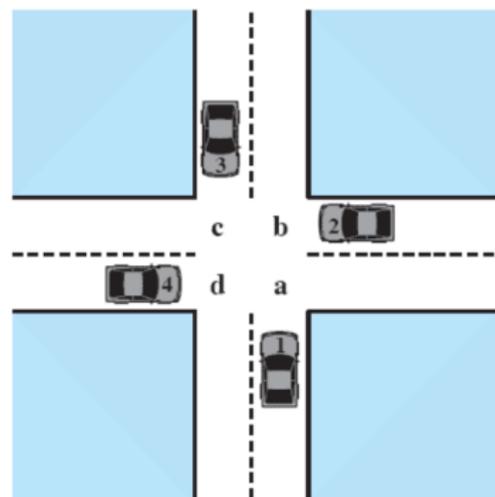


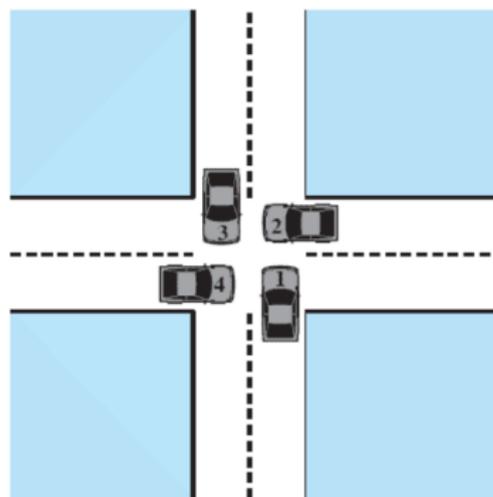




# Deadlock

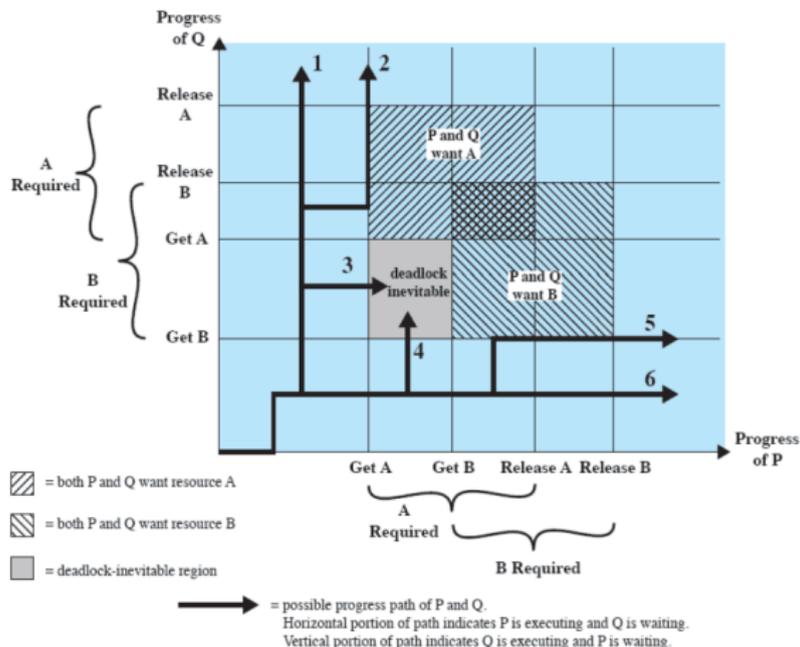


(a) Deadlock possible

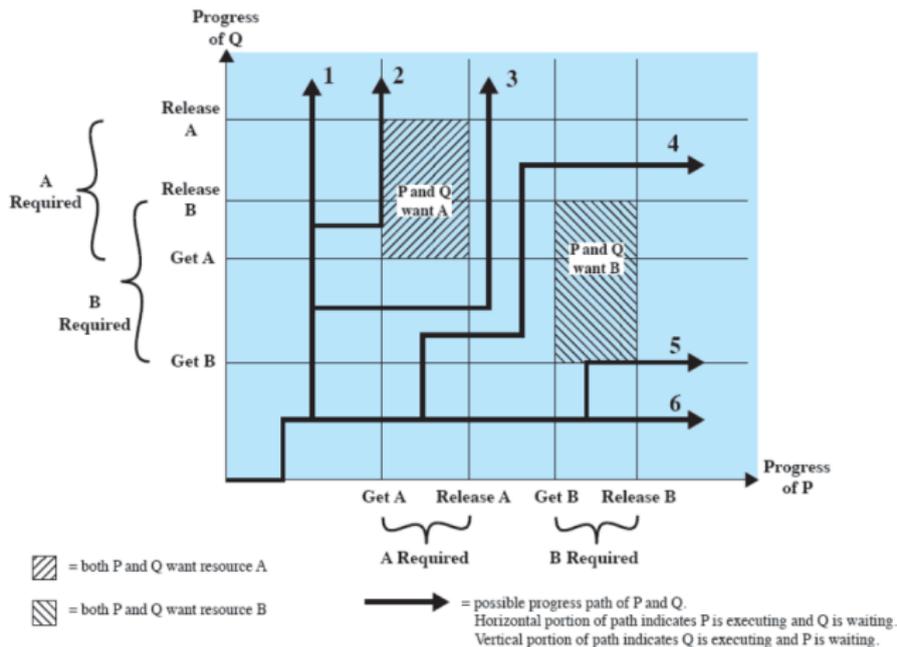


(b) Deadlock

# Deadlock: Joint Progress Diagram



# Deadlock: Joint Progress Diagram





# Risorse Riutilizzabili: Esempio 1 con 2 Processi

## Process P

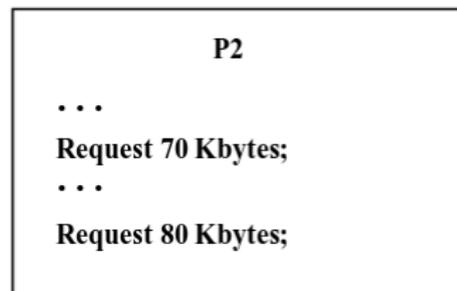
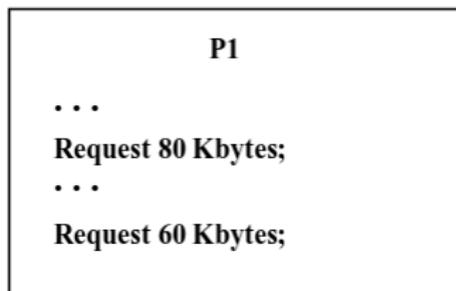
Step	Action
p <sub>0</sub>	Request (D)
p <sub>1</sub>	Lock (D)
p <sub>2</sub>	Request (T)
p <sub>3</sub>	Lock (T)
