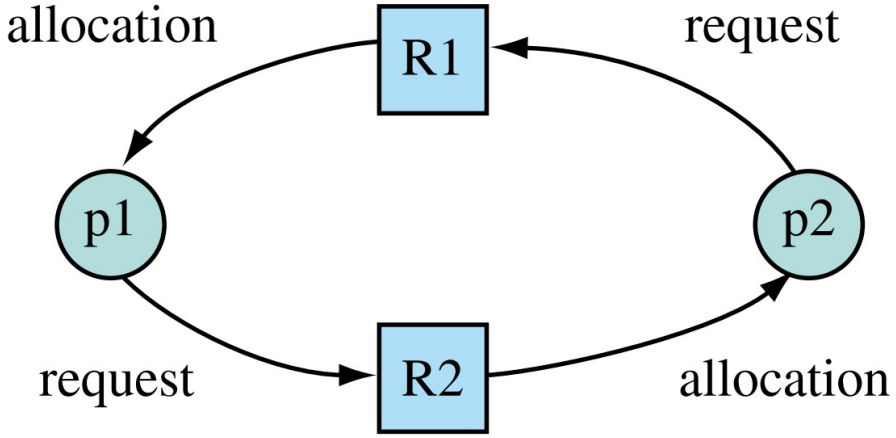
Deadlock

Problema del deadlock

- Uno o più processi sono bloccatti in attesa di una risorsa che non potrà mai essere concessa.
- Generalmente, ci sono tre possibili approcci al deadlock:
  - **Prevenire** i deadlock attraverso un'attenta analisi del sistema
  - **Rilevare** il verificarsi di un deadlock, e intraprendere azioni correttive
  - **Ignorare** il problema e sperare per il meglio (quello che succede di solito, es. in Linux e Windows)

Possibili situazioni di deadlock

- Allocazione circolare di risorse:



- Informalmente si può considerare deadlock anche la situazione in cui un processo attende per una quantità di risorse maggiori delle disponibili (es. più memoria di quella installata nel sistema) → non considerato nella presente trattazione

Caratterizzazione dei deadlock

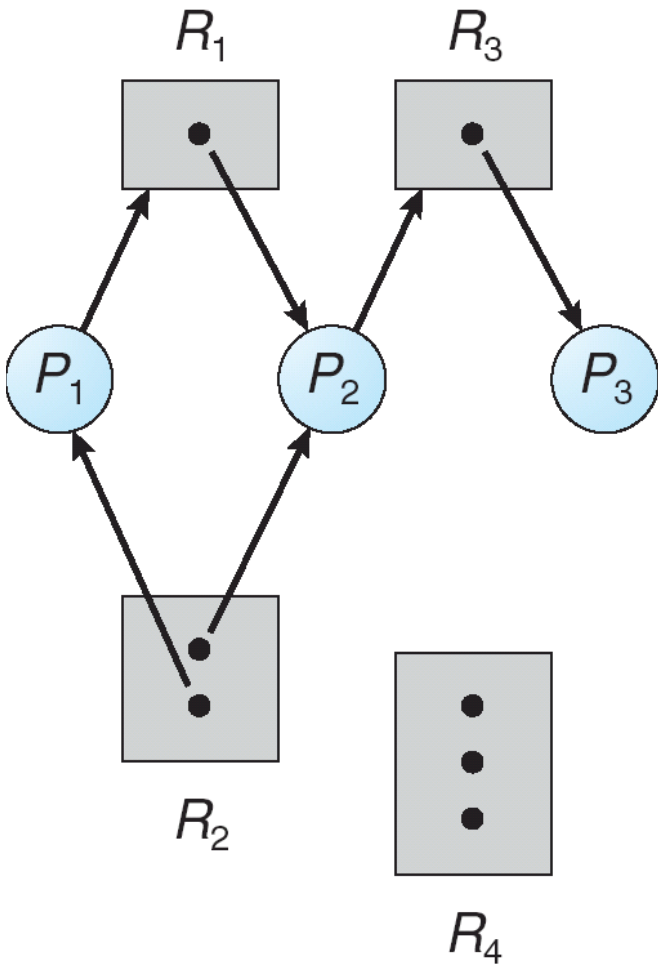
- Un deadlock può verificarsi quando le seguenti 4 condizioni sono simultaneamente soddisfatte:
  - **(1) Mutua esclusione**: una risorsa può essere usata solo da un processo per volta
  - **(2) Acquisizione e attesa**: un processo che ha acquisito almeno una risorsa sta attendendo di acquisire altre risorse correntemente acquisite da altri processi
  - **(3) Preemption non possibile**: una risorsa può essere rilasciata solo volontariamente dal processo che ne ha acquisito il controllo, dopo aver completato il proprio task
  - **(4) Attesa circolare**: esiste un set di processi {P0, P1, …, Pn} in attesa di acquisire una risorsa tale che P0 sta attendendo una risorsa già acquisita da P1, P1 sta attendendo una risorsa già acquisita da P2, …, Pn sta attendendo una risorsa già acquisita da P0.

