

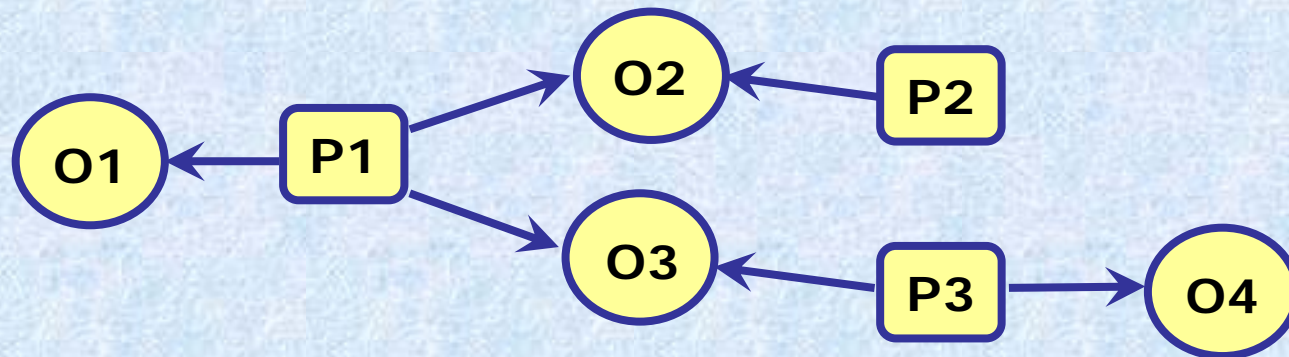
Sincronizzazione dei processi

Ambiente globale/locale

Il sistema è visto come un insieme di processi e oggetti (dati del processo, strutture dati che rappresentano le risorse)

Dal punto di vista della proprietà delle risorse di memoria, sono possibili i due modelli ad *ambiente globale* e ad *ambiente locale*

Modello ad ambiente globale

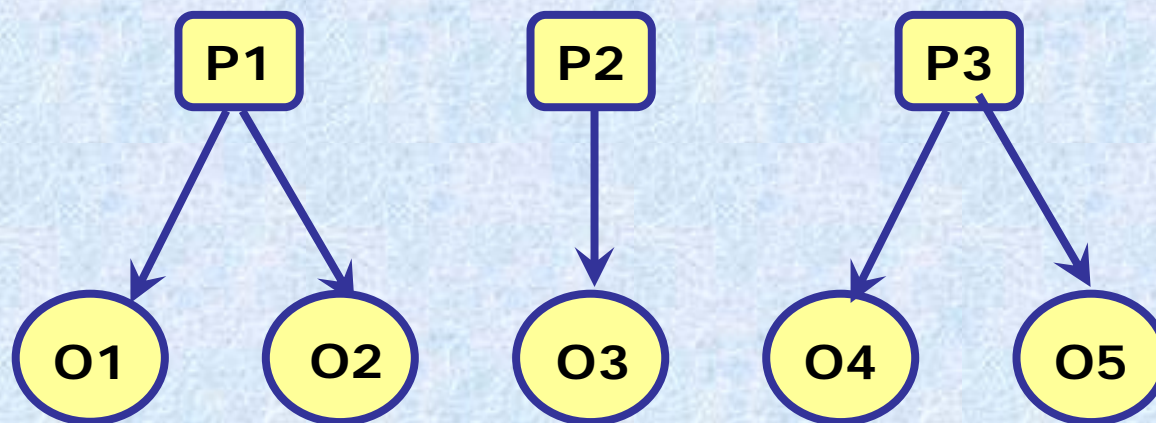


O1, O4 oggetti privati

O2, O3 oggetti comuni

► *Competizione, cooperazione*

Modello ad ambiente locale

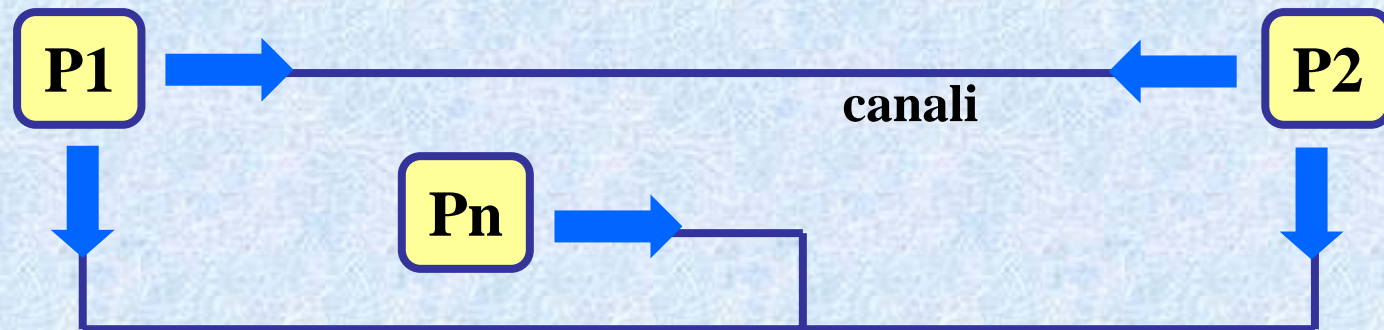


O1, O2, O3, O4, O5 risorse private

► *Competizione, cooperazione attraverso i processi serventi*

Ambiente locale

- Ambiente locale (sistemi distribuiti, ma anche sistemi monoprocesso)
- L'ambiente di un processo non è accessibile direttamente da nessun altro processo
 - Ogni forma di interazione tra processi (comunicazione, sincronizzazione) avviene tramite lo scambio di messaggi



Interazione fra processi

- **Processi concorrenti**
 - Insieme di processi la cui esecuzione si sovrappone nel tempo.
 - Più in generale: due processi si dicono concorrenti se la prima operazione di uno comincia prima dell'ultima dell'altro

Processi Concorrenti

- **Processi indipendenti:**

- due processi P1 e P2 sono indipendenti se l'esecuzione di P1 non è influenzata da P2, e viceversa
 - Proprietà della riproducibilità