p <sub>4</sub>	Perform function
p <sub>5</sub>	Unlock (D)
p <sub>6</sub>	Unlock (T)

## Process Q

Step	Action
q <sub>0</sub>	Request (T)
q <sub>1</sub>	Lock (T)
q <sub>2</sub>	Request (D)
q <sub>3</sub>	Lock (D)
q <sub>4</sub>	Perform function
q <sub>5</sub>	Unlock (T)
q <sub>6</sub>	Unlock (D)

## Risorse Riutilizzabili: Esempio 2 con 2 Processi

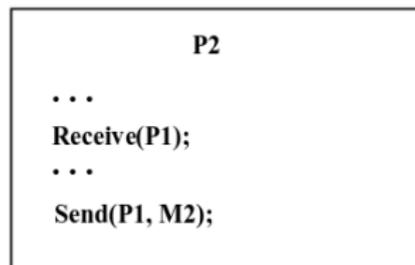
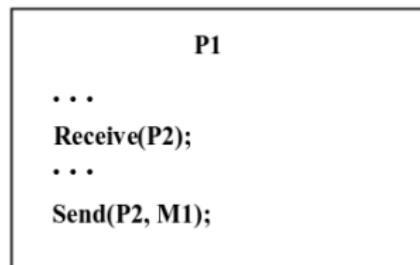
- Supponiamo che ci siano 200 KB di memoria disponibili, e che ci sia la seguente sequenza di richieste



