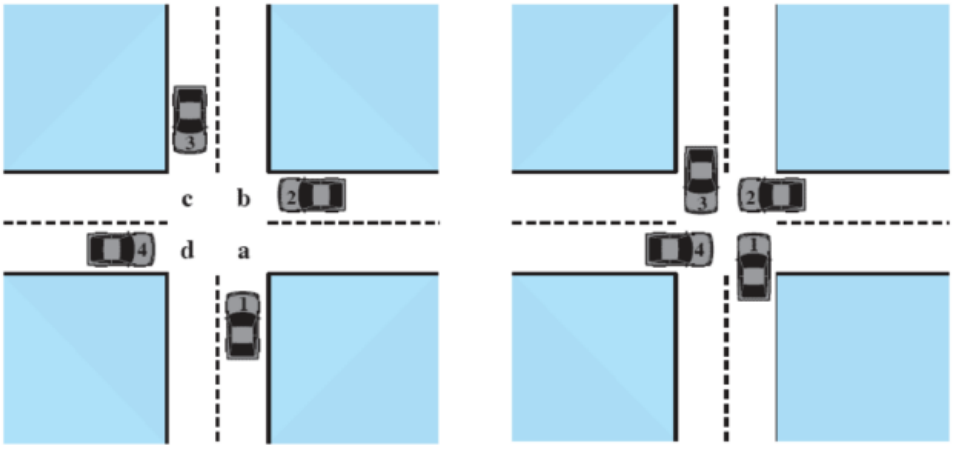
Deadlock

Index

Introduction

Il **deadlock** (o stallo) è un blocco permanente di un certo insieme di processi che competono per delle risorse di sistema o comunicano tra loro.
Il motivo di base del deadlock è la **richiesta contemporanea delle stesse risorse** da parte di due o più processi.