- **Processi interagenti:**

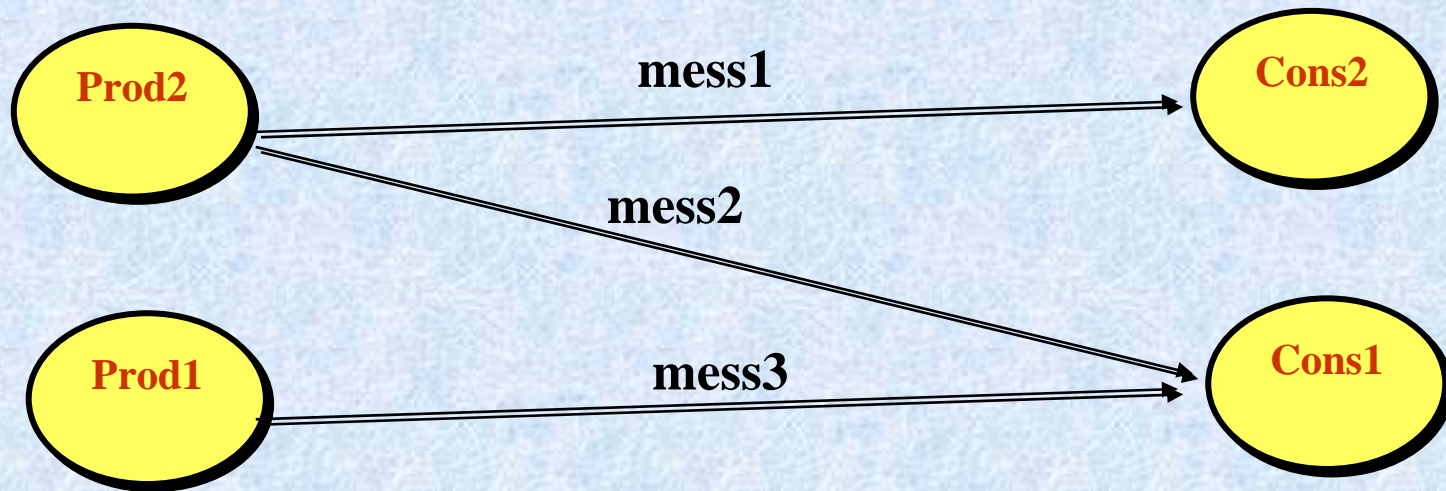
- due processi P1 e P2 sono interagenti se l'esecuzione di P1 è influenzata da P2, e viceversa
 - Effetto dell'interazione dipende dalla velocità relativa dei processi.
 - Comportamento non riproducibile

Processi Interagenti



Ambiente globale

Processi Interagenti



Ambiente locale

Tipi di interazione

- **Competizione:** per l'uso di risorse comuni che non possono essere utilizzate contemporaneamente (mutua esclusione)
- **Cooperazione:** nell'eseguire un'attività comune mediante scambio di informazioni (marca temporale, dati)
esempio: scambio di messaggi attraverso un buffer

Sincronizzazione

In tutti i casi esaminati, per un corretto funzionamento, è necessario imporre dei **vincoli** nella esecuzione delle operazioni dei processi.

Vincoli

- **Competizione:** un solo processo alla volta deve avere accesso alla risorsa comune (sincronizzazione indiretta o implicita)
- **Cooperazione:** le operazioni dei processi cooperanti (esempio: quelle con le quali produttore e consumatore operano sul buffer) devono seguire una sequenza prefissata
=>> **sincronizzazione diretta o esplicita**

Strumenti di sincronizzazione

Modello ad ambiente globale:

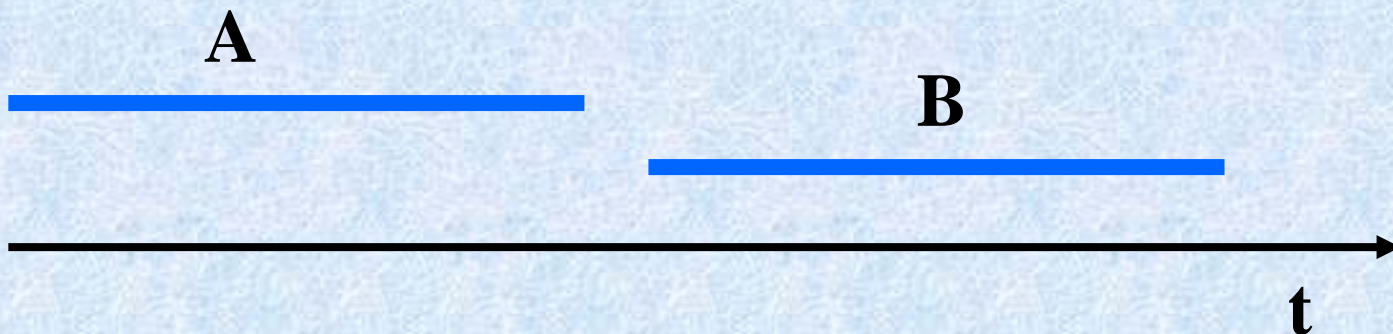
- *semafori e primitive wait e signal*

Modello ad ambiente locale:

- *messaggi e primitive send e receive*

Strumenti di sincronizzazione

- Problema della mutua esclusione: le operazioni con le quali i processi accedono alla risorsa comune (*sezioni critiche*) devono escludersi mutuamente nel tempo
 - Nessun vincolo sull'ordine con il quale le sezioni critiche vengono eseguite
- **Esempio:** A, B sezioni critiche



Problema della Mutua Esclusione

P_1 e P_2 interagiscono attraverso una pila condivisa

P_1

.....

Top ++
Stack[Top]=y

.....

P_2

.....

E=Stack[Top]
Top --

.....