- Il deadlock avverrà quando uno dei due processi farà la seconda richiesta



# Risorse Non Riusabili: Esempio con 2 Processi



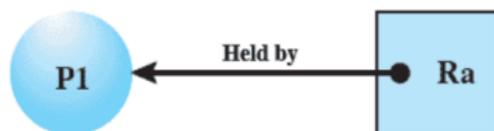
Deadlock inevitabile, processi scritti molto male...

# Grafo dell'Allocazione delle Risorse

- Grafo diretto che rappresenta lo stato di risorse e processi
  - tanti pallini quante istanze di una stessa risorsa
- Ok per le risorse riusabili
- Per le risorse consumabili, non esiste mai l' "Held"; invece, i pallini compaiono e scompaiono...



(a) Resource is requested

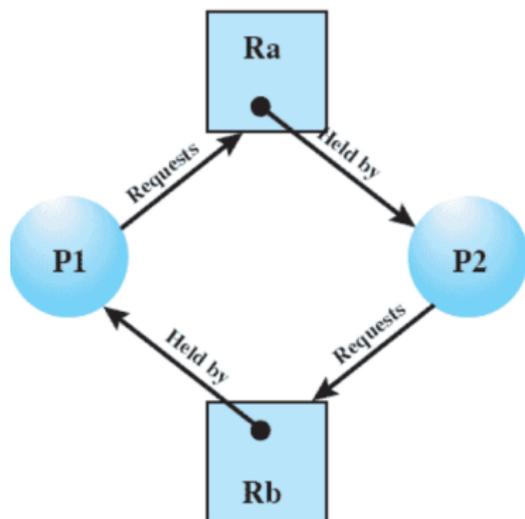


(b) Resource is held

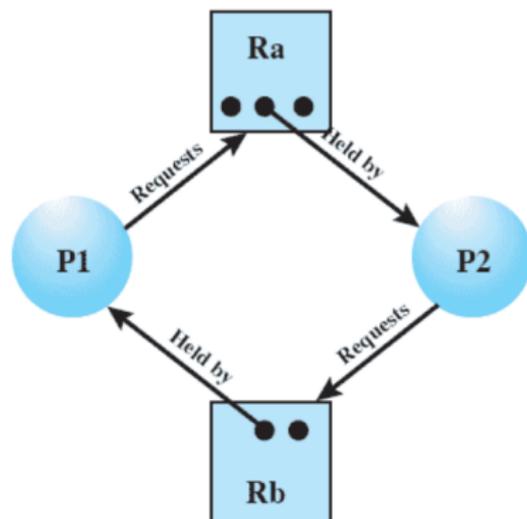




# Grafi dell'Allocazione delle Risorse

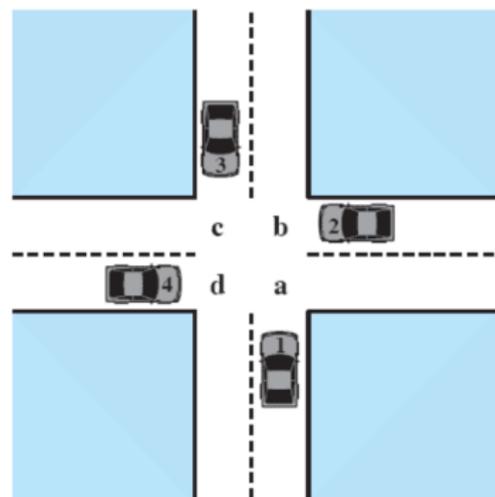


(c) Circular wait

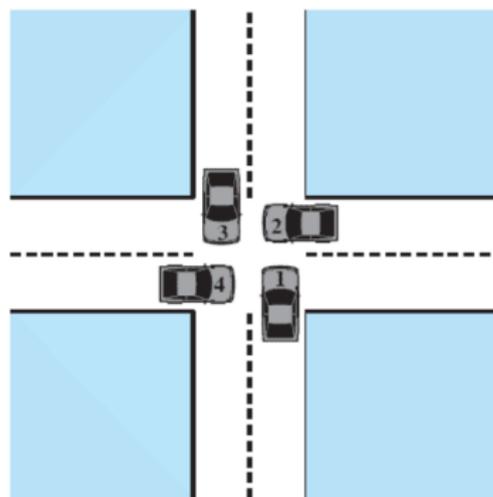


(d) No deadlock

# Grafi dell'Allocazione delle Risorse

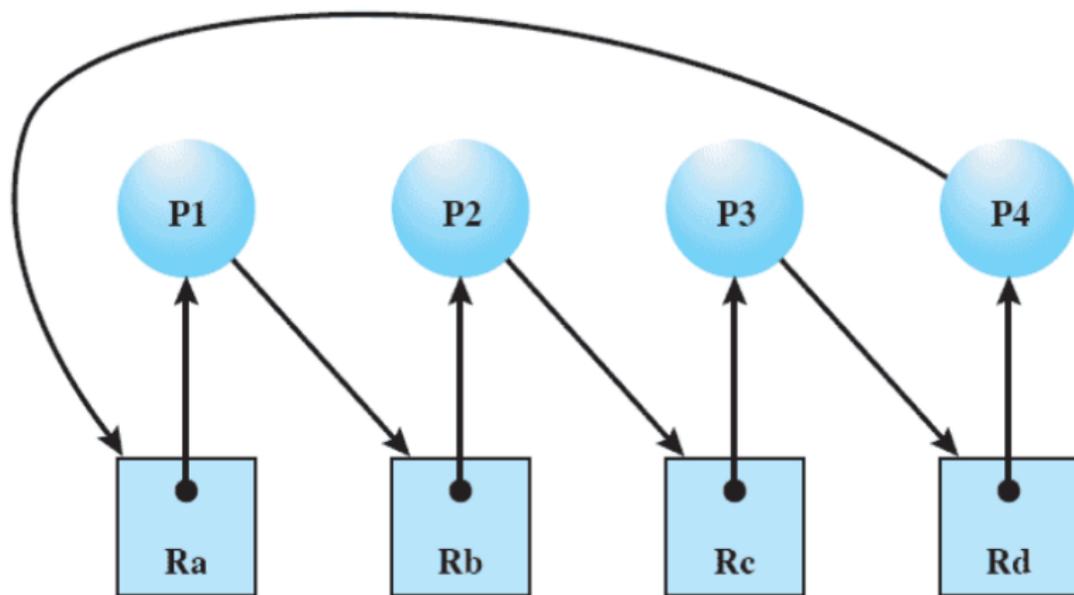


(a) Deadlock possible



(b) Deadlock

# Grafi dell'Allocazione delle Risorse



1) c'è un ciclo; 2) nessun pallino è scoperto

# Possibilità di Deadlock

- Mutua esclusione
- Richiesta di una risorsa quando se ne ha già una (*hold-and-wait*)
- Niente preemption per le risorse

# Esistenza di Deadlock

- Mutua esclusione
- Richiesta di una risorsa quando se ne ha già una (*hold-and-wait*)
- Niente preemption per le risorse
- Attesa circolare

# Deadlock e SO: Che fare?

- Prevenire
  - cercando di far sì che una delle 4 condizioni sia sempre falsa
- Evitare
  - decidendo di volta in volta cosa fare con l'assegnazione di risorse
- Rilevare
  - il SO lascia che eventualmente ci sia deadlock
  - ogni tanto, il SO vede se si è verificato, e o notifica all'utente o prende decisioni per rimuoverlo
- Ignorare
  - se dei processi utente vanno in deadlock, è colpa dell'utente
  - non accettabile, in generale, per processi del sistema operativo

