Sistemi Operativi Modulo I

Primo canale (A-L) e Teledidattica
A.A. 2018/2019
Corso di Laurea in Informatica

Il Deadlock (o Stallo)

Igor Melatti

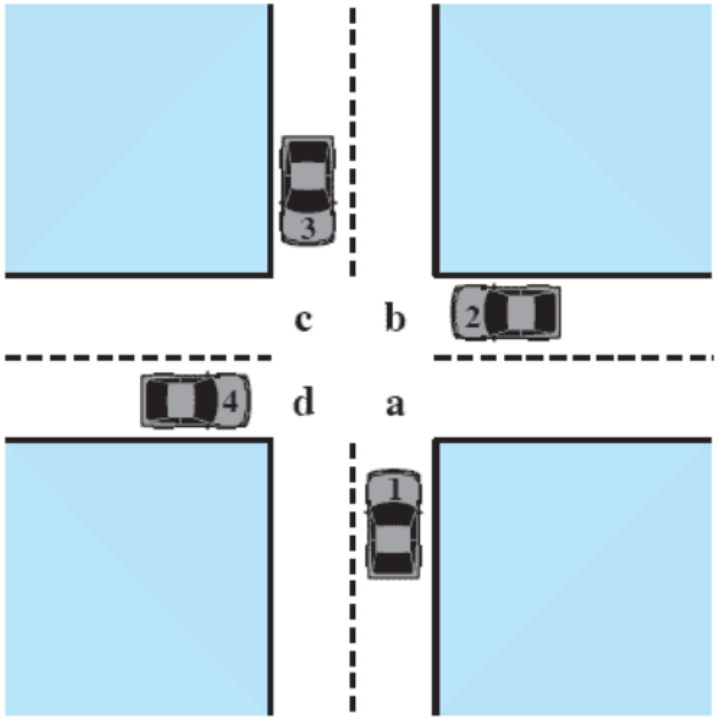
Sapienza Università di Roma
Dipartimento di Informatica

SAPIENZA
UNIVERSITÀ DI ROMA

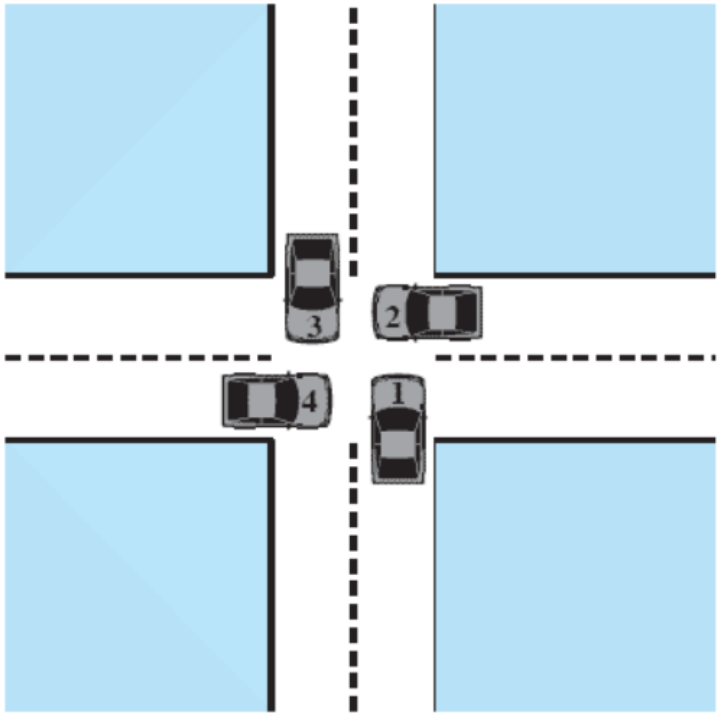
Il Deadlock o Stallo

- Blocco permanente di un insieme di processi, che o competono per delle risorse di sistema o comunicano tra loro
- Il motivo di base è la richiesta contemporanea delle stesse risorse da parte di due o più processi
- Non esiste una soluzione efficiente
  - diversi approcci, da valutare nei vari casi
- Il deadlock potrebbe essere causato da una combinazione rara di eventi
  - *corner case*