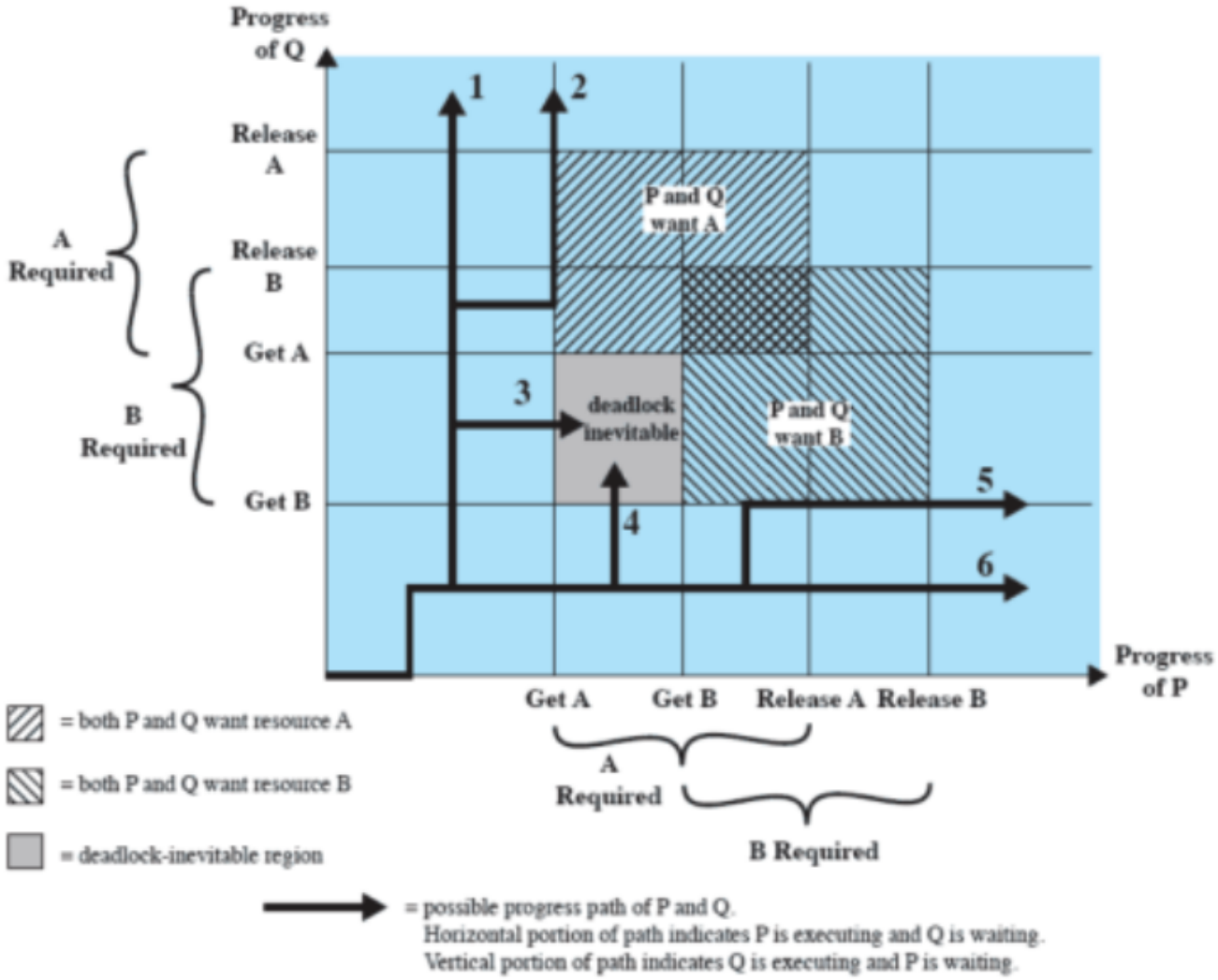


Nonostante tutto non esiste una soluzione universale per risolvere questo problema, bisogna infatti analizzare caso per caso e risolverlo in una maniera opportuna.

Joint progress diagram

Quando ci troviamo di fronte ad un deadlock tra due processi, questo può essere analizzato attraverso questo semplice diagramma

> ⓘ **I due processi richiedono la risorsa successiva prima di aver rilasciato quella che stanno usando**