# Prevenzione del Deadlock

- Mutua esclusione
  - non si può. Punto
- Hold-and-wait
  - si impone ad un processo di richiedere tutte le sue risorse in una volta
  - può essere difficile per software complessi, e si tengono risorse bloccate per un tempo troppo lungo
- Niente preemption per le risorse
  - il SO può richiedere ad un processo di rilasciare le sue risorse (le dovrà richiedere in seguito)
  - per esempio, se una sua richiesta di un'altra risorsa non è stata concessa
- Attesa circolare
  - si definisce un ordinamento crescente delle risorse; una risorsa viene data solo se segue quelle che il processo già detiene



# Evitare il Deadlock: Diniego di Risorse

- Algoritmo del banchiere
  - valido per risorse riusabili
- Lo stato del sistema è l'attuale allocazione delle risorse ai processi
- Uno stato è sicuro (*safe*) se da esso parte almeno un cammino che non porta ad un deadlock
- Uno stato non sicuro è insicuro (*unsafe*)

# Algoritmo del Banchiere: Strutture Dati

[Resource = $\mathbf{R} = (R_1, R_2, \dots, R_m)$	total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \dots, V_m)$	total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{bmatrix}$	$C_{ij}$ = requirement of process $i$ for resource $j$
Allocation = $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{bmatrix}$	$A_{ij}$ = current allocation to process $i$ of resource $j$

# Determinazione dello Stato Sicuro

	R1	R2	R3		R1	R2	R3		R1	R2	R3
P1	3	2	2	P1	1	0	0	P1	2	2	2
P2	6	1	3	P2	6	1	2	P2	0	0	1
P3	3	1	4	P3	2	1	1	P3	1	0	3
P4	4	2	2	P4	0	0	2	P4	4	2	0
Claim matrix C				Allocation matrix A				C - A			
Resource vector R			Available vector V								
R1	R2	R3	R1	R2	R3						
9	3	6	0	1	1						

(a) Initial state

# Determinazione dello Stato Sicuro

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
6	2	3

Available vector V

(b) P2 runs to completion

# Determinazione dello Stato Sicuro

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
7	2	3

Available vector V

(c) P1 runs to completion

# Determinazione dello Stato Sicuro

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

(d) P3 runs to completion

# Determinazione dello Stato Non Sicuro

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	1	1	2

Available vector V

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	0	1	1

Available vector V

(b) P1 requests one unit each of R1 and R3

# Evitare il Deadlock: Pseudocodice

```
struct state {  
    int resource[m];  
    int available[m];  
    int claim[n][m];  
    int alloc[n][m];  
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])  
    < error >; /* total request > claim*/  
else if (request [*] > available [*])  
    < suspend process >;  
else { /* simulate alloc */  
    < define newstate by:  
    alloc [i,*] = alloc [i,*] + request [*];  
    available [*] = available [*] - request [*] >;  
}  
if (safe (newstate))  
    < carry out allocation >;  
else {  
    < restore original state >;  
    < suspend process >;  
}
```

(b) resource alloc algorithm

# Evitare il Deadlock: Pseudocodice

```
boolean safe (state S) {
  int currentavail[m];
  process rest[<number of processes>];
  currentavail = available;
  rest = {all processes};
  possible = true;
  while (possible) {
    <find a process Pk in rest such that
      claim [k,*] - alloc [k,*] <= currentavail;>
    if (found) {
      /* simulate execution of Pk */
      currentavail = currentavail + alloc [k,*];
      rest = rest - {Pk};
    }
    else possible = false;
  }
  return (rest == null);
}
```

# Evitare il Deadlock: Pseudocodice

- Occorre dichiarare il massimo uso di risorse
- I processi devono essere indipendenti, senza requisiti di sincronizzazione
  - ovvero, devono essere “liberi” di andare in esecuzione in un qualsiasi ordine
  - altrimenti, non si può simularne l'esecuzione fino al completamento...
  - in sostanza, l'unica sincronizzazione presente è proprio quella sulle richieste di risorse
- Ci deve essere un numero fissato di risorse da allocare
  - quindi, non va bene per le risorse consumabili
- Nessun processo deve terminare senza rilasciare le sue risorse