Deadlock

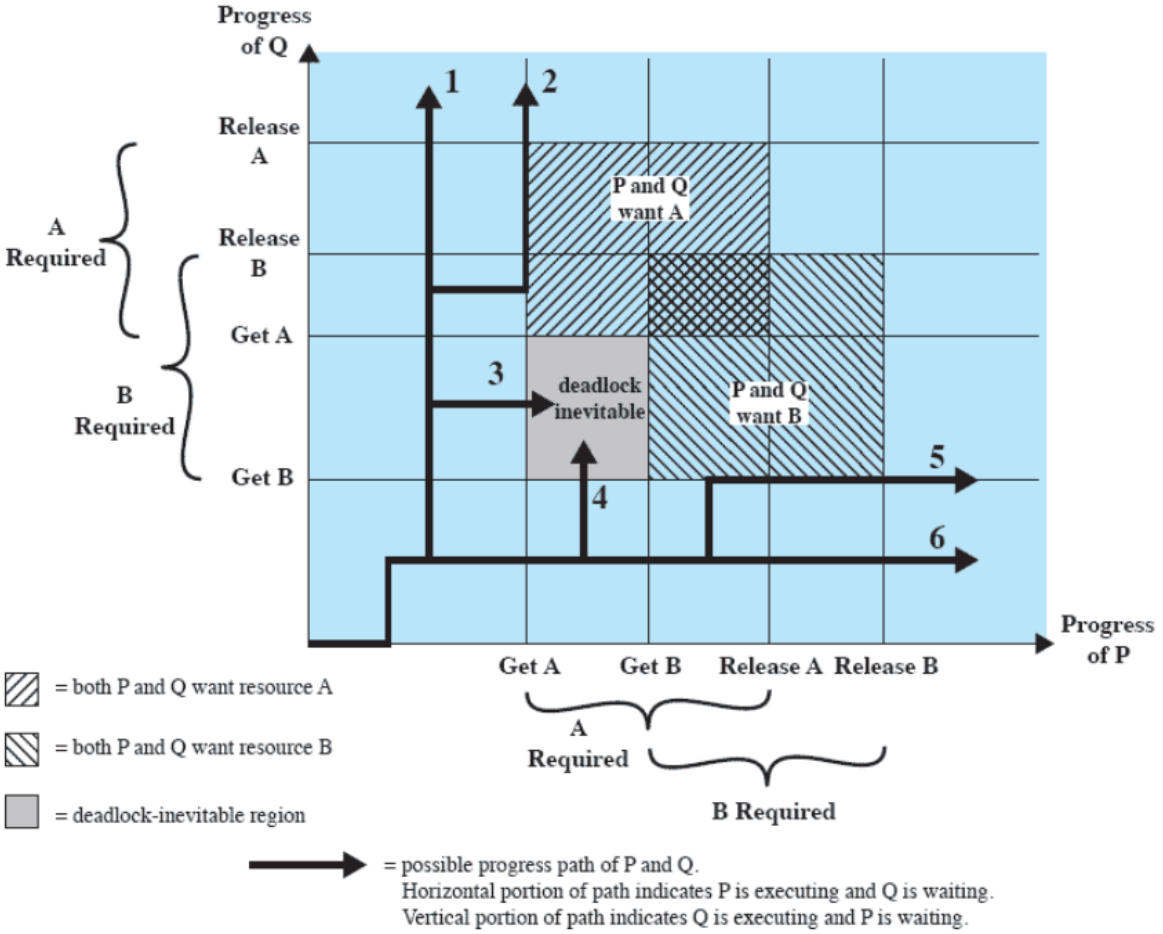


(a) Deadlock possible

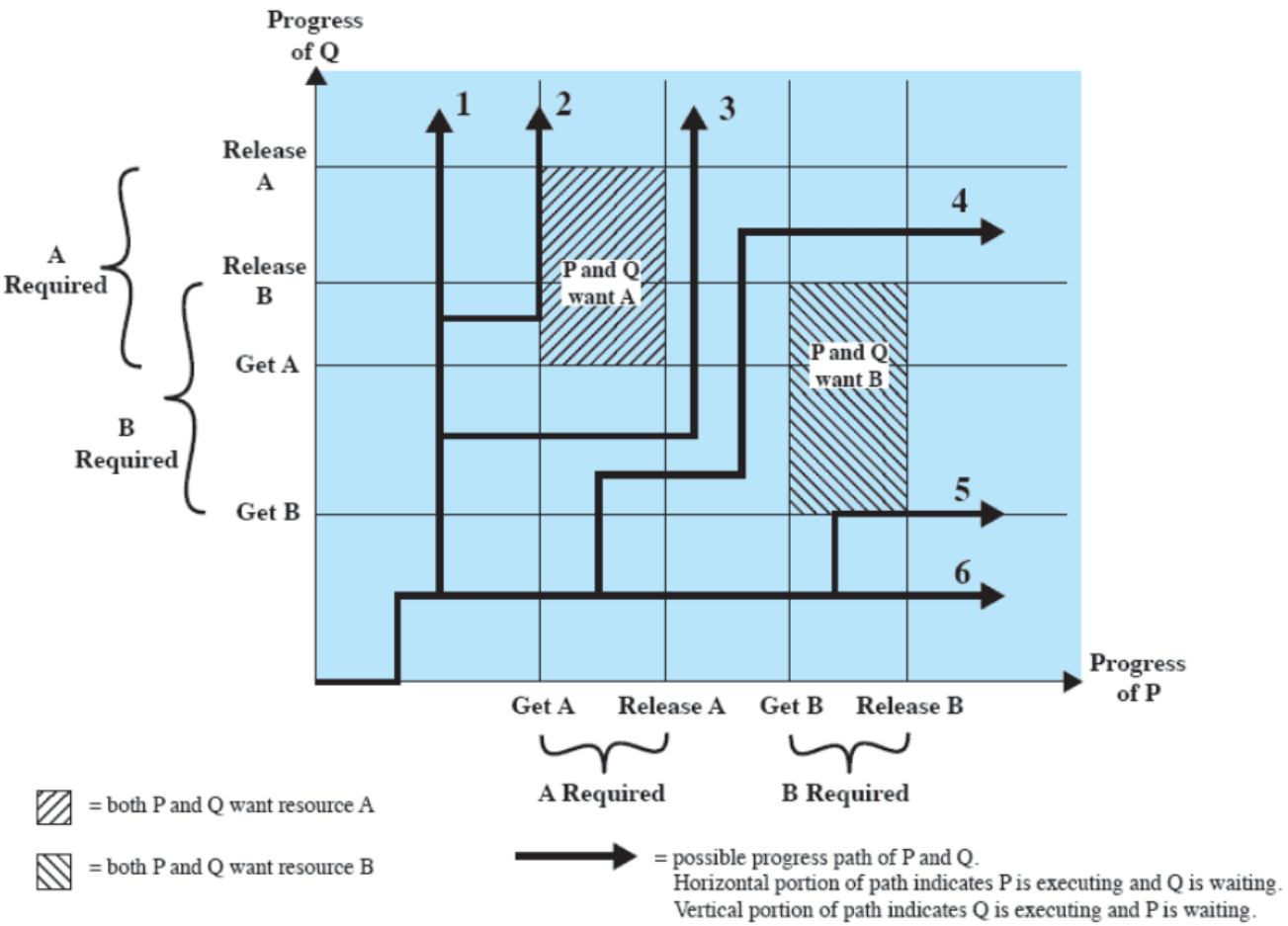


(b) Deadlock

Deadlock: Joint Progress Diagram



Deadlock: Joint Progress Diagram



Risorse Riusabili (*Reusable Resources*)