 Sezioni critiche della stessa classe

Soluzione al problema della mutua esclusione

occupato=1 → risorsa occupata

occupato=0 → risorsa libera

P_1

```
prologo: while (occupato==1);  
           occupato=1;  
           <sezione critica A>  
epilogo: occupato=0;
```

P_2

```
prologo: while (occupato==1);  
           occupato=1;  
           <sezione critica B>  
epilogo: occupato=0;
```

A,B: Sezioni critiche della stessa classe

La precedente soluzione è errata
Esempio di esecuzione errata
(errore dipendente dal tempo):

T0: P1 esegue while e trova occupato=0

T1: P2 esegue while e trova occupato=0

T2: P1 pone occupato=1 ed entra in A

T3: P2 pone occupato=1 ed entra in B

→ Entrambi i processi sono contemporaneamente nella loro sezione critica

Istruzione TSL

(Test and Set Lock)

- **TSL R, x:**
 - Il valore contenuto in x viene copiato nel registro R del processore e viene scritto in x un valore diverso da zero.
 - Le operazioni di lettura e scrittura sono eseguite in modo indivisibile (hardware)
- Lettura e modifica di una parola in un solo ciclo di memoria
 - P1 può leggere e modificare la variabile occupato senza interferenze da parte di P2

lock e unlock

lock(x) :

MOV registro, 1

loop TSL registro, x

(copia x nel registro e pone x=1)

CMP registro, 0

(il valore di x era 0 ?)

JNE loop

(se no, ricomincia il ciclo)

RET

(ritorna al chiamante)

unlock(x) :

MOV x, 0

(inserisce 0 in x)

RET

(ritorna al chiamante)

Soluzione al problema della mutua esclusione con lock e unlock

P₁

prologo: `lock(x);`
 `<sez. critica A>;`
epilogo: `unlock(x);`

P₂

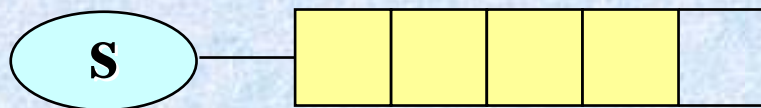
prologo: `lock(x);`
 `<sez. critica B>;`
epilogo: `unlock(x);`

Caratteristiche della soluzione

- **Attesa attiva (busy form of waiting)**
- **Applicabilità:**
 - solo a multiprocessori
 - solo sezioni critiche brevi

Semafori

- Semaforo s:
 - Variabile alla quale è associato un intero non negativo, (**s.value**), con valore iniziale $s_0 \geq 0$
 - Al semaforo è associata una lista di attesa (**s.queue**), nella quale sono inseriti i descrittori dei processi che attendono l'autorizzazione a procedere



Semafori

- Sul semaforo sono ammesse solo due operazioni (primitive) utilizzabili col meccanismo della system call:
 - wait (s)
 - signal (s)

wait

- ```
void wait (s)
{
 if (s.value==0)
 <il processo viene sospeso e il suo
 descrittore inserito in s.queue; lo
 scheduler riassegna il processore>;
 else
 s.value= s.value-1;
}
```

# signal

```
• void signal (s)
 { if (<esiste almeno un processo in
 s.queue>)
 <il descrittore del primo di questi
 viene estratto da s.queue e il suo
 stato modificato in pronto; può
 intervenire lo scheduler per
 riassegnare il processore>;
 else
 s.value = s.value + 1;
 }
```



# Soluzione al problema della mutua esclusione con semaforo

- Semaforo **mutex** associato alla risorsa condivisa
- (valore iniziale: **mutex=1**)

```
prologo: wait(mutex);
 <sez. critica A>;
epilogo: signal(mutex);
```

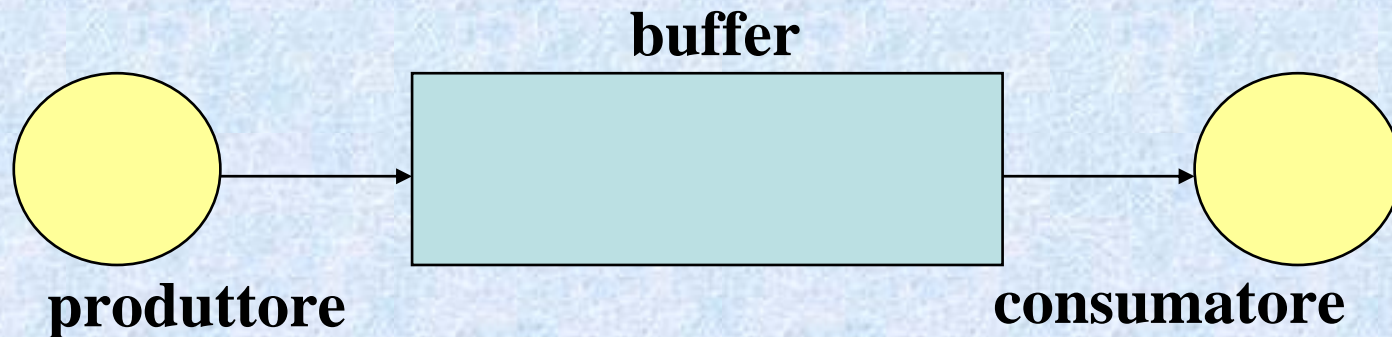
# Semafori

- wait e signal devono essere azioni indivisibili (*azioni atomiche*):
  - Modifica del valore del semaforo ed eventuale sospensione o riattivazione di un processo devono avvenire in modo indivisibile
- Indivisibilità di wait e signal:
  - processi eseguiti sullo stesso processore: disabilitazione delle interruzioni durante la loro esecuzione (automatico grazie al meccanismo della system call).
  - processi eseguiti su processori diversi: utilizzo di lock(x), unlock(x).