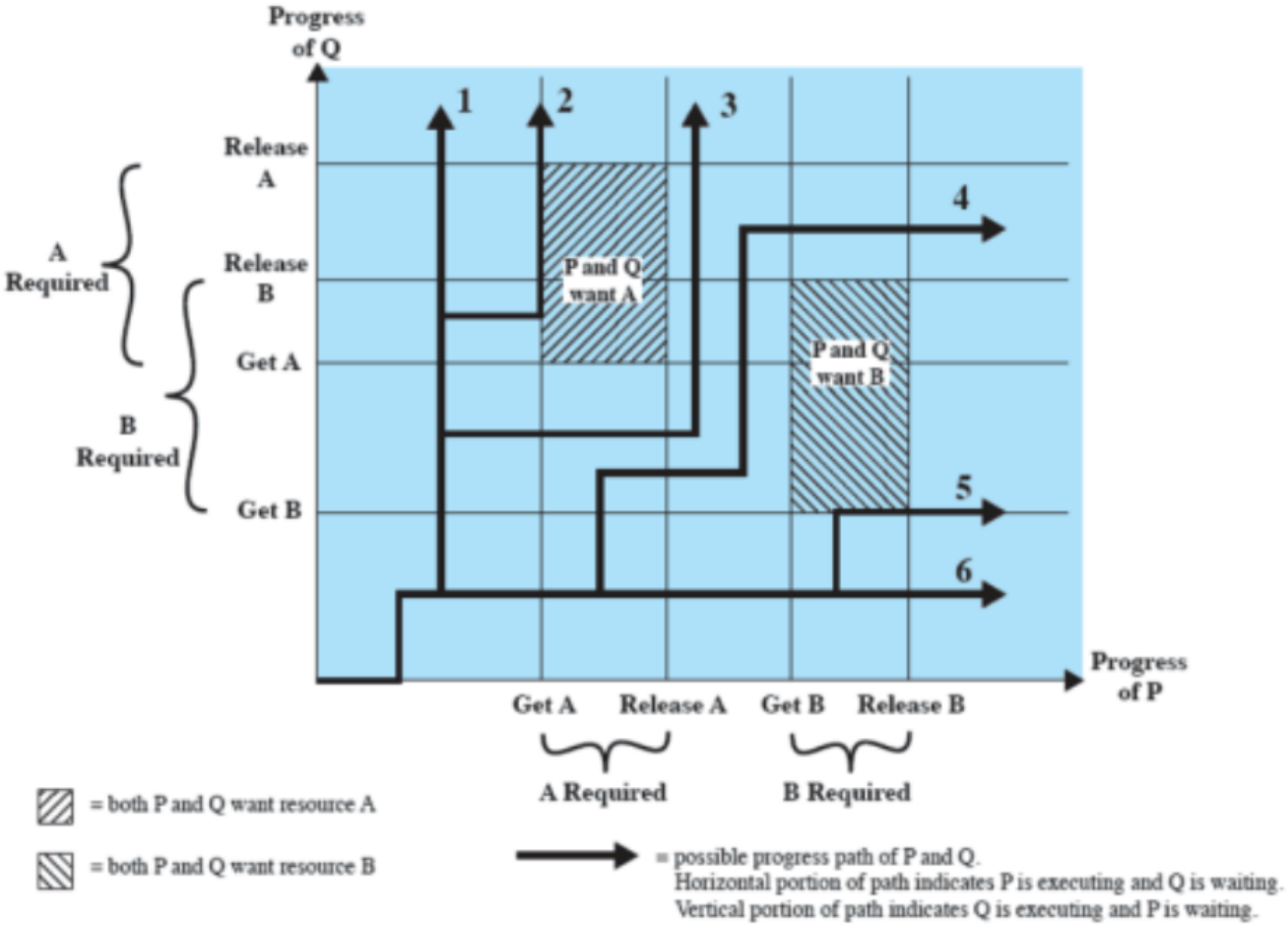


Note

Linee orizzontali → momento in cui `P` è in esecuzione
Linee verticali → momento in cui `Q` è in esecuzione
Quando intercettano un quadrante vuol dire che viene eseguita l'operazione indicata (es. Get A)

5 non va in deadlock, infatti prova a richiedere una risorsa già occupata, quindi il processo viene mezzo in blocked
Dopo 3 e 4 non si può andare da nessuna parte per come sono scritti i processi (devono richiedere l'altra risorsa prima di poter rilasciare quella che stanno usando ma è occupata)

`P` prima di richiedere la seconda risorsa rilascia quella che sta usando, mentre `B` si comporta come prima



> **Note**
>
> Per come è strutturato questo diagramma, non ci può essere deadlock

---

Risorse

Le risorse si distinguono in:

- **risorse riusabili**
- **risorse non riusabili**

Risorse riusabili

Le risorse riusabili sono quelle risorse usabili da un solo processo alla volta, ma 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
Esempio: processori, I/O channels, memoria primaria e secondaria, dispostivi, file…

Nel caso delle risorse riusabili il deadlock può avvenire solo se un processo ha una risorsa e ne richiede un'altra

Esempio

Esempio 1

Process P		Process Q	
Step	Action	Step	Action
p_0	Request (D)	q_0	Request (T)
p_1	Lock (D)	q_1	Lock (T)
p_2	Request (T)	q_2	Request (D)
p_3	Lock (T)	q_3	Lock (D)
p_4	Perform function	q_4	Perform function
p_5	Unlock (D)	q_5	Unlock (T)
p_6	Unlock (T)	q_6	Unlock (D)

Note

Perform action → sezione critica

Si bloccano in quanto `P` richiede `T` prima di rilasciare `D`, mentre `Q` richiede `D` prima di rilasciare `T`

Esempio 2

Supponiamo di trovarci in un sistema con $200\ \mathrm{KB}$ 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 al seconda richiesta (non rilasciano la memoria)

Condizioni per il deadlock

Il deadlock si verifica solamente se ci sono queste quattro condizioni:

- **Mutua esclusione** → solo un processo alla volta può usare una data risorsa
- **Hold-and-wait** → un processo può richiedere una risorsa mentre ne tiene già bloccate altre
- **Niente preemption** per le risorse → non si può sottrarre una risorsa ad un processo senza che quest'ultimo 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

---

Risorse non riusabili

Le risorse non riusabili sono quelle risorse che vengono **create da un processo e distrutte da un altro processo**
Esempi: interruzioni, segnali, messaggi, informazione nei buffer di I/O

Nel caso delle risorse non riusabili il deadlock può avvenire se si fa una richiesta (bloccante) per una risorsa che non è stata ancora creata, ad esempio un deadlock può avvenire su una ricezione bloccante

Esempio

Esempio

Condizioni per il deadlock

Il deadlock si verifica solamente se ci sono queste quattro condizioni:

- **Mutua esclusione** → la risorsa va a chi ne riesce a fare richiesta per primo (un messaggio arriva al primo che riceve)
- **Hold-and-wait** → si può richiedere (in modo bloccante) una risorsa che non è stata ancora creata (receive ma non ci sta la send corrispondente)
- **Niente preemption** per le risorse → non appena viene concessa, una risorsa viene distrutta

- **Attesa circolare** → esiste una catena chiusa di processi, in cui ciascun dovrebbe creare una risorsa richiesta (invano) dal processo che lo segue nella catena

---

Grafo dell'allocazione delle risorse

Dato che il joint process diagram non è sufficiente per rappresentare le interazioni tra più processi che richiedono più risorse, si utilizza il **grafo dell'allocazione delle risorse**