- Usabili da un solo processo alla volta
- Il fatto di essere usate non le "consuma"
- Una volta che un processo ottiene una risorsa riusabile, prima o poi la rilascia
  - cosicché altri processi possano usarla a loro volta
- Esempi: processori, I/O channels, memoria primaria e secondaria, dispositivi, file, database, semafori
- Lo stallo può esistere solo se un processo ha una risorsa e ne richiede un'altra

Risorse Riusabili: Esempio 1 con 2 Processi

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Risorse Riusabili: Esempio 2 con 2 Processi

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

P1	P2
. . .	. . .
Request 80 Kbytes;	Request 70 Kbytes;
. . .	. . .
Request 60 Kbytes;	Request 80 Kbytes;

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

Risorse Non Riusabili (*Consumable Resources*)

- Vengono create (prodotte) e distrutte (consumate)
- Esempi: interruzioni, segnali, messaggi, informazione nei buffer di I/O
- Deadlock possibile se si fa una richiesta (bloccante) di una risorsa ancora non creata
- Ad esempio, un deadlock può avvenire su una ricezione bloccante

Risorse Non Riusabili: Esempio con 2 Processi

P1

```
. . .
Receive(P2);
. . .
Send(P2, M1);
```

P2

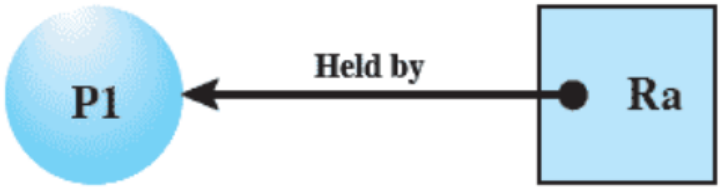
```
. . .
Receive(P1);
. . .
Send(P1, M2);
```

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) Resouce is requested



(b) Resource is held

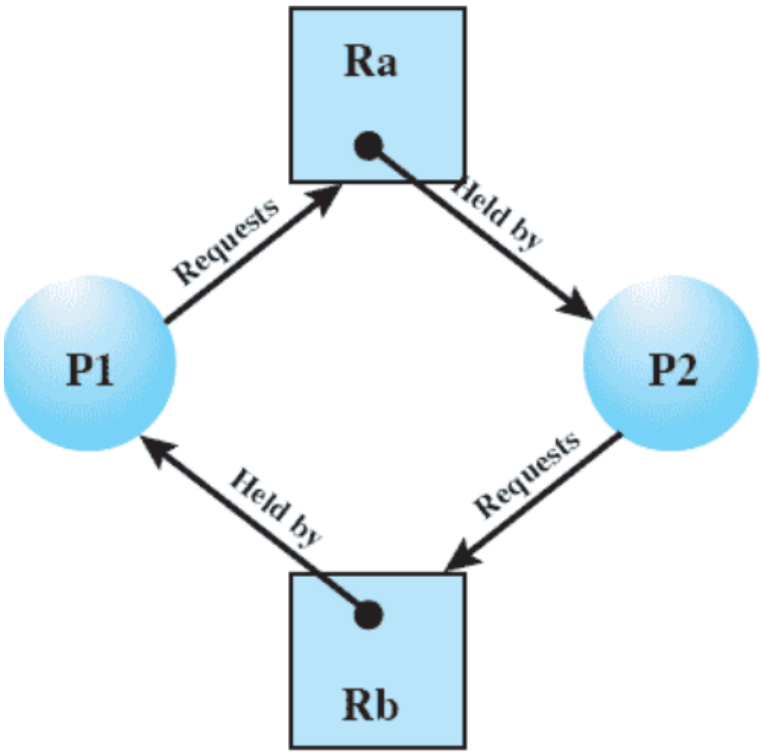
Condizioni per il Deadlock (Risorse Riusabili)