Grafo di allocazione delle risorse

- R = {R1, R2, ..., Rm}: insieme delle risorse del sistema (memoria, dispositivi di I/O, …)
  - Ogni risorsa j ha Kj istanze
- P = {P1, …, Pn}: processi in esecuzione
- Ogni processo:
  - Richiede un'istanza di una risorsa j (es. fopen())
  - Usa quell'istanza (es. fwrite())
  - Rilcascia quella risorsa (es. fclose())
- Grafo di allocazione delle risorse (Resource Allocation Graph, RAG): **grafo orientato {V, E}** ove V : (P ∪ R), E: 2 tipi di arco:
  - Arco di richiesta Pi → Rj
  - Arco di assegnazione Rj → Pi

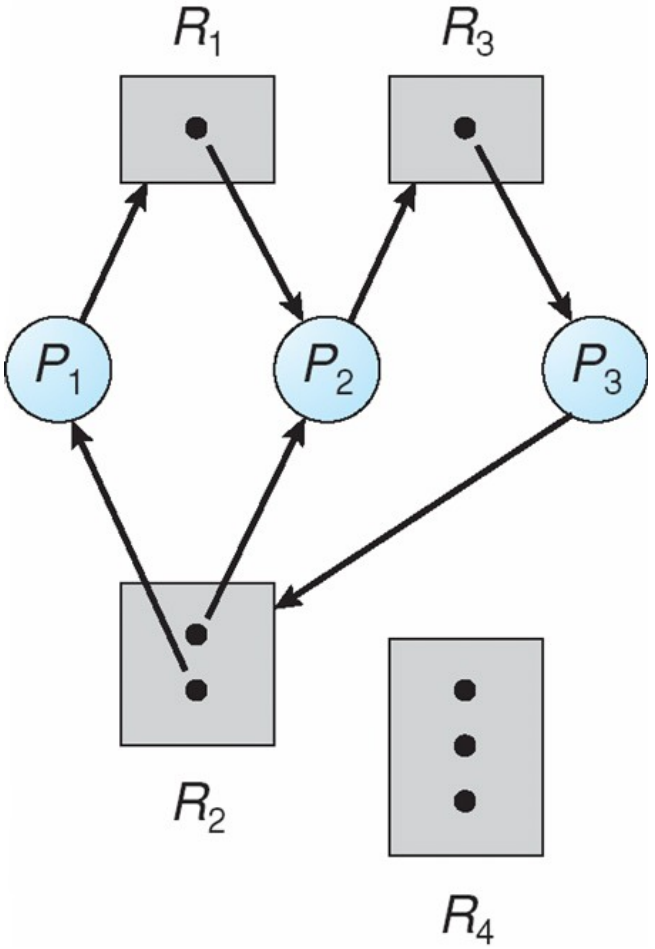
Esempio di RAG

- Un arco di richiesta può puntare solo ad una risorsa (rettangolo in figura), mentre un arco di assegnazione parte sempre da un'istanza (punti in figura)
- Quando un processo necessita di una richiesta, viene aggiunto un arco di richiesta: se la richiesta può essere soddisfatta, tale arco viene **immediatamente sostituito con un arco di assegnazione**.