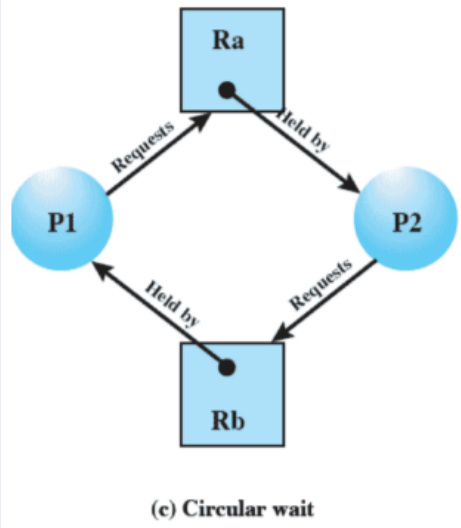
Questo è un grafo diretto che rappresenta lo stato di risorse e processi (tanti pallini quante istanze di una stessa risorsa, tre pallini tre stampanti). Come nodi si utilizzano i cerchi per i processi mentre i quadrati per le risorse

In base alla direzione della freccia si determina se la risorsa è richiesta o tenuta da un processo. Mentre questo sistema risulta 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

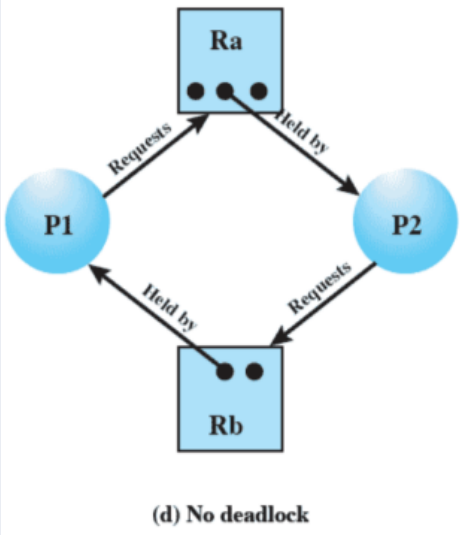
Info



(c) Circular wait

- Mutua esclusione → sia `Ra` che `Rb` possono essere prese da un solo processo alla volta
- Hold-and-wait → `P1` richiede `Ra` e detiene `Rb` e `P2` viceversa

- Niente preemption → SO non può togliere le risorse
- Attesa circolare → visivamente può essere vista da un ciclo