# Rilevare il Deadlock: Pseudocodice

- Stesse strutture dati del banchiere, ma la claim matrix è sostituita da  $Q$ : le richieste effettuate da tutti i processi
  - come il request del banchiere, ma relativo appunto a tutti i processi e non ad uno solo
- Algoritmo:
  - 1 marca tutti i processi che non hanno allocato nulla
  - 2  $w \leftarrow V$
  - 3 sia  $i$  un processo non marcato t.c.  $Q_{ik} \leq w_k \forall 1 \leq k \leq m$ 
    - le sue risorse possono essere accordate
  - 4 se  $i$  non esiste, vai al passo 6
  - 5 marca  $i$  e aggiorna  $w \leftarrow w + A_i$ ; poi torna al passo 3
    - facciamo finta che le sue risorse vengano liberate
  - 6 c'è un deadlock se e solo se esiste un processo non marcato

# Rilevare il Deadlock: Esempio

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

# Deadlock Rilevato: e Poi?

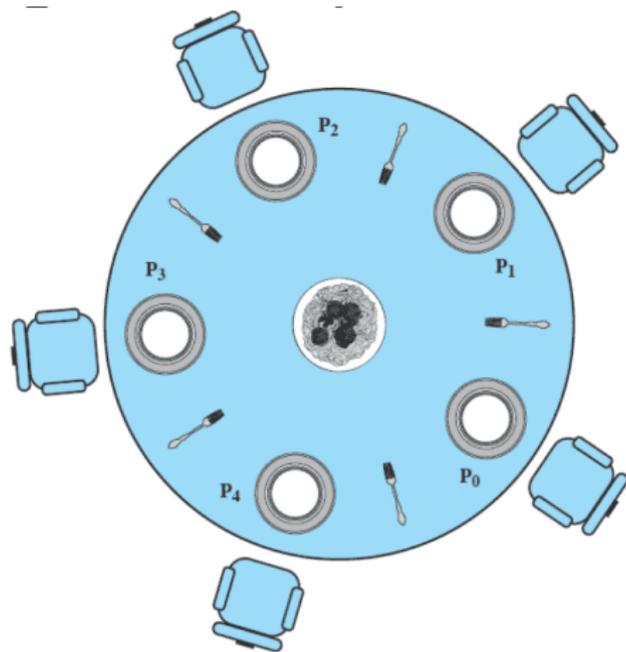
- Terminare forzosamente tutti i processi coinvolti nel deadlock
  - così fan (quasi) tutti
  - almeno un processo non in deadlock resta sempre, basta che non faccia richiesta di risorse...
- Mantenere dei punti di ripristino, ed effettuare il ripristino al punto precedente
  - lo stallo può verificarsi nuovamente, ma è improbabile che lo faccia all'infinito
- Terminare forzosamente i processi coinvolti nel deadlock uno ad uno, finché lo stallo non c'è più
- Sottrarre forzosamente risorse ai processi coinvolti nel deadlock uno ad uno, finché lo stallo non c'è più

# Vantaggi e Svantaggi

Approccio	Politica di Allocazione	Possibili Schemi	Vantaggi Principali	Svantaggi Principali
Prevenire	Conservativa: concede meno risorse di quelle richieste	Richiesta contemporanea di tutte le risorse	OK per processi con singolo <i>burst</i> di computazione; Non richiede preemption	Inefficiente: ritarda l'inizializzazione dei processi; Un processo deve conoscere tutte le sue richieste future, difficile per gli interattivi
		Preemption	OK se le risorse hanno uno stato facile da salvare e ripristinare	La preemption può avvenire troppo spesso
		Ordinamento delle risorse	Possibile con controlli a tempo di compilazione; Niente controlli a run-time, risolto con il progetto del SO	Non possibile per processi interattivi
Evitare	A metà tra prevenire e rilevare	Si cerca di trovare almeno un cammino sicuro	Niente prevenzione	Necessità di risorse future da conoscere in anticipo
Rilevare	Molto liberale: concede più risorse di quelle possibili	Controllo del deadlock da fare periodicamente	Niente ritardo sull'inizializzazione; Facilita la gestione delle risorse online	Gesione del deadlock quando avviene