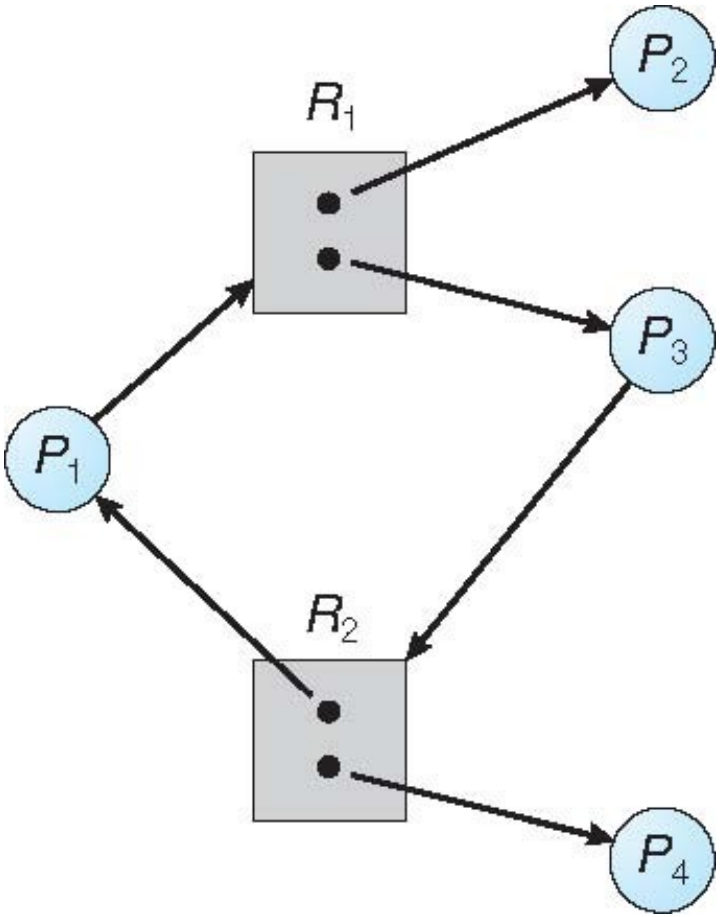
Esempio di RAG con un deadlock

- Un ciclo in un RAG è una condizione necessaria, ma non sufficiente, per l'esistenza di un deadlock.
- Nell'esempio ci sono 2 cicli, ed effettivamente c'è un situazione di deadlock tra i processi



Esempio di RAG senza deadlock

- C'è un ciclo, ma non vi è un deadlock



RAG: fatti

- Se il grafo non contiene cicli → **non c'è deadlock**
- Se il grafo contiene cicli:
  - Se ogni risorsa possiede una sola istanza → **deadlock certo**
  - Altrimenti, **deadlock possibile**
- Importante: il RAG rappresenta **istanti di esecuzione…**

Prevenzione dei deadlock (1/4)

- (1) Evitare la **mutua esclusione**: anche se alcune risorse sono condivisibile,
- **Problema (1)**
  - Eliminare la mutua esclusione è impossibile.
- (2) Evitare qualsiasi **acquisizione e attesa**:
  - Il processo è servito (eseguito) solo quando tutte le risorse di cui necessita sono disponibili → *non alloca alcuna risorsa, alloca tutto quando disponibile*
  - Il processo può allocare alcune risorse ma, se ne richiede altre, le può allocare solo a patto di aver rilasciato **tutte** le risorse precedentemente allocate

Prevenzione dei deadlock (2/4)