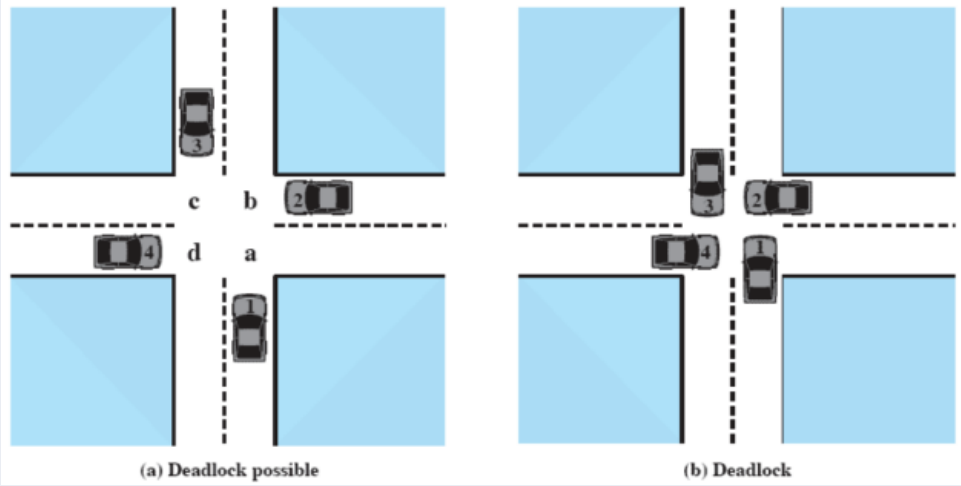


(d) No deadlock

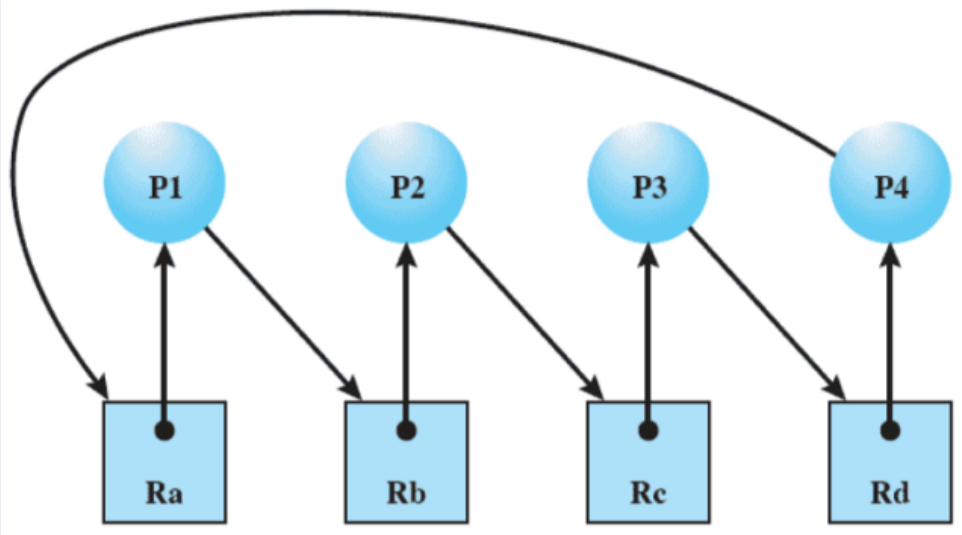
- Niente mutua esclusione

ⓘ Info

Anche questo esempio delle macchine può essere visualizzato attraverso un grafo



(a) Deadlock possible (b) Deadlock



1. C’è un ciclo
2. Nessun pallino è scoperto

Possibilità ed esistenza di deadlock

Possibilità di deadlock

Ci sta la possibilità che si presenti un deadlock quando sono verificate le prime tre condizioni:

- mutua esclusione
- richiesta di una risorsa quando se ne ha già una (hold-and-wait)
- niente preemption per le risorse
  Queste infatti dipendono da come è fatto il sistema

Esistenza di deadlock

Effettivamente è presente un deadlock quando tutte e quattro le condizioni sono verificate:

- mutua esclusione
- richiesta di una risorsa quando se ne ha già una (hold-and-wait)
- niente preemption per le risorse
- attesa circolare
  L’attesa circolare invece dipende da come evolve l’esecuzione di certi processi
  Si parla invece di deadlock **inevitabile** quando non è al momento presente l’attesa circolare, ma sicuramente ad un certo punto arriverà

Deadlock e SO: che fare?

Esistono diverse tecniche per gestire problemi che riguardano il deadlock:

- **Prevenire** → cercando di far si che una delle 4 condizioni sia sempre falsa
- **Evitare** → decidendo di volta in volta cosa fare con l'assegnazione di risorse
- **Rilevare** → una volta che avviene il deadlock, viene notificato al SO che agisce di conseguenza
- **Ignorare** → se dei processi vanno in deadlock è colpa dell'utente, non accettabile, in generale, per processi del SO

Prevenzione del deadlock

Per la prevenzione bisogna evitare che esistano contemporaneamente le 4 condizioni per un deadlock. Vediamo cosa si può evitare

- Mutua esclusione → inevitabile
- Hold-and-wait → si impone ad un processo di richiedere tutte le sue risorse in una solva 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
  Le cose più ragionevoli da risolvere sono l'hold-and-wait o l'attesa circolare

---

Evitare il deadlock

Per evitare il deadlock bisogna decidere se l'attuale richiesta di una risorsa può portare ad un deadlock se esaudita. Ma ciò richiede la conoscenza delle richieste future, in particolare si hanno 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 (algoritmo del banchiere, mando in esecuzione il processo ma non concedo la risorsa)

Diniego delle risorse

L'algoritmo del banchiere e valido per le risorse riusabili e fa si che il programma proceda da un certo stato ad un altro stato (per stato si intende una certa situazione per l'uso delle risorse). Uno stato è **sicuro** se da essi parte almeno un cammino che non porta ad un deadlock, uno stato non sicuro se da essi partono solo cammini che portano a deadlock

Algoritmo del banchiere

L'algoritmo del banchiere permette di passare solamente da uno stato sicuro ad uno stato altrettanto sicuro

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 = $\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{bmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{bmatrix}$	A_{ij} = current allocation to process i of resource j