# Indivisibilità di wait e signal nel multiprocessore

```
lock(x);
wait(mutex);
unlock(x);
 <sezione critica>;
lock(x);
signal(mutex);
unlock(x);
```

# Sincronizzazione di processi cooperanti: problema produttore - consumatore





# Problema produttore - consumatore

- **Due semafori:**
  - spazio\_disponibile (v.i. 1)
  - messaggio\_disponibile (v.i. 0)

- **Processo Produttore:**

```
produttore{
do {
 <produzione del nuovo messaggio>;
 wait (spazio_disponibile);
 <deposito del messaggio nel buffer>;
 signal (messaggio_disponibile)
} while (!fine);
}
```

# Problema produttore - consumatore

- Processo Consumatore:

```
consumatore{
do {
 wait (messaggio_disponibile);
 <prelievo del messaggio dal buffer>;
 signal (spazio_disponibile);
 <consumo messaggio>;
} while (!fine);
}
```

# Problema produttore - consumatore

## Mutua esclusione sul buffer

Esempio: più produttori, più consumatori, vettore circolare

Produttore

```
{
 do {
 <produzione del messaggio>
 wait (spazio_disponibile);
 wait(MutexTesta);
 testa=(testa+1)%N;
 <deposito in vettore(testa)>
 signal(MutexTesta);
 signal (messaggio_disponibile);
 }
 while (!fine);
}
```

# Problema produttore - consumatore

## Mutua esclusione sul buffer

Esempio: più produttori, più consumatori, vettore circolare

Consumatore

```
{
 do {
 wait (messaggio_disponibile);
 wait(MutexCoda);
 coda=(coda+1)%N
 <prelievo da vettore(coda)>
 signal(MutexCoda);
 signal (spazio_disponibile);
 <consumo del messaggio>
 }
 while (!fine);
}
```



# Scambio di messaggi

## Paradigma tipico dei sistemi ad ambiente locale

### Meccanismo del sistema operativo:

- Canale di comunicazione
- Primitive **send** e **receive**

*send(destinazione, messaggio)*

*receive(origine, messaggio)*

### Designazione di destinazione e origine

# Scambio di messaggi

## Formato del messaggio

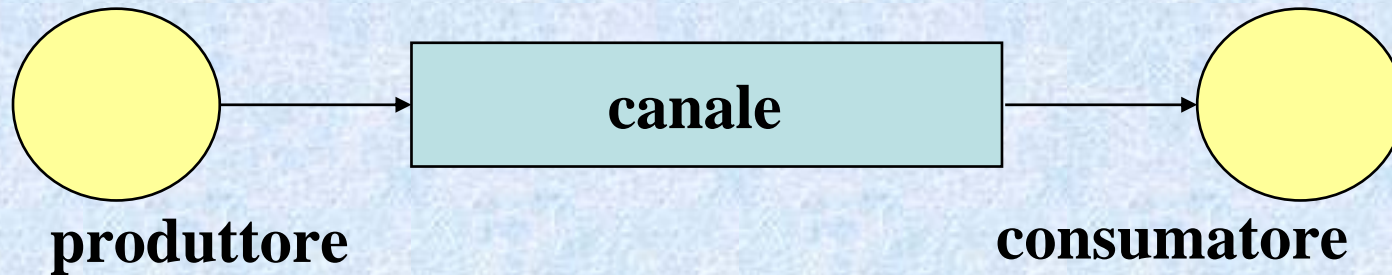
|                |
|----------------|
| origine        |
| destinazione   |
| tipo           |
| lunghezza      |
| Info controllo |
| MESSAGGIO      |

# Scambio di messaggi

## Designazione di destinazione e origine

- diretta
  - *send(dest, messaggio); receive (mitt, &messaggio)*
- indiretta
  - *send(mailbox, messaggio); receive (mailbox, &messaggio)*
  - *mailbox o porta*
- ricezione senza designazione del mittente
  - *receive (&id, &messaggio)*
  - *tipica dei processi serventi (collegamenti “client-server”)*

# Scambio di messaggi con primitive di comunicazione



Processo produttore

```
{
 do{
 produzione(&mess);
 send(consumatore, mess);
 }
 while(!fine);
}
```

Processo consumatore

```
{
 do{
 receive(produttore, &mess);
 consuma(mess);
 }
 while(!fine);
}
```



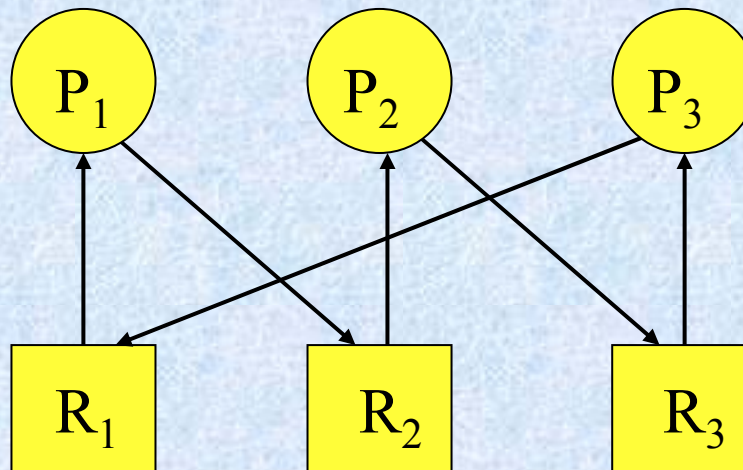
# Scambio di messaggi

## Sincronizzazione tra processi comunicanti

- *send* sincrona
- *send* asincrona
  - =>> capacità del canale
- *chiamata di procedura remota*
- *receive* bloccante
- *receive* non bloccante
- Protocollo *rendez-vous* esteso

# Stallo (blocco critico; deadlock)

Sospensione irreversibile di **processi**  
in competizione per un insieme di **risorse**