- **Problemi (2)**:
  - Allocazione multipla → Scarso utilizzo delle risorse, lunghe attese per combinazioni complesse (es. stampante + disco)
  - Strarvation: ovvio, se un processo attende molte risorse utilizzatissime, avrà scarsissima probabilità di ottenerle tutte contemporaneamente

Prevenzione dei deadlock (3/4)

- (3) **Abilitare il preemption delle risorse**: se un processo che occupa alcune risorse richiede un'altra risorsa correntemente occupata da un altro processo, le risorse occupate vengono forzatamente rilasciate dal sistema operativo
  - Oltre alle risorse richieste, tale processo dovrà attendere a tutte le risorse che gli sono state tolte
- **Problema (3)**: le risorse lente (es. la stampante, ma anche l'HD) determinano un overhead non trascurabile

Prevenzione dei deadlock (4/4)

- (4) **Evitare l'attesa circolare**: l'idea è di assegnare un ordine (attraverso un indice univoco $I(R_i)$) alle varie risorse R che possono essere allocate (memoria, dischi, ecc...), accertando che ogni processo richieda tali risorse in ordine crescente di enumerazione.
- Ogni risorsa può essere allocata solo in **ordine crescente**
  - Si può dimostrare per assurdo: se il processo P_n sta attendendo una risorsa occupata dal processo P_0, il quale a sua volta sta attendendo una risorsa dal processo P_1, ecc... significa che la risorsa occupata dal processo successivo nella lista avrà un indice maggiore di quella richiesta → **impossibile**
- **Problema (4)**: tecnica in cui il responsabile principale è lo sviluppatore finale

Tecniche per evitare i deadlock

- Prevenzione dei deadlock → in parte funzionano, ma compromettono l'efficienza del sistema.
- Esempi per evitare **attivamente** i deadlock:
  - Ogni processo dichiara il **numero massimo di risorse richieste** (ovvero, numero massimo di istanze per ogni tipo).
  - Il sistema operativo esamina dinamicamente lo stati di allocazione in ogni momento, evitando i deadlock.

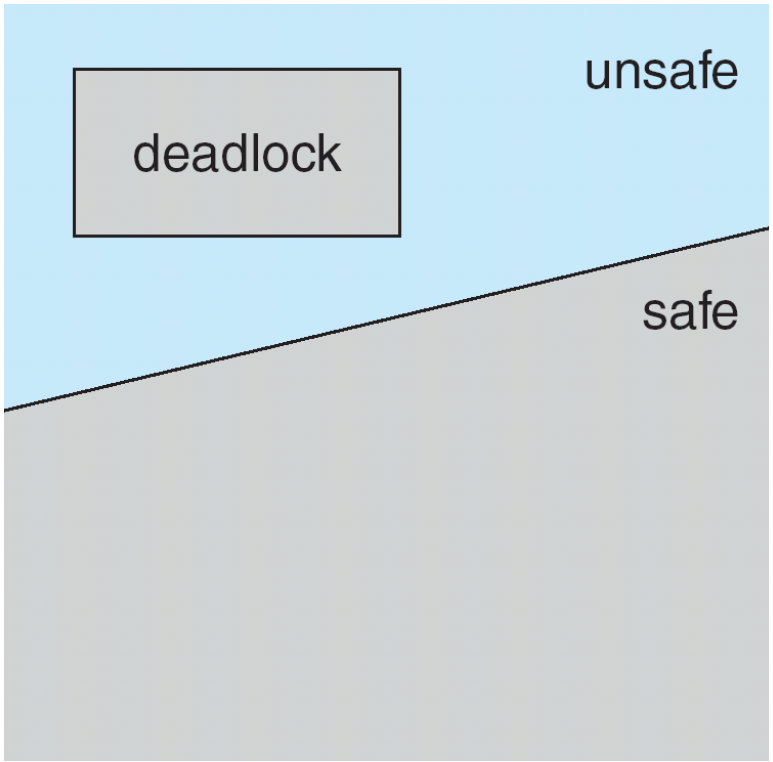
    Input del sistema operativo:
    - numero di risorse disponibili
    - numero di risorse assegnate
    - Numero massimo di risorse richieste dai processi

