Sistemi Operativi e Reti di Calcolatori (SOReCa)

Corso di Laurea in *Ingegneria Informatica e Automatica (BIAR)*
Terzo Anno | Primo Semestre
A.A. 2024/2025

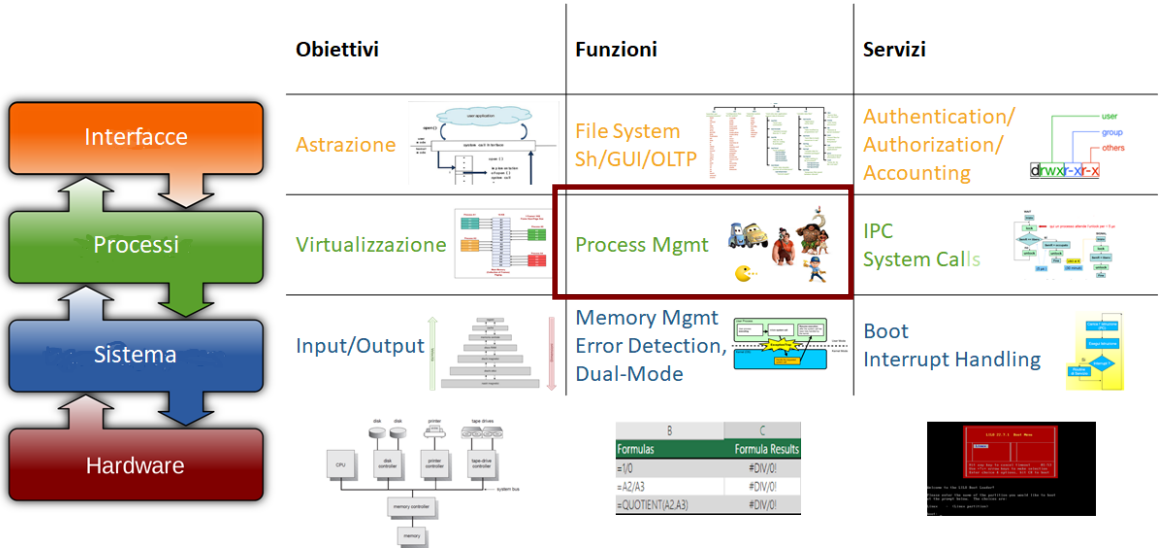
Deadlock

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Sistemi operativi (3 CFU)

- Il sistema operativo
- Concorrenza e sincronizzazione
- Deadlock
- Inter-process communication (IPC)
- Scheduling
- Memoria centrale e virtuale
- Memoria di massa e File system
- Sicurezza informatica



Lezioni: Settembre - Ottobre

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

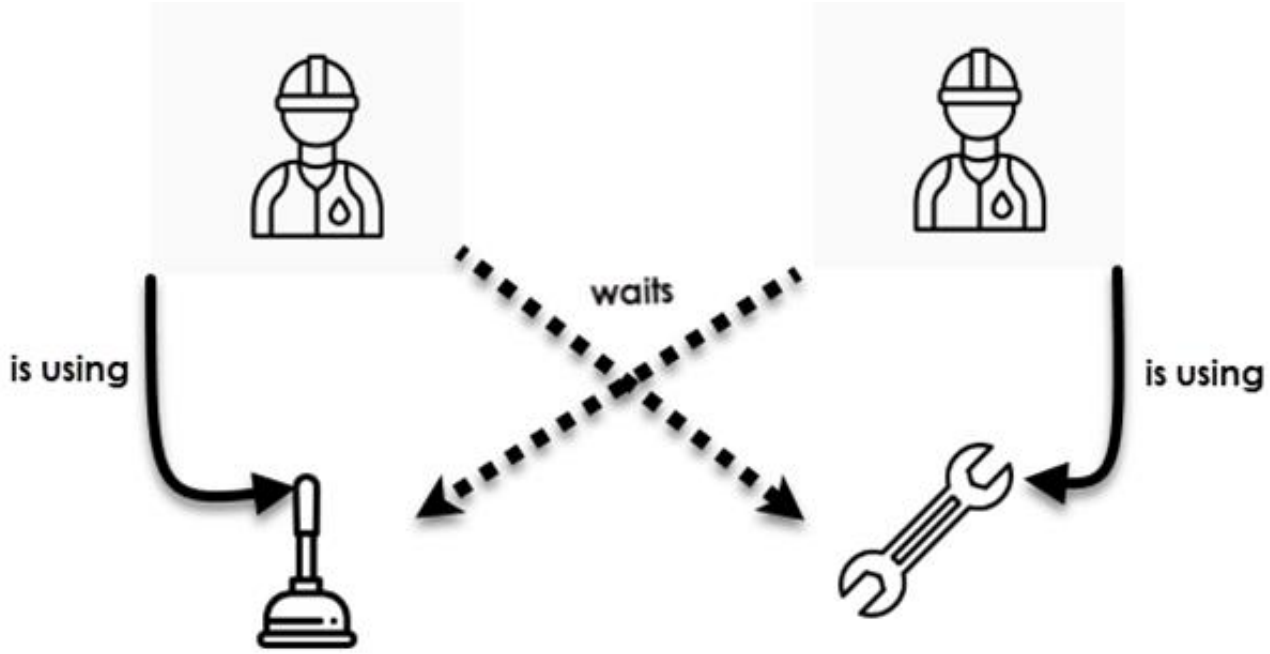
DeadLock (Stallo)