- Mutua esclusione
  - solo un processo alla volta può usare una data risorsa
- Richiesta di una risorsa quando se ne ha già una (*hold-and-wait*)
  - un processo può richiedere risorse mentre ne tiene già bloccate altre
- Niente preemption per le risorse
  - non si può sottrarre una risorsa ad un processo senza che quest'ultimo non la rilasci
- Attesa circolare
  - esiste una catena chiusa di processi, in cui ciascun processo detiene una risorsa richiesta (invano) dal processo che lo segue nella catena

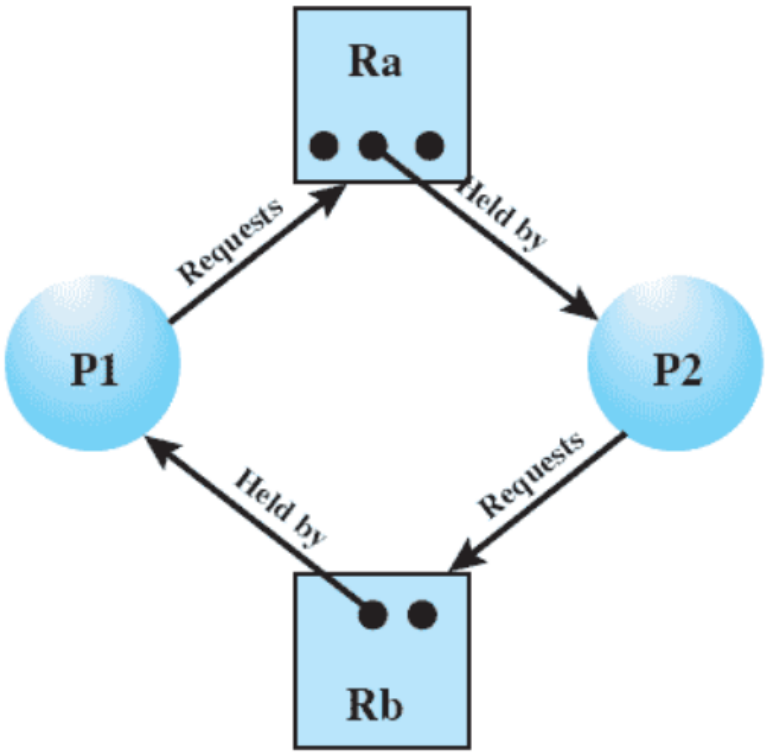
Condizioni per il Deadlock (Risorse Consumabili)

- Mutua esclusione
  - ovvia (la risorsa va a chi ne riesce a fare richiesta per primo)
- Richiesta di una risorsa quando se ne ha già una (*hold-and-wait*)
  - diventa: si può richiedere (in modo bloccante) una risorsa quando ancora non sia stata creata
- Niente preemption per le risorse
  - ovvia (non appena viene concessa, una risorsa viene distrutta...)
- Attesa circolare
  - esiste una catena chiusa di processi, in cui ciascun processo dovrebbe creare una risorsa richiesta (invano) dal processo che lo segue nella catena