Safe State (1/3)

- Se un processo richiede una risorsa disponibile, il sistema deve decidere se tale allocazione lascerà il sistema in uno **stato sicuro (safe state)**.
- Il sistema è in safe state se esiste **una sequenza di processi** $<P_1, P_2, \ldots, P_n>$ tale che, per ogni processo P_i, le risorse da esso richieste (e non ancora allocate) possono essere soddisfatte con le risorse attualmente disponibili dal sistema più le risorse correntemente allocate da tutti i P_j, $j < i$
- Se le risorse non sono immediatamente disponibili, allora Pi attende al più finché tutti i Pj hanno concluso l'utilizzo delle risorse.
- **Problema**: possibile sotto-utilizzo delle risorse

Safe State (2/3)

- Il sistema è in safe state → deadlock impossibili
- Viceversa → deadlock **possibili** (non certi!)
- **Tecnica di base**: le risorse sono ogni volta assegnate solo se vi è la certezza che il sistema dopo l'allocazione rimane in safe state



Safe State (3/3)

- Esempio: vi sono 12 istanze disponibili di una data risorsa, e al tempo t_0 le risorse siano allocate come da tabella.
- La sequenza P_1, P_0, P_2 soddisfa la condizione di safe state.

Process	Max Needs	Allocated	Current Needs
P0	10	5	5
P1	4	2	2
P2	9	2	7

- Se P_2 richiede una risorsa aggiuntiva e tale richiesta viene soddisfatta → **deadlock**

Algoritmi base

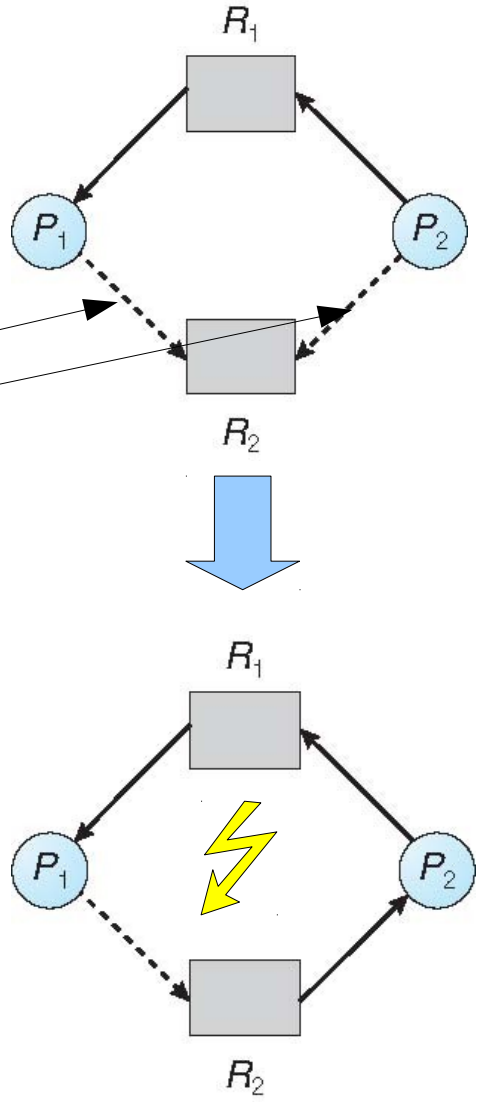
- Esiste un'unica istanza per ogni tipo di risorsa
  - →Utilizzare il grafo di allocazione delle risorse (RAG)
- Esistono più istanze per ogni tipo di risorsa
  - → Utilizzare l'**algoritmo del banchiere**

Evitare i deadlock con il RAG (1/2)