# Stallo - Ipotesi

Si considerano **risorse riusabili**

esempio: archivi

- **richiesta**

esempio: open-file

**ipotesi: richiesta bloccante**

- **assegnazione e utilizzo**

- **rilascio**

esempio: close-file

# Stallo

## Condizioni per il verificarsi dello stallo

- **Mutua esclusione**

risorse utilizzabili da un processo alla volta

- **Possesso e attesa**

i processi che si sospendono trattengono le risorse assegnate

- **Assegnazione non revocabile**

risorse esplicitamente rilasciate dai processi

- **Attesa circolare**



# Stallo

## Attesa circolare

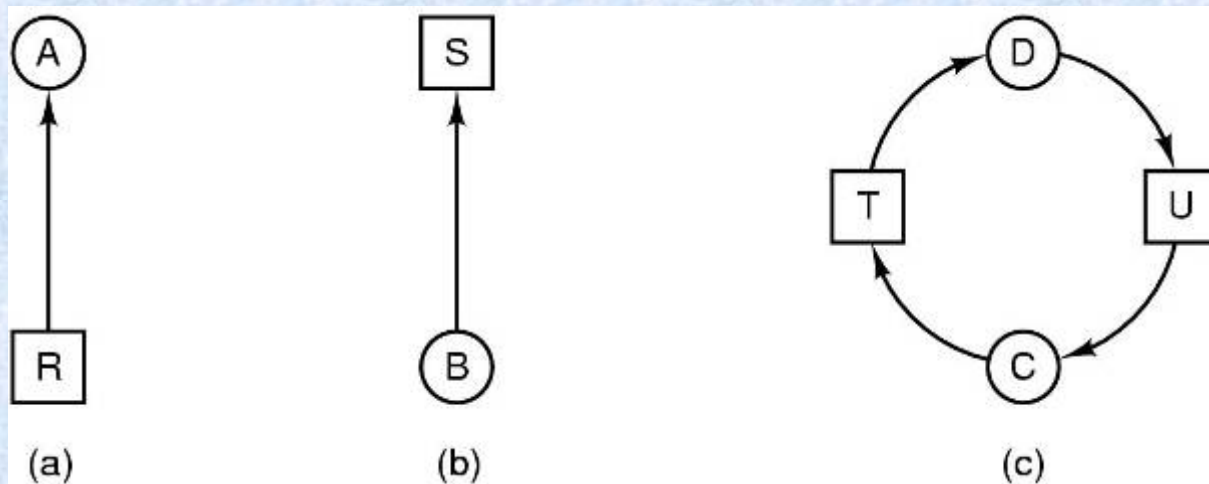
Esiste un insieme  $\mathcal{P}$  di processi e un insieme  $\mathcal{R}$  di risorse tali che:

- ogni risorsa di  $\mathcal{R}$  è assegnata a un processo di  $\mathcal{P}$
- ogni processo di  $\mathcal{P}$  è in attesa di una risorsa di  $\mathcal{R}$

--> il verificarsi dell'attesa circolare dipende dall'ordine con cui i processi eseguono richieste e rilasci

# Modelli per lo Stallo

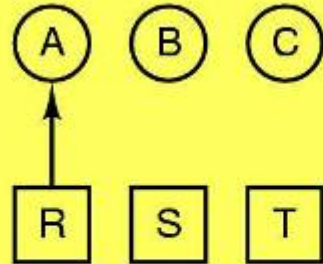
- Grafi di allocazione delle risorse



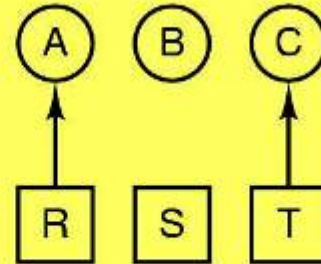
- La risorsa R è assegnata al processo A
- Il processo B richiede/aspetta la risorsa S
- I processi C e D sono attesa circolare

### Sequenza di richieste e rilasci che non provoca stallo

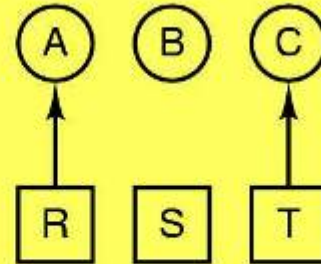
1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S  
no deadlock



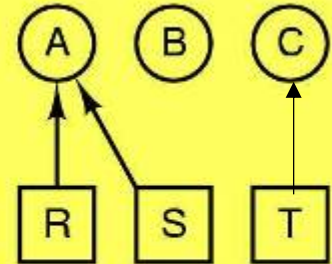
(k)



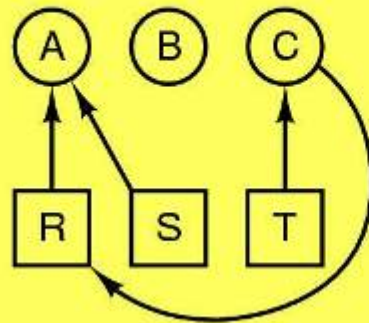
(l)



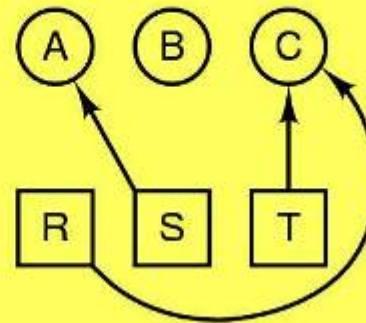
(m)



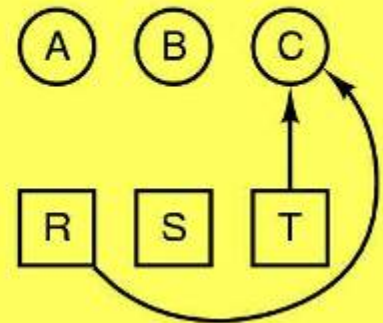
(n)



(o)