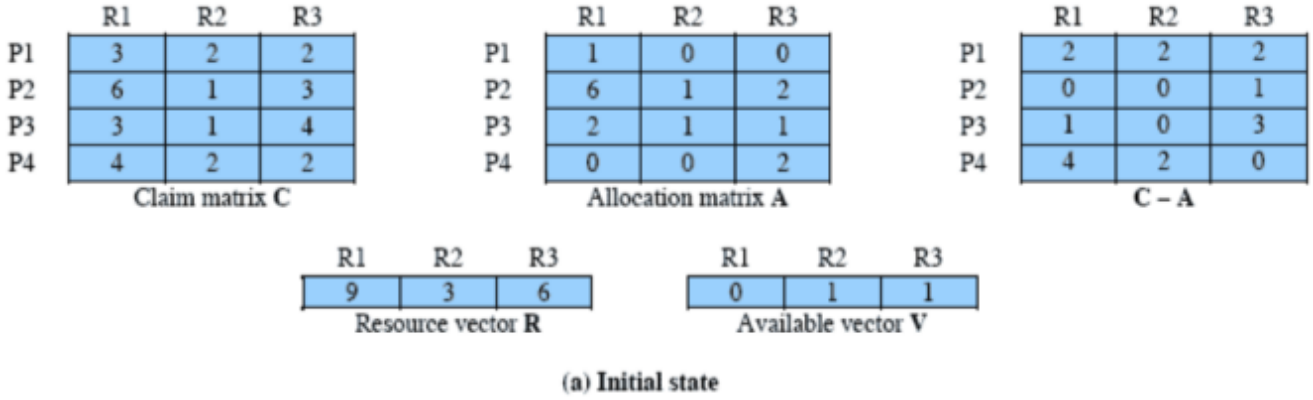
Legenda

- $m \rightarrow$ numero di diversi tipi di risorse
- $R_i \rightarrow$ numero di istanze dell'i-esima risorsa (effettivamente presente, non può essere modificata)
- $V_i \rightarrow$ numero di istanze disponibili per la i-esima risorsa
- $C_{ij} \rightarrow j$ come nei precedenti si riferisce alla risorsa, i invece si riferisce al numero di processi che voglio monitorare affinché non vadano in deadlock; con C_{ij} ci si riferisce a quante istanze della risorsa j verranno richieste (al massimo) dal processo i-esimo contemporaneamente
- $A_{ij} \rightarrow$ si riferisce a quante istanze della risorsa j sono state concesse al

processo i

- manca un vettore che indica che tipo di richiesta viene fatta

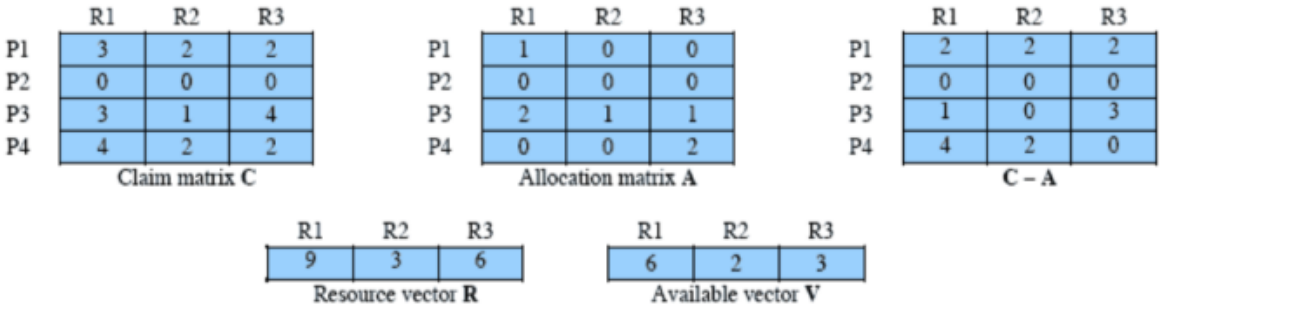
Determinazione dello stato sicuro



(a) Initial state

Per fare un iniziale sanity check dobbiamo verificare che per ciascun processo (ogni riga) della matrice C il numero di risorse richieste sia minore o uguale al numero di istanze per singola risorsa in R. V risulta essere la sottrazione tra un elemento del vettore R e la somma della colonna in A corrispondente. In $C - A$ ci stanno le istanze delle risorse che mi devono ancora esser richieste

Questo è uno stato sicuro (deve esistere almeno un cammino che non porta un deadlock). Supponiamo infatti di mandare in esecuzione `P2` fino alla fine; questo può terminare tutta la sua esecuzione in quanto, nonostante per terminare abbia bisogno di un'istanza di `R3`, questa gli può essere concessa in quanto ancora disponibile



(b) P2 runs to completion

Da questo punto è possibile completare anche tutti gli altri processi (il numero di istanze necessarie è maggiore di quelle disponibili), ma ciò non era possibile dallo stato iniziale (l'unico che poteva terminare dallo stato iniziale era `P2`)

	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

	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

Da questa situazione iniziale (sicura, mando in esecuzione `P2`), immaginiamo che l'algoritmo del banchiere fallisca e che quindi ci si ritrovi nella situazione descritta dall'immagine sottostante

	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

A questo punto ci si ritrova in uno stato non sicuro (nessun processo può essere

mandato in esecuzione)

Pseudocodice

Il seguente è lo pseudocodice per poter implementare l'algoritmo del banchiere

Strutture dati globali

```
struct state {
        int resource[m];  // R
        int available[m]; // V
        int claim[n][m];  // C
        int alloc[n][m];  // A
}
```