- Aggiungo alla struttura descritta una nuova tipologia di arco → **arco di reclamo (claim)**
- Un arco di reclamo $P_i \rightarrow R_j$ indica che il processo P_i può richiedere la risorsa R_j in un qualsiasi momenti futuro.
- Quando P_i **richiede** la risorsa R_j l'arco di reclamo diventa arco di richiesta
- Quando P_i **rilascia** la risorsa R_j l'arco di assegnazione $R_j \rightarrow P_i$ diventa arco di reclamo reclamo $P_i \rightarrow R_j$
- **E' necessario che le risorse vengano richieste a priori**

Evitare i deadlock con il RAG (2/2)

- Idea: **evitare di formare cicli** (considerando anche gli archi di **reclamo**)
- Se vi sono cicli siamo in **unsafe state**



Algoritmo del banchiere

- Applicabile a risorse con istanze multiple
- Per ogni risorsa, ogni processo deve dichiarare a priori il massimo numero di istanze usate
- L'allocazione è permessa solo se lo stato risultante è **safe**, altrimenti il processo dovrà attendere
- Ogni processo, ovviamente, deve rilasciare le risorse allocate in un intervallo di tempo finito

Alg. del banchiere: strutture base

- Sia n il numero dei processi e m il numero delle risorse:
  - **available[ ]**: vettore di lunghezza m, se available[j] = k, allora ci sono k istanze delle risorsa Rj disponibili
  - **max[ ][ ]**: matrice n x m, se max[i][j] = k, allora il processo Pi potrà richiedere al massimo k istanze della risorsa di tipi Rj.
  - **allocation[ ][ ]**: matrice n x m, se allocation[i][j] = k, allora il processo Pi sta correntemente utilizzando k istanze della risorsa di tipi Rj
  - **need[ ][ ]**: matrice n x m, se need[i][j] = k, allora il processo Pi potrebbe richiedere altre k istanze della risorsa di tipi Rj
- Ovviamente **need [i,j] = max[i,j] – allocation [i,j]**

Alg. del banchiere: Safety Algorithm

→ **Per verificare se un sistema è in safe-state**

1. Siano **work** and **finish** vettori di lunghezza m ed n, rispettivamente, inizializzati come segue:
    1. **work = available**
    2. **finish[i] = false** per ogni i = 0, 1, …, n- 1
2. Trova i tale che:
    1. **finish [i] == false**
    2. **need$_i$ <= work**

    Se non esiste alcun i che soddisfa queste condizioni, vai al punto 4
3. **work = work + allocation$_i$, finish[i] = true** e vai al punto 2
4. Se **finish [i] == true** per ogni i, allora il sistema si trova in un safe state

Alg. del banchiere: richiesta di risorse per il processo P_i

- → **Per verificare se le richieste di P_i possono essere soddisfatte**
- **request[ ]**: vettore delle richieste per il processo P_i. Se request[j] = k, allora P_i necessita di k istanze di tipo R_j

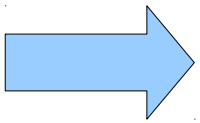
1. Se **$request_i$ <= $need_i$** vai al punto 2, altrimenti solleva un'eccezione (il processo richiede più risorse di quelle dichiarate a priori)
2. Se **$request_i$ <= available** vai al punto 3, altrimenti P_i deve attendere
3. Prova a modifica lo stato come segue:
   - **available = available - $request_i$**
   - **$allocation_i$ = available + $request_i$**
   - **$need_i$ = $need_i$ - $request_i$**

- **Se lo stato è safe → le risorse sono allocate a P_i**
- **Viceversa, le modifiche sono annullate e P_i deve attendere**

Alg. del banchiere: esempio (1/2)

- 5 processi P0,...,P4, 3 tipi di risorse:
  - A (10 ist.), B (5 ist.), C (7 ist):

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	



Con Safety Algorithm
<P1, P3, P4, P2, P0>
= Safe State

Alg. del banchiere: esempio (2/2)