Grafi dell'Allocazione delle Risorse

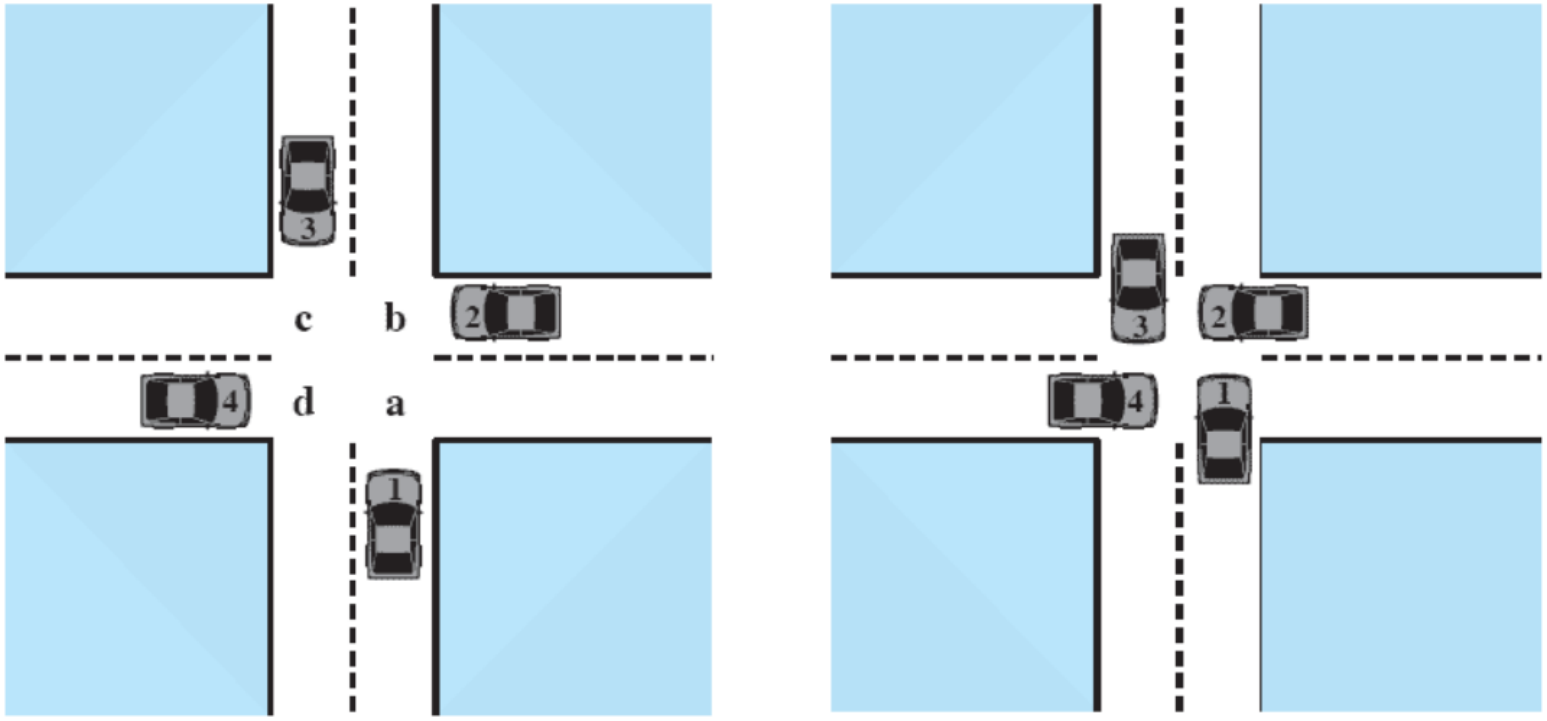


(c) Circular wait



(d) No deadlock

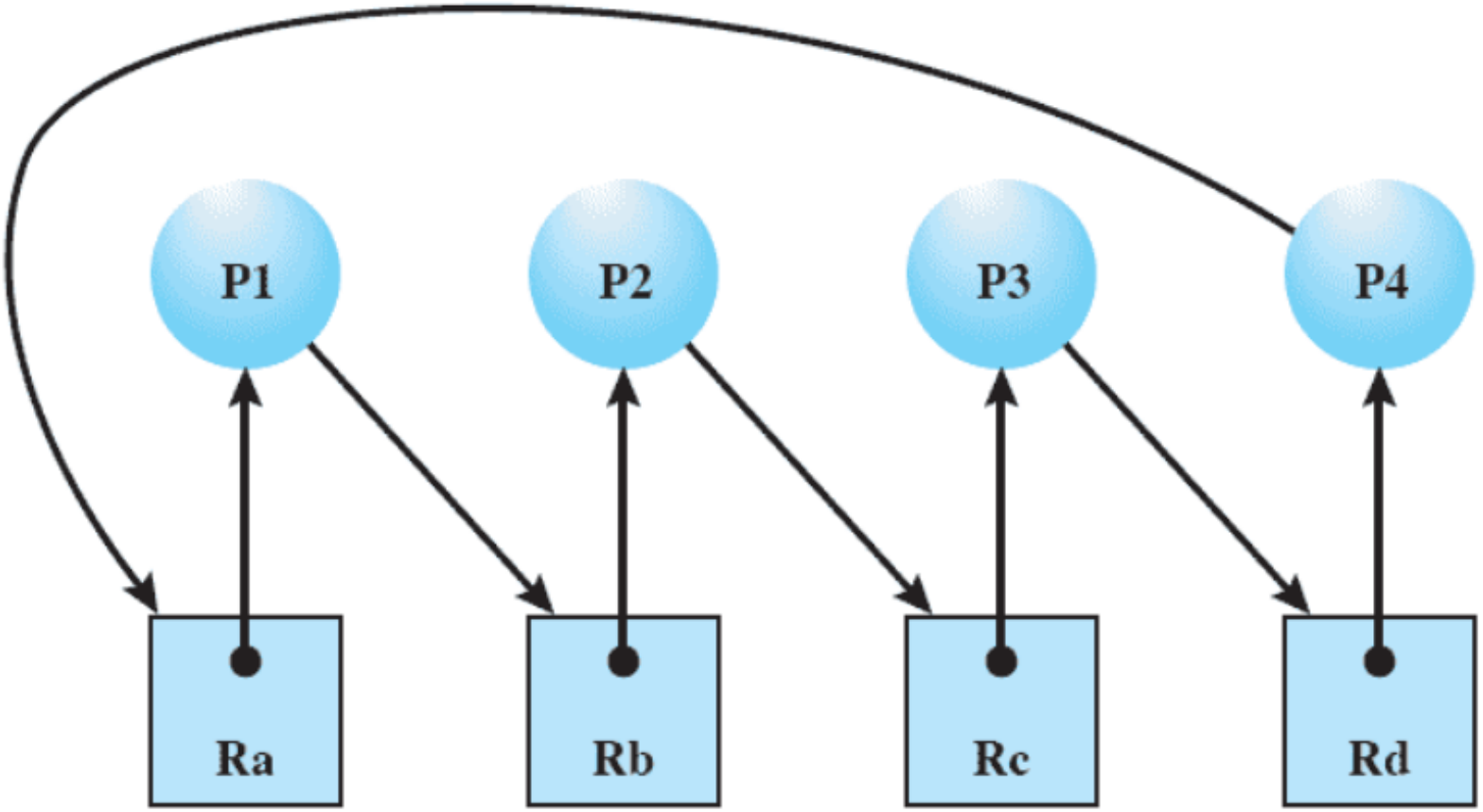
Grafi dell'Allocazione delle Risorse



(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

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

- Occorre decidere se l'attuale richiesta di una risorsa può portare ad un deadlock, se esaudita
- Richiede la conoscenza delle richieste future
- Due possibilità:
  - non mandare in esecuzione un processo se le sue richieste possono portare a deadlock
  - non concedere una risorsa ad un processo se allocarla può portare a deadlock

Evitare il Deadlock: Diniego di Risorse

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

Determinazione dello Stato 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	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
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**

(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...
- 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