# I Filosofi a Cena



# I Filosofi a Cena: Prima Soluzione

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher
(2),
            philosopher (3), philosopher (4));
}
```

# I Filosofi a Cena: Prima Soluzione

- La dichiarazione della variabile  $i$  globale (terza riga) è inutile
  - vale anche per gli altri pseudocodici
- Per il filosofo  $i$ ,  $\text{fork}[i]$  è la forchetta sinistra,  $\text{fork}[(i + 1) \bmod n]$  è la destra
- Ci può essere deadlock
  - lo scheduler fa sì che ogni processo faccia la wait sulla sua forchetta sinistra
    - nessuno viene bloccato
  - e poi fa fare a tutti la wait su quelle destre
    - tutti vengono bloccati

# I Filosofi a Cena: Seconda Soluzione

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}

void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
             philosopher (3), philosopher (4));
}
```

## I Filosofi a Cena: Seconda Soluzione

- Non si pensa al tavolo, ma in un'altra stanza
- Quando si ha fame, si entra nella sala da pranzo
- C'è un cameriere che fa entrare al massimo  $n - 1$  filosofi alla volta
  - semaforo room
- Niente più deadlock

# I Filosofi a Cena: III, IV e V Soluzione

- Nella soluzione precedente, si sono leggermente variati i termini del problema
- Si impone ai filosofi di non mangiare tutti insieme
  - o meglio, tutti seduti a tavola
- Ma può essere fatto, perché limitarsi?
- È possibile risolvere il problema dei filosofi senza deadlock e senza queste assunzioni
- Lo vedremo con i semafori e con i messaggi
  - per quella con i messaggi, due varianti (più una versione errata)

## I Filosofi a Cena: Terza Soluzione

```
semaphore fork[N] = {1, 1, ... 1};
philosopher(int me) {
    int left, right, first, second;
    left = me;
    right = (me + 1) % N;
    first = right < left ? right : left;
    second = right < left ? left : right;
    while (1) {
        think();
        wait(fork[first]);
        wait(fork[second]);
        eat();
        signal(fork[first]);
        signal(fork[second]);
    }
}
```

## I Filosofi a Cena: Terza Soluzione

- È come la prima soluzione, ma si fa in modo che non tutti i filosofi prendano prima la forchetta sinistra e poi la destra
- Uno di loro farà il contrario
  - ovvero, quello con PID  $n - 1$
- A questo punto, il deadlock non è più possibile
  - anche considerando il caso di prima, il filosofo  $n - 1$  non riuscirà a prendere la sua prima forchetta, che è già bloccata
  - quindi, almeno il filosofo  $n - 2$  può procedere

# I Filosofi a Cena: Quarta Soluzione (Sbagliata)

Attenzione, ci può essere stallo

```
mailbox fork[N];

init_forks() {
    int i;

    for (i = 0; i < N; ++i)
        nbsend(fork[i], "fork");
}
```



## I Filosofi a Cena: Quarta Soluzione (Corretta)

```
philosopher(int me) {
    int left, right;
    message fork1, fork2;
    left = me;
    right = (me + 1) % N;
    first = right < left ? right : left;
    second = right < left ? left : right;

    while (1) {
        /* come sopra */
    }
}
```

## I Filosofi a Cena: Quinta Soluzione

```
philosopher(int me) {
    int left, right;
    message fork1, fork2;
    left = me;
    right = (me + 1) % N;
    while (1) {
        think_for_a_random_time();
        if (nbreceive(fork[left], fork1)) {
            if (nbreceive(fork[right], fork2)) {
                eat();
                nbsend(fork[left], fork2);
            }
            nbsend(fork[right], fork1);
        }
    }
}
```