Algoritmo di allocazione di risorse

```
// un processo ha richiesto più istanze di quante ne aveva dichiare
in claim
if(alloc[i,*]+request[*] > claim[i,*]) // almeno un elemento maggiore
        <error>;                          // total request > claim
// non ci sono abbastanza risorse da allocare
else if(request[*] > available[*])
        <suspend process>;
// prima di concedere le risorse "simulo" il nuovo stato
else {
        <define newstate by:
        alloc[i,*] = alloc[i,*] + request[*];
        available[*] = available[*] - request[*]>;
}

if(safe(newstate)) {
        <carry out allocation>;
} else {
        <restore original state>;
        <suspend process>;
}

boolean safe(state S) {
        int currentavail[m];
        process rest[<number of processes>];
        currentavail = available;
        rest = {all processes};
        possibile = true;
        while(possible) {
```

```
                // itero sui processi e verifico se ce ne sta uno con
uno stato
                // successivo sicuro, se presente lo tolgo dalla
lista dei processi
                // e continuo
                <find a process Pk in rest such that
                claim[k, *]-alloc[k, *] <= currentavail;>
                if(found) {
                        currentavail = currentavail+alloc[k,*];
                        rest = rest - {Pk}
                } else possible = false;
        }
        // se non ci stanno processi in rest vuol dire che lo stato è
sicuro
        return (rest == null)
}
```

Come assunzione per il funzionamento di questo algoritmo, ci deve essere il fatto che i processi devono essere indipendenti, ovvero devono essere “liberi” di andare in esecuzione in qualsiasi ordine altrimenti non si può simulare l’esecuzione fino alla fine (l’unica sincronizzazione presente è proprio quella sulle richieste di risorse, non ci possono essere scambi di messaggi). Inoltre ci deve essere un numero fissato di risorse da allocare (non va bene per le risorse consumabili) e nessun processo deve terminare senza rilasciare le sue risorse

Rilevare il deadlock

Per rilevare il deadlock sono usate le stesse strutture dati dell’algoritmo del banchiere, ma la claim matrix è sostituita da `Q` che racchiude tutte le richieste effettuate da tutti i processi (come `request` del banchiere ma relativo a tutti i processi)

Algoritmo:

1. marca tutti i processi che non hanno allocato nulla (riga con tutti zero, non sono un problema in quanto non richiedono nulla)
2. $v \leftarrow V$
3. sia i un processo non marcato t.c. $Q_{ik} \leq w_k \quad \forall 1 \leq k \leq m$ (le sue risorse possono essere concesse)
4. se i non esiste, vai al passo 6
5. marca i e aggiorna $w \leftarrow w + A$; 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

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

- `P1` non può essere esaudito
- `P2` non può essere esaudito
- `P3` può essere esaudito (viene marcato) e viene liberato `R4` (in avalable avremo un 1 in `R4`)
  Nonostante questo cambiamento non può essere eseguito nessun processo, ci si trova in un deadlock

E poi?

Una volta rilevato un deadlock abbiamo diverse opzioni:

- **terminare** forzosamente tutti i processi coinvolti nel deadlock (usato spesso)
- mantenere dei **punti di ripristino**, ed effettuare il ripristino al punto precedente (ma lo stallo può verificarsi nuovamente)
- **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; 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

---

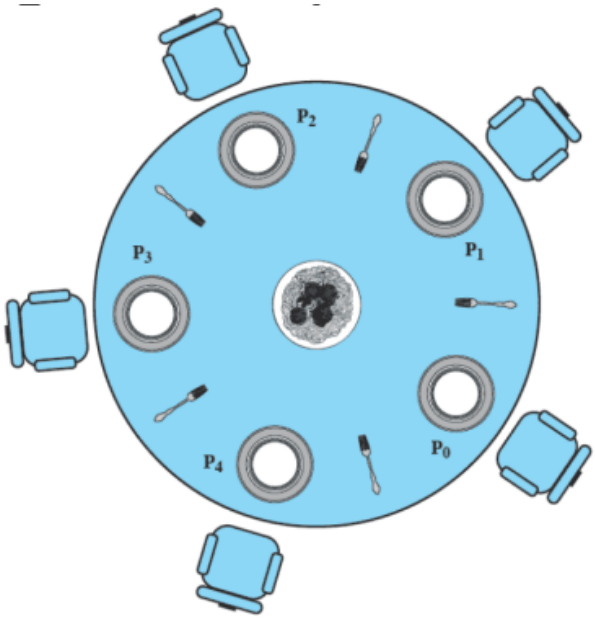
Deadlock e Linux

Come abbiamo visto in altri casi Linux cerca di essere il più minimale possibile a vantaggio dell'efficienza (non ha il long term scheduler) quindi se i processi utente sono "scritti male" e **possono andare in deadlock**, peggio per loro (saranno tutti bloccati e sta all'utente killarli).
Invece per quanto riguarda il kernel, c'è la **prevenzione dell'attesa circolare**