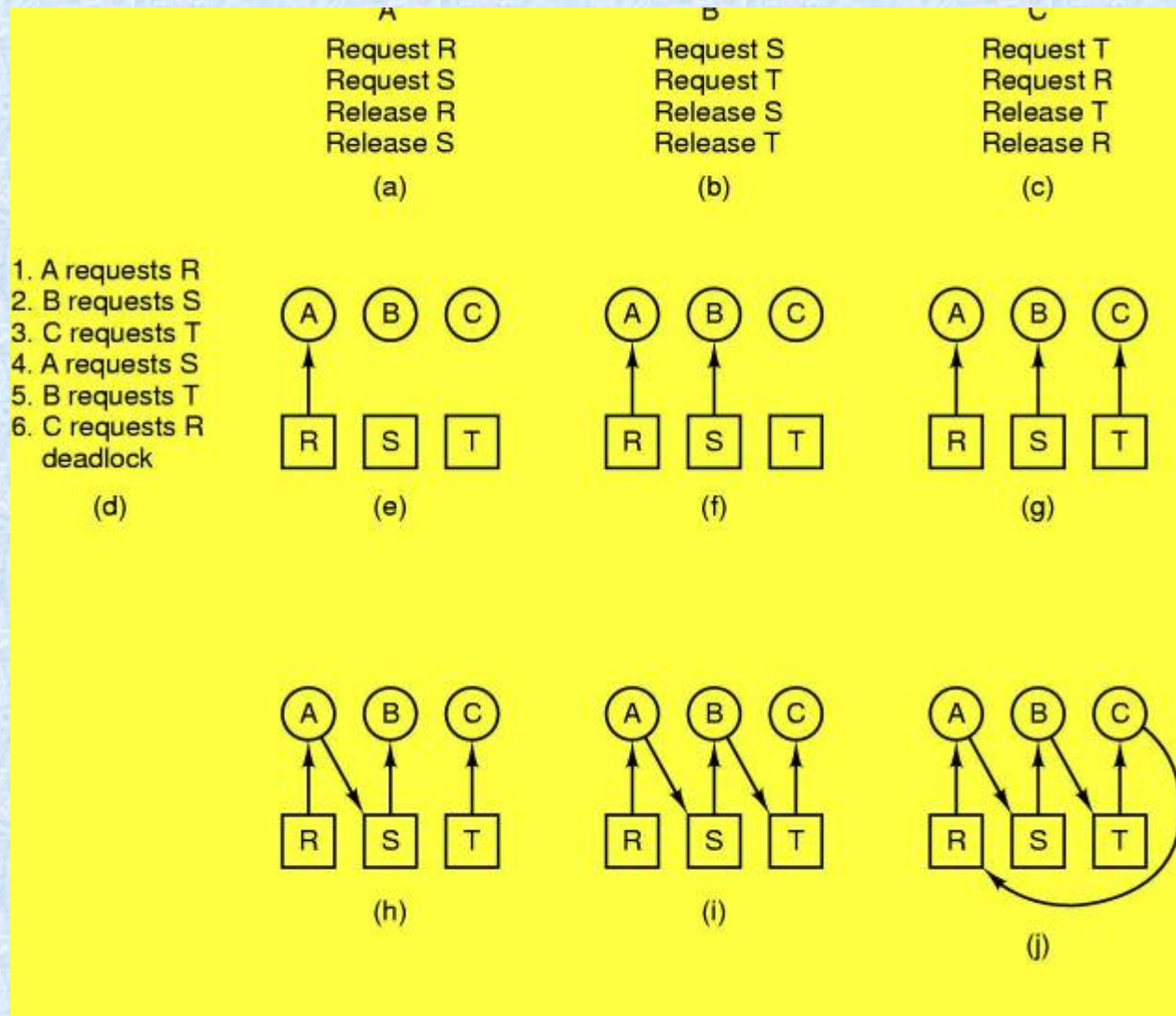
(p)



(q)



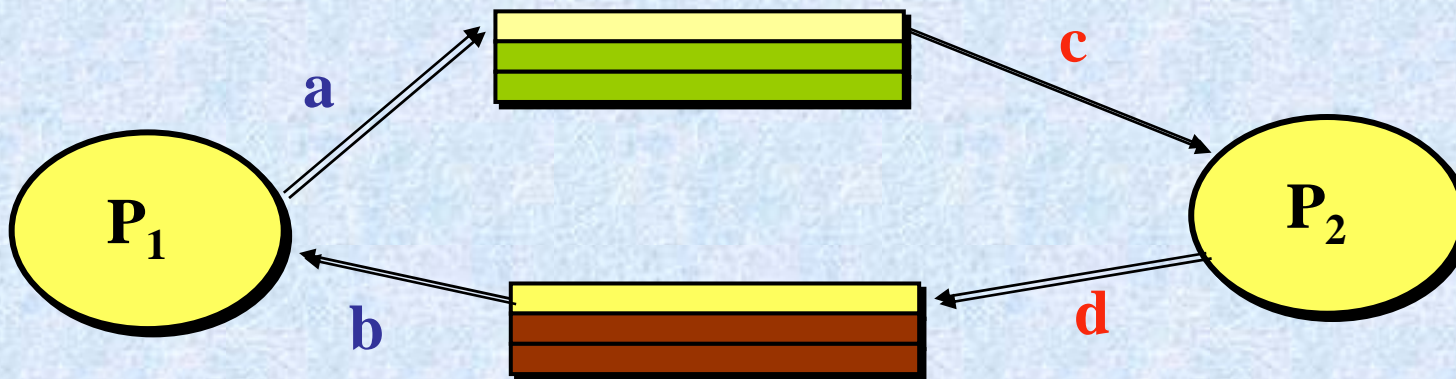
# Sequenza di richieste e rilasci che provoca stallo





## Esempio con risorse di più tipi

- Scambio di messaggi attraverso due pile
- $P_1$  produttore,  $P_2$  consumatore
  - a **a** d **d** c **b** provoca lo stallo
  - a d b **a** c d non provoca lo stallo



## Metodi per il trattamento dello stallo

- Prevenzione statica
- Prevenzione dinamica
- Riconoscimento ed eliminazione
- In realtà molti sistemi (Unix, Windows) non affrontano il problema
  - Ragionevole se:
    - le risorse sono abbondanti e lo stallo si verifica raramente
    - Il costo per evitare lo stallo è troppo elevato

# Stallo

## Prevenzione statica

La prevenzione statica agisce sulle condizioni che portano allo stallo:

- **Mutua esclusione**
  - **Tramite spool**
- **Possesso e attesa**
  - **Imponendo vincoli sulle modalità di richiesta**
- **Attesa circolare**
  - **Tramite ordinamento delle risorse**

## Prevenzione statica sulla condizione di mutua esclusione

- Alcuni dispositivi (ad esempio le stampanti) possono essere gestiti con spool
  - I processi scrivono l'output in un'area di spool
  - Solo il gestore della stampante richiede e usa la stampante
  - Quindi lo stallo per la stampante è eliminabile
- Non tutti i dispositivi possono essere gestiti con spool
- Ci può essere stallo nell'accesso all'area di spool
  - buffer: risorsa riusabile multipla
- Principio:
  - Evitare di assegnare una risorsa quando non strettamente necessario
  - Far in modo che il minor numero possibile di processi possa richiedere una risorsa