Operating Systems: Deadlock

Deadlock (Stallo)

➔ **Stallo**: Situazione derivante dalla necessità di:

- utilizzo di 2 o più risorse
- da parte di 2 o più processi.

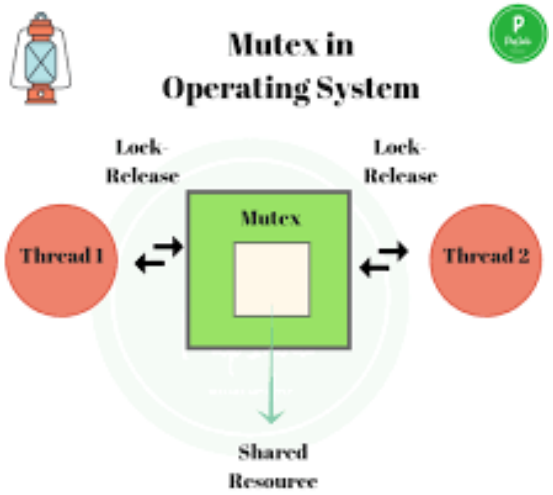


Un sistema offre ai processi le risorse di sistema:

- in **numero finito** (es. Cicli di CPU, spazio di memoria, periferiche di I/O, ecc..)
- Molte sono classificabili in **classi**, poiché identiche fra loro (es. SMP, pagine di memoria, porte USB, etc)
- Condivisibili: in modo **mutuamente esclusivo** o **non-esclusivo**
- **Preemptable** (requisibile) o **non-Preemptable** (non requisibile)

Operating Systems: Deadlock

Risorse Condivise: utilizzo 1/3



➔ **Uso di Risorse Condivise**: Un processo segue il seguente schema per utilizzare una risorsa:

- Richiesta di uso della risorsa
- Uso della risorsa
- Rilascio della risorsa

La richiesta e il rilascio delle risorse avvengono attraverso chiamate di sistema (e.g. acquire() , release())

Il sistema operativo mantiene una tabella che memorizza lo stato delle risorse e l'eventuale processo utilizzatore

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Deadlock

Risorse Condivise: utilizzo 2/3


```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
        down(&resource_1);
        use_resource_1( );
        up(&resource_1);
}
```

(a)

(a) Una risorsa

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}
```

(b)

(b) Due risorse

Acquisizione delle Risorse: utilizzando un semaforo per proteggere le risorse

Operating Systems: Deadlock

Risorse Condivise: utilizzo 3/3