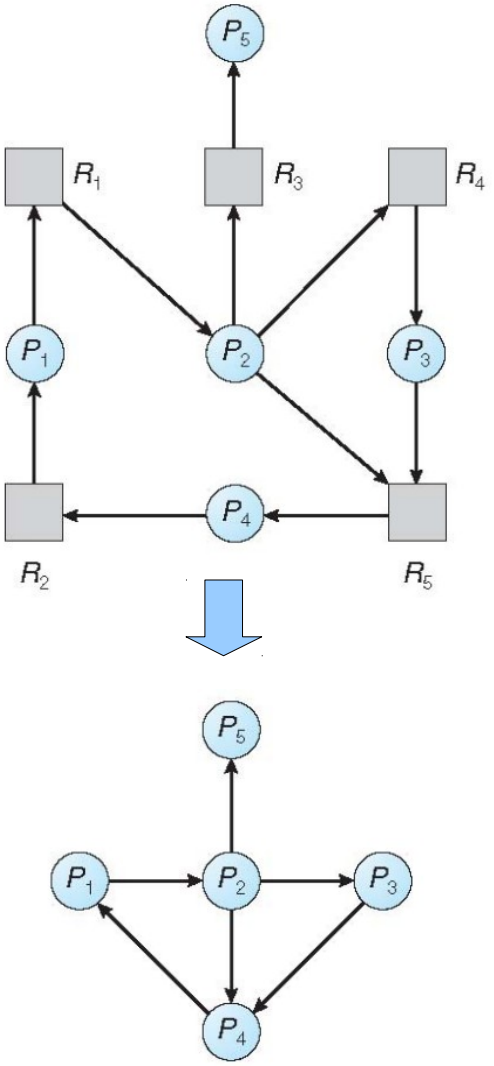
- P1 richiede (1,0,2) → request <= available ((1,0,2) <= (3,3,2) → OK!

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Eseguendo ora il Safety Algorithm → < P1, P3, P4, P0, P2> = Safe State → **vengono allocate le risorse per P1**

Rilevamento dei deadlock: wait-for graph

- Ogni risorsa ha una sola istanza → **wait-for graph** (Pi → Pj se Pi sta attedendo Pj)
- Se il wait-for graph contiene cicli → **deadlock**
- Algoritmo invocato periodicamente alla ricerca di cicli



Rilevamento dei deadlock: Istanze multiple (1/2)

- **available[ ]**: vettore di lunghezza m, se available[j] = k, allora ci sono k istanze delle risorsa Rj disponibili
- **allocation[ ][ ]**: matrice n x m, se allocation[i][j] = k, allora il processo Pi sta correntemente utilizzando k istanze della risorsa di tipi Rj
- **request[ ][ ]**: matrice n x m che indica il numero delle richieste di ogni processo, se request[i][j] = k, allora il processo Pi sta richiedendo k istanze della risorsa di tipi Rj

Rilevamento dei deadlock: Istanze multiple (2/2)

- Siano **work** and **finish** vettori di lunghezza m ed n, rispettivamente, inizializzati come segue:
  - **work = available**
  - Se allocation$_i$!= 0, **finish[i] = false** altrimenti **finish[i] = true** (per ogni i = 0, 1, …, n- 1)
- Trova i tale che (se non esiste, vai a punto 4):
  - **finish [i] == false**
  - **request$_i$ <= work**
- **work = work + allocation$_i$, finish[i] = true** e vai al punto 2
- Se **finish [i] == false** per qualche i, allora il sistema si trova in uno stato di **deadlock**

Rilevamento dei deadlock: esempio

- 5 processi P0,...,P4, 3 tipi di risorse:
  - A (7 ist.), B (2 ist.), C (6 ist):

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequenza <P0,P2,P3,P1,P4> → finish[*] == true

Contromisure ai deadlock

- Terminazione di tutti i processi in deadlock, oppure:
- Terminazione di un processo alla volta fino all'eliminazione del deadlock →
  In quale ordine?
  - Priorità
  - Tempo CPU
  - …
- Preemption della risorsa