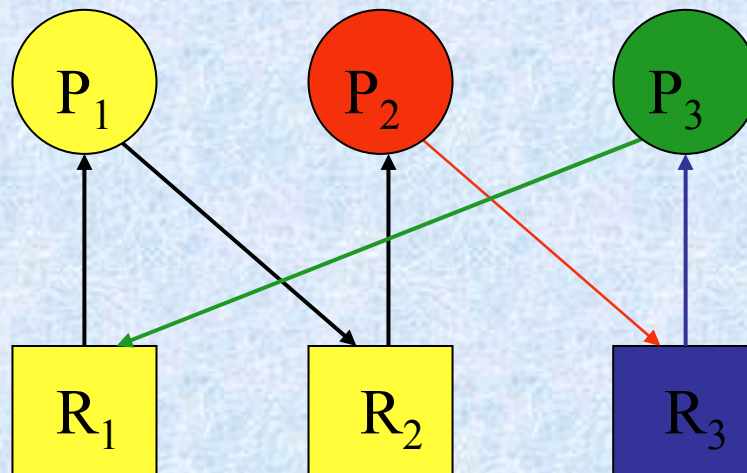


## Prevenzione statica sulla condizione di possesso e attesa

- il processo conosce in anticipo tutte le risorse che utilizzerà
  - le richiede prima dell'utilizzo con un'unica richiesta
  - --> il processo che possiede risorse non attende per assegnazione
- Problemi
  - Può non sapere di quali risorse avrà bisogno
  - Vincola risorse che altri processi potrebbero usare
- Variazioni:
  - Un processo deve prima rilasciare tutte le risorse...
  - ... e quindi richiedere immediatamente tutte quelle necessarie

## Prevenzione statica sulla condizione di attesa circolare

- Ordinamento delle risorse
  - richieste ordinate secondo l'ordinamento delle risorse



Processo che causa attesa circolare

Risorsa di indice massimo

Processo che ha violato il vincolo di richiesta ordinata

## Prevenzione statica

| Condizione        | Approccio                                |
|-------------------|------------------------------------------|
| Mutua esclusione  | Usare spool                              |
| Possesso e attesa | Richiedere tutte le risorse inizialmente |
| Attesa circolare  | Ordinare le risorse                      |
|                   |                                          |

## Sommario degli approcci per la prevenzione statica

# Stallo

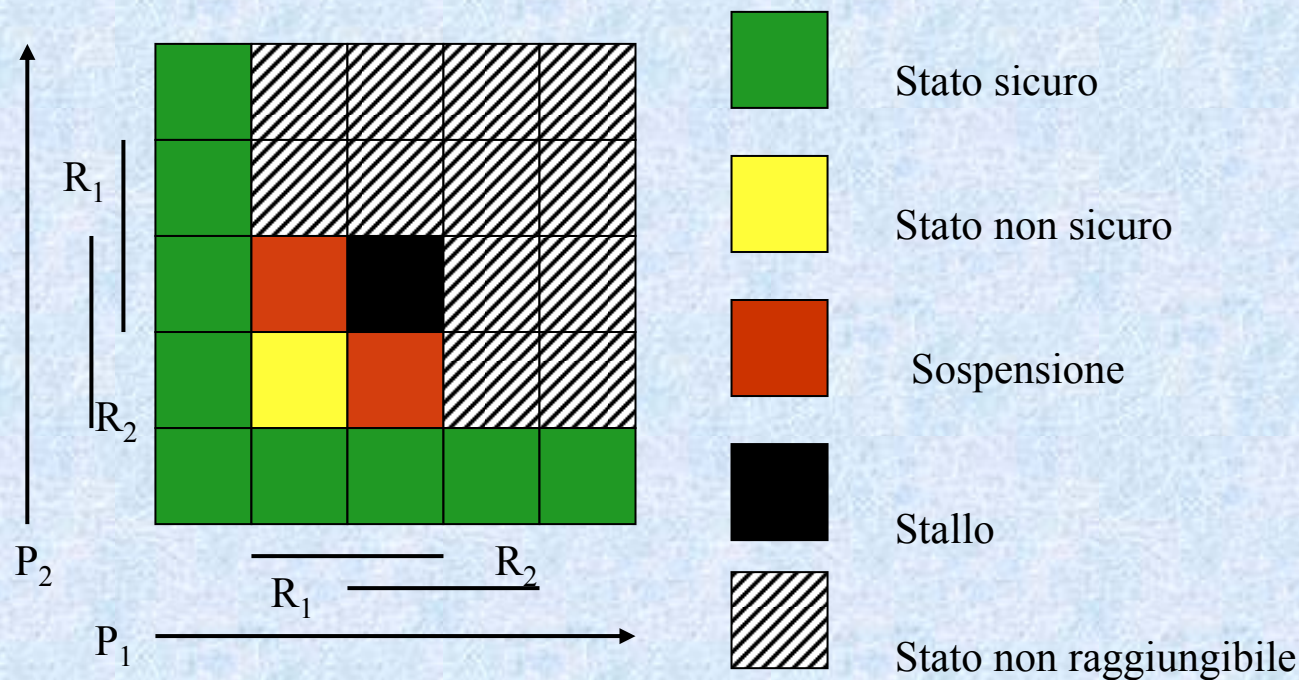
## Prevenzione dinamica

- banchiere: gestore delle risorse
- politica: algoritmo del banchiere
  - processo P richiede risorsa R
  - banchiere assegna risorsa solo se assegnazione conduce in uno **stato sicuro**
- Stato S sicuro se a partire da S esiste una sequenza di assegnazioni e rilasci che permetta di soddisfare tutte le richieste



# Algoritmo del banchiere

Evoluzione del sistema per stati sicuri:



## Stati sicuri e insicuri

### Risorsa multipla di un solo tipo (10 istanze)

| Has Max |   |   | Has Max |   |   | Has Max |   |   | Has Max |   |   | Has Max |   |   |
|---------|---|---|---------|---|---|---------|---|---|---------|---|---|---------|---|---|
| A       | 3 | 9 | A       | 3 | 9 | A       | 3 | 9 | A       | 3 | 9 | A       | 3 | 9 |
| B       | 2 | 4 | B       | 4 | 4 | B       | 0 | — | B       | 0 | — | B       | 0 | — |
| C       | 2 | 7 | C       | 2 | 7 | C       | 2 | 7 | C       | 7 | 7 | C       | 0 | — |
| Free: 3 |   |   | Free: 1 |   |   | Free: 5 |   |   | Free: 0 |   |   | Free: 7 |   |   |
| (a)     |   |   | (b)     |   |   | (c)     |   |   | (d)     |   |   | (e)     |   |   |