```
typedef int semaphore;
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}

void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
}
```

(b)

Ordine invertito di richiesta/rilascio delle risorse fra i 2 processi

(a) Codice esente da Deadlock

(b) Codice con Deadlock potenziale

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

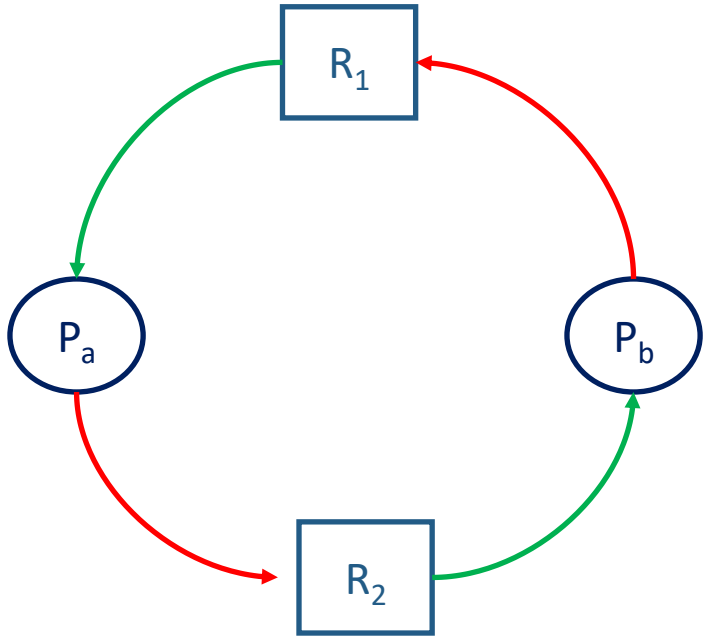
Operating Systems: Deadlock

Deadlock (Stallo): Definizione

➔ **Stallo**: un gruppo di processi entra in uno stallo quando tutti i processi del gruppo attendono il rilascio di una risorsa che può essere liberata solo da uno dei processi in attesa. Esempio con due processi (P_a e P_b) e due risorse (R_1 e R_2)

- Il processo P_a ottiene il possesso della risorsa R_1
- Il processo P_b ottiene il possesso della risorsa R_2
- Il processo P_a richiede la risorsa R_2

➔ Il processo P_b richiede la risorsa R_1

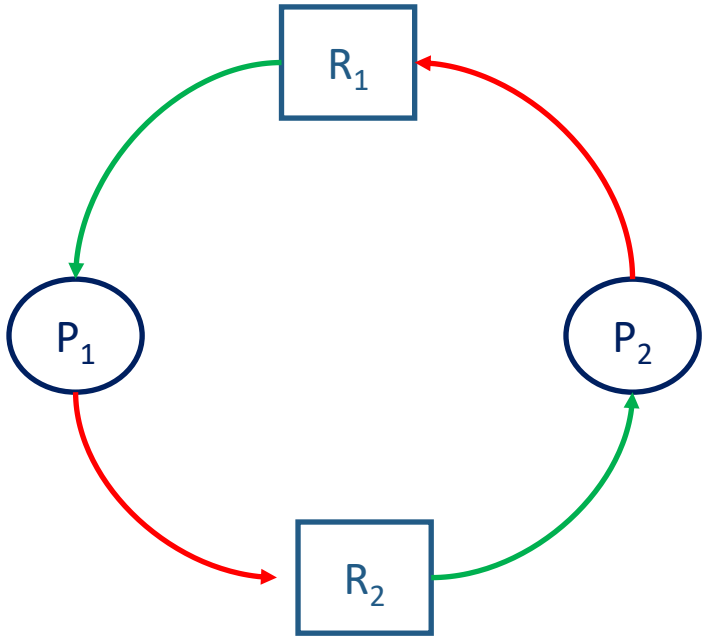


Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Deadlock

Deadlock (Stallo): Caratterizzazione



Si ha deadlock se si verificano simultaneamente le seguenti condizioni (necessarie):

- **Mutua esclusione**: Almeno una risorsa deve poter essere acceduta da un solo processo alla volta (gli altri vengono messi in attesa)
- **Possesso e attesa**: Un processo possiede una risorsa ed è in attesa per un'altra risorsa
- **Non-preemptive**: Una risorsa posseduta da un processo non può essere rilasciata se non per

spontanea volontà del processo stesso

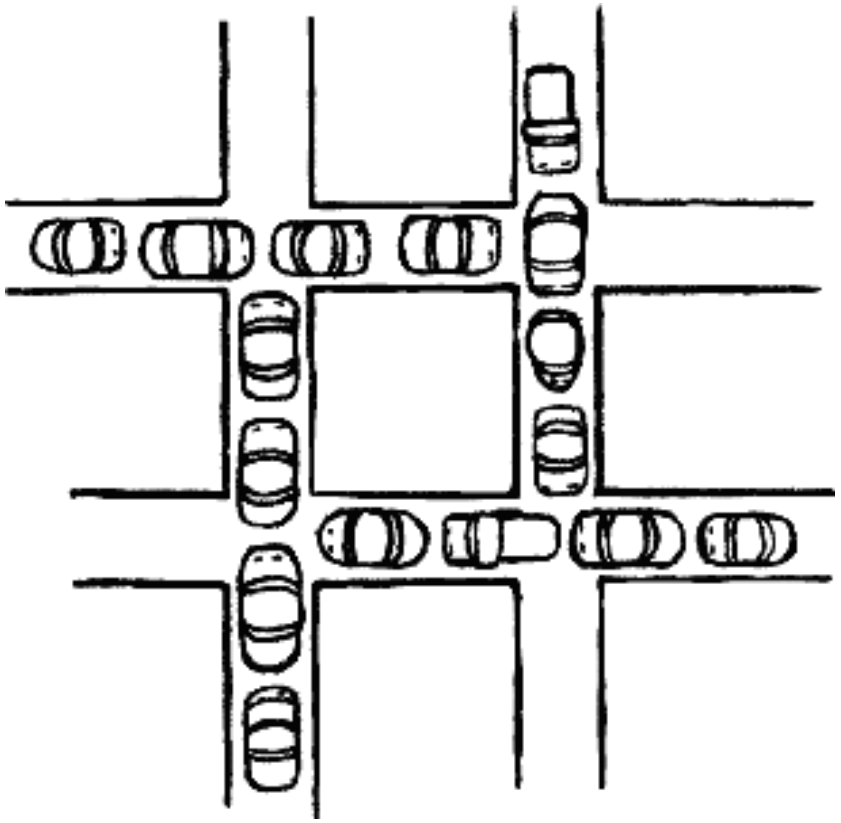
- **Attesa circolare**: $\{P_0 , P_1 , \ldots, P_N \}$, P_0 attende una risorsa posseduta da P_1 , P_1 attende una risorsa posseduta da P_2 , …, P_N attende una risorsa posseduta da P_0

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Deadlock

Deadlock (Stallo): Propagazione



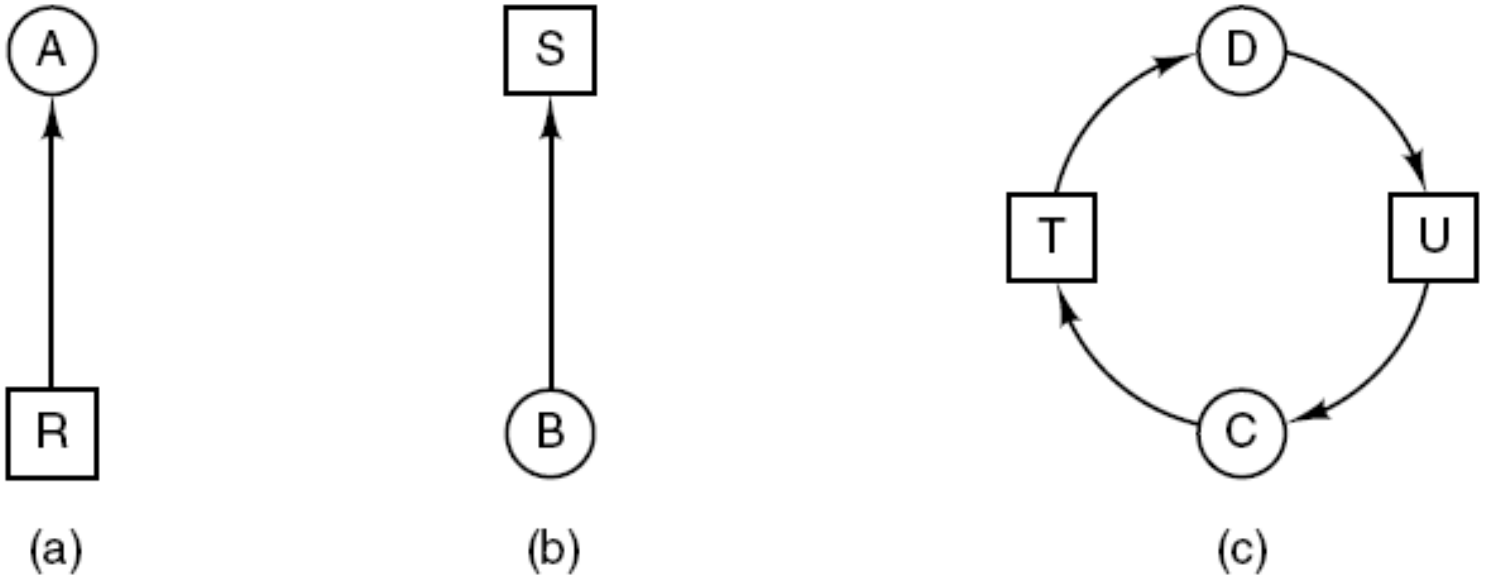