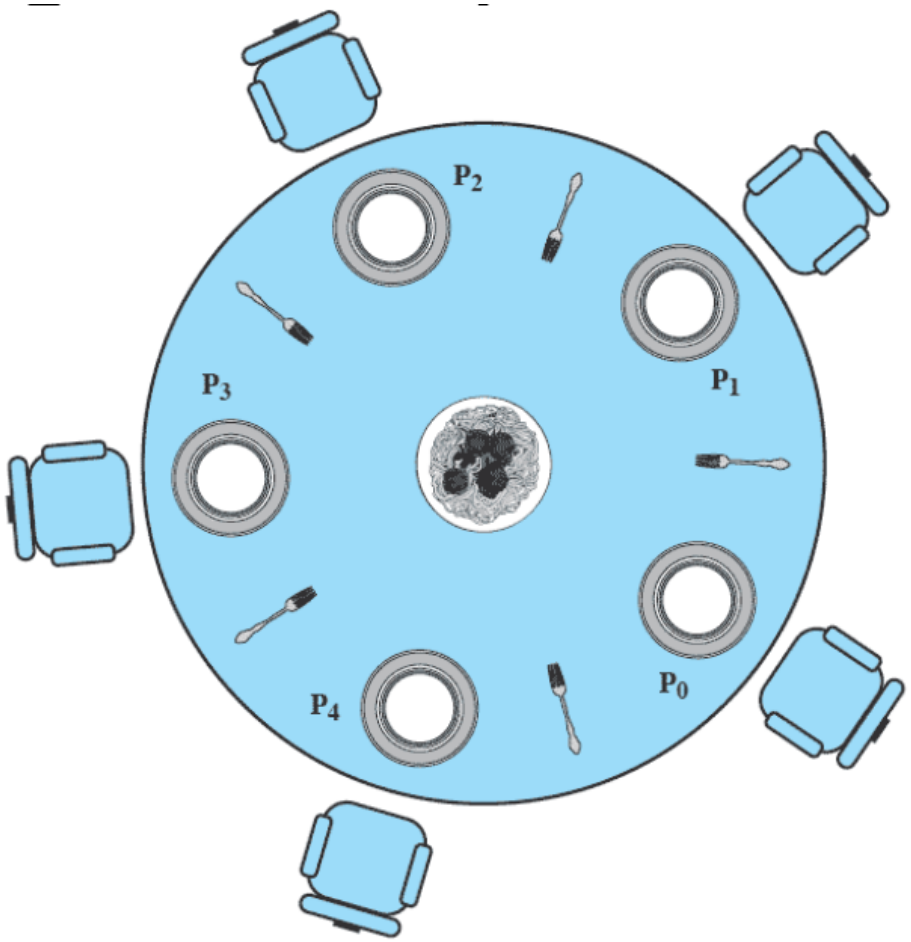
Deadlock Rilevato: e Poi?

- 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 *burst* di computazione; Non richiede preemption	Inefficiente; Ritarda l'inizializzazione dei processi; Necessità di risorse future da conoscere in anticipo
		Preemption	OK se le risorse hanno uno stato facile da salvare e ripristinare	La preemption avviene più volte del necessario
		Ordinamento delle risorse	Possibile con controlli a tempo di compilazione; Niente controlli a run-time, risolto con il progetto del SO	Inefficiente; Non permette richieste incrementali
Evitare	A metà tra prevenire e rilevare	Si cerca di trovare almeno un cammino sicuro	Niente preemption	Necessità di risorse future da conoscere in anticipo
Rilevare	Molto liberale: le richieste sono concesse quando è possibile	Controllo del deadlock da fare periodicamente	Niente ritardo sull'inizializzazione; Facilita la gestione delle risorse online	Perdite da preemption

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, `fork[`i`]` è la forchetta sinistra, `fork[`$(i+1)$ `mod` 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 (Sbagliata)

Attenzione, ci può essere stallo

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

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);
    }
  }
}
```

I Filosofi a Cena: Quinta Soluzione

- Ci possono essere livelock e starvation
- Ma non lo stallo
- Si basa sul fatto che i filosofi "pensino" per un tempo casuale, scorrelato da quello degli altri