Dimostrazione che lo stato (a) è sicuro

- Has : numero di risorse acquisite dal processo
- Max: massimo numero di richieste di risorse da parte del processo

## Stati sicuri e insicuri

### Risorsa multipla di un solo tipo (10 istanze)

| Has Max |   |   | Has Max |   |   | Has Max |   |   | Has Max |   |   |
|---------|---|---|---------|---|---|---------|---|---|---------|---|---|
| A       | 3 | 9 | A       | 4 | 9 | A       | 4 | 9 | A       | 4 | 9 |
| B       | 2 | 4 | B       | 2 | 4 | B       | 4 | 4 | B       | — | — |
| C       | 2 | 7 | C       | 2 | 7 | C       | 2 | 7 | C       | 2 | 7 |
| Free: 3 |   |   | Free: 2 |   |   | Free: 0 |   |   | Free: 4 |   |   |
| (a)     |   |   | (b)     |   |   | (c)     |   |   | (d)     |   |   |

Dimostrazione che lo stato (b) non è sicuro

- Has : numero di risorse acquisite dal processo
- Max: massimo numero di richieste di risorse da parte del processo

## L'algoritmo del banchiere per un unico tipo di risorse multiple

| Has Max  |   |   |
|----------|---|---|
| A        | 0 | 6 |
| B        | 0 | 5 |
| C        | 0 | 4 |
| D        | 0 | 7 |
| Free: 10 |   |   |

(a)

| Has Max |   |   |
|---------|---|---|
| A       | 1 | 6 |
| B       | 1 | 5 |
| C       | 2 | 4 |
| D       | 4 | 7 |
| Free: 2 |   |   |

(b)

| Has Max |   |   |
|---------|---|---|
| A       | 1 | 6 |
| B       | 2 | 5 |
| C       | 2 | 4 |
| D       | 4 | 7 |
| Free: 1 |   |   |

(c)

- Tre stati di allocazione delle risorse
  - (a) sicuro
  - (b) sicuro
  - (c) insicuro



# L'algoritmo del banchiere per risorsa singola

- Prima di iniziare l'esecuzione ogni processo dichiara il massimo numero di risorse che gli sono necessarie
- Ad ogni richiesta di una nuova risorsa l'algoritmo controlla se accogliere la richiesta porta ad uno stato sicuro o insicuro
  - per ogni processo si calcolano le unità di risorsa ancora richiedibili ( $R = \text{Max} - \text{Has}$ )
  - si considerano i processi in ordine di  $R$  crescente controllando che ognuno possa ancora richiedere  $R$  risorse e terminare correttamente
  - se tutti i processi possono terminare correttamente lo stato è sicuro
- Solo le richieste che portano a stati sicuri sono accolte

# Algoritmo del banchiere per risorse multiple

|       | Process            | Tape drives | Plotters | Scanners | CD ROMs |
|-------|--------------------|-------------|----------|----------|---------|
| $C =$ | A                  | 3           | 0        | 1        | 1       |
|       | B                  | 0           | 1        | 0        | 0       |
|       | C                  | 1           | 1        | 1        | 0       |
|       | D                  | 1           | 1        | 0        | 1       |
|       | E                  | 0           | 0        | 0        | 0       |
|       | Resources assigned |             |          |          |         |

|       | Process                | Tape drives | Plotters | Scanners | CD ROMs |
|-------|------------------------|-------------|----------|----------|---------|
| $R =$ | A                      | 1           | 1        | 0        | 0       |
|       | B                      | 0           | 1        | 1        | 2       |
|       | C                      | 3           | 1        | 0        | 0       |
|       | D                      | 0           | 0        | 1        | 0       |
|       | E                      | 2           | 1        | 1        | 0       |
|       | Resources still needed |             |          |          |         |

$E = (6342)$   
 $P = (5322)$   
 $A = (1020)$

- $E = \langle e_0, \dots, e_h \rangle$ ;  $e_i$  = numero di risorse di classe  $i$
- $P = \langle p_0, \dots, p_h \rangle$ ;  $p_i$  = numero di risorse di classe  $i$  occupate
- $A = \langle a_0, \dots, a_h \rangle$ ;  $a_i$  = numero di risorse di classe  $i$  disponibili ( $A = E - P$ )

# Algoritmo del banchiere per risorse multiple

Inizialmente ogni processo  $P_j$  è non marcato

while ( $\exists$  processi non marcati) {

if ( $\exists$  una riga  $R_j$  di  $R$  tale che  $R_j \leq A$ ) {

    Marca il processo  $P_j$ ;

$A = A + C_j$ ;

} else termina: lo stato non è sicuro;

}

Termina con successo:

non ci sono processi in stallo



# Stallo

## Riconoscimento ed eliminazione

- **Riconoscimento dei processi in attesa circolare**
  - Tramite algoritmo del banchiere o tramite grafo di allocazione delle risorse
- **Soppressione di uno o più processi in attesa circolare**
  - > recupero delle risorse
- **Prerilascio forzato di alcune risorse**
- **Check-pointing e roll-back**



# Eliminare lo stallo

- **Tramite prerilascio**
  - forzare il prerilascio di una risorsa da parte di uno dei processi in stallo
  - dipende dalla natura della risorsa
- **Tramite “rollback”**
  - Salvataggio periodico dello stato dei processi
  - In caso di stallo:
    - Si ripristina il processo all’ultimo stato salvato
    - Si fa ripartire il processo

# Eliminare lo stallo

- **Tramite terminazione dei processi**
  - È il modo più semplice e drastico per eliminare lo stallo
  - Si forza la terminazione di uno dei processi nel ciclo dello stallo
  - Gli altri processi acquisiscono le risorse del processo terminato
  - Si sceglie un processo che può essere fatto ripartire dall'inizio