I filosofi a cena

Quello dei filosofi a cena è il problema per eccellenza collegato al deadlock

Ad un tavoli ci sono uno stesso numero di forchette, piatti e sedie per egual numero di filosofi che vogliono sedersi a mangiare. Il problema è che per mangiare hanno bisogno di due forchette e il filosofo può solamente prendere le forchette che stanno accanto al suo piatto (quella di destra e di sinistra); una volta finito di mangiare ripone le forchette



Ovviamente in questo problema due filosofi vicini non possono mangiare contemporaneamente. Dobbiamo far mangiare il maggior numero possibile di filosofi senza che ci sia deadlock

Prima soluzione

```
semaphore fork[5] = {1};

void philosopher(int i) {
        while(true) {
                think();
                wait(fork[i]);
                wait(fork[(i+1)%5]);
                eat();
                signal(fork[(i+1)%5]);
                signal(fork[i]);
        }
}

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

Per il filosofo i, `fork[i]` è la forchetta sinistra e `fork[(i+1)%n]` è la destra
In questo modo però ci può essere deadlock infatti lo scheduler può far si che ogni processo faccia la wait sulla forchetta di sinistra (nessuno viene bloccato) e poi fare la wait su quelle destre (tutti vengono bloccati, non ci sono più forchette disponibili)

Seconda soluzione

```
semaphore fork[5] = {1};
semaphore room = {4}

void philosopher(int i) {
        while(true) {
                think();
                wait(room)
                wait(fork[i]);
                wait(fork[(i+1)%5]);
                eat();
                signal(fork[(i+1)%5]);
                signal(fork[i]);
                signal(room)
        }
}

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

In questa soluzione si suppone che i filosofi pensino fuori dalla sala da pranzo e ci sia un cameriere che fa entrare a mangiare al massimo $n-1$ filosofi alla volta
Utilizzando l'esempio del deadlock di prima qui in ogni caso uno riesce a mangiare (niente deadlock)

Terza soluzione

Per risolvere il problema, nella soluzione precedente, sono stati leggermente cambiati i termini del problema (si impone ai filosofi di non poter stare tutti seduti a tavola). Da questa soluzione in poi risolveremo il problema senza deadlock e senza queste assunzioni

```
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(true) {
                think();
```

```
                wait(fork[first]);
                wait(fork[second]);
                eat();
                signal(fork[first]);
                signal(fork[second]);
        }
}
```

Nelle soluzioni precedenti si faceva in modo che si prendesse prima la forchetta di sinistra e poi la destra, qui invece tutti prendono prima la sinistra e poi la destra eccetto uno che le prende al contrario (l'ultimo) e ciò è sufficiente a risolvere il deadlock. Infatti, considerando l'esempio di prima, l'ultimo prova a prendere la seconda forchetta al posto della prima ma verrà bloccato in quanto è occupata, lasciando quindi la possibilità al primo filosofo di prendere la seconda forchetta

Quarta soluzione (sbagliata)

```
mailbox fork[N];

// rendo tutte le forchette prendibili
init_forks() {
        int i;
        for(i=0; i<N; i++) {
                nbsend(fork[i], "fork");
        }
}

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(true) {
                think_for_a_while();
                receive(fork[first], fork1);
                receive(fork[second], fork2);
                eat();
                nbsend(fork[first], fork1);
                nbsend(fork[second], fork2)
        }
}
```

Quinta soluzione

```
philosopher(int me) {
        int left, right;
        message fork1, fork2;
        left = me;
        right = (me+1)%N;

        while(true) {
                think_for_a_while();
                // provo a prendere la forchetta di sinistra
                if(nbreceive(fork[left], fork1)) {
                        // provo a prendere la forchetta di destra
                        if(nbreceive(fork[right], fork2)) {
                                eat();
                                nbsend(fork[right], fork1);
                        }
                nbsend(fork[left], fork2)
                }
        }
}
```

In questa soluzione però, con il fatto che i processi non possono essere bloccati, ci si può imbattere nel livelock. Infatti se tutti i processi prendono la forchetta di sinistra e poi vengono bloccati si entra in un circolo vizioso in cui continuando a prendere e rilasciare la forchetta di sinistra però senza mai poter prendere la destra in quanto risulta occupata (prima soluzione)
Nonostante ciò questa soluzione è accettabile in quanto si basa sul fatto che i filosofi “pensino” per un tempo casuale, scorrelato da quello degli altri