Nuovi processi possono via via entrare indefinitamente in tale stato se le risorse richieste sono in possesso di altri processi a loro volta in stallo

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Deadlock

Modelizzazione del Deadlock: Resource Allocation Graph



Resource Allocation Graph (Petri Network semplificate):

(a) A possiede la risorsa R

(b) B richiede la risorsa S

(c) D possiede la risorsa T e richiede la risorsa U

(d) C possiede la risorsa U e richiede la risorsa T

➔ **Deadlock**

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Deadlock

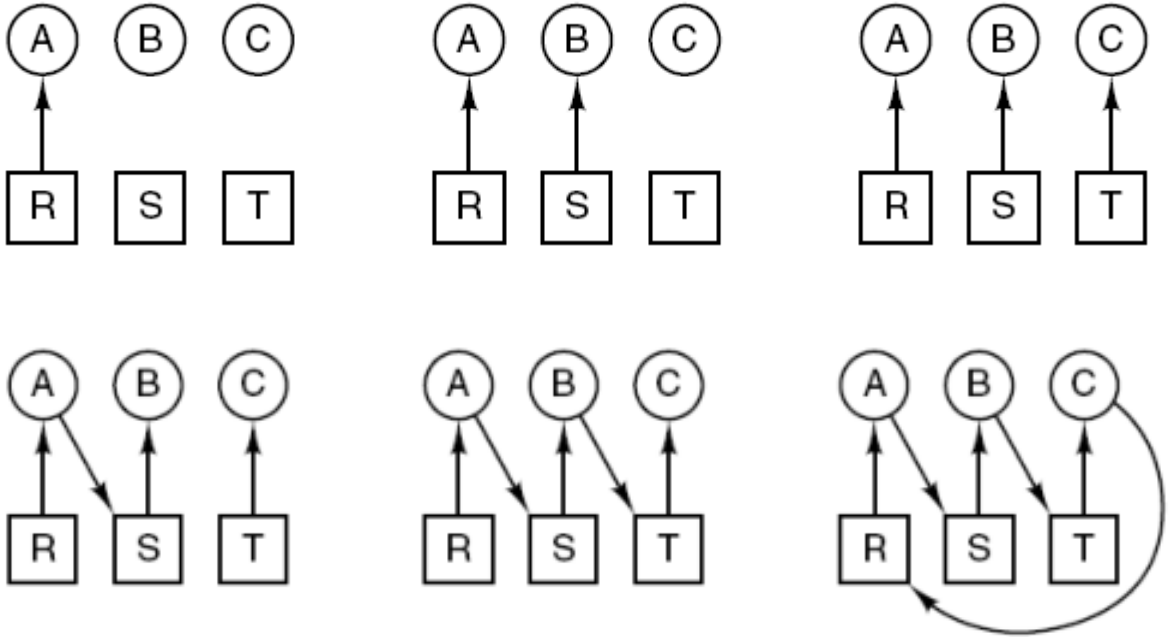
Deadlock: Resource Allocation Graph esempio 1/2

Computazione dei Processi

A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

Svolgimento Lineare ➔ Deadlock

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock



Operating Systems: Deadlock

Deadlock: Resource Allocation Graph esempio 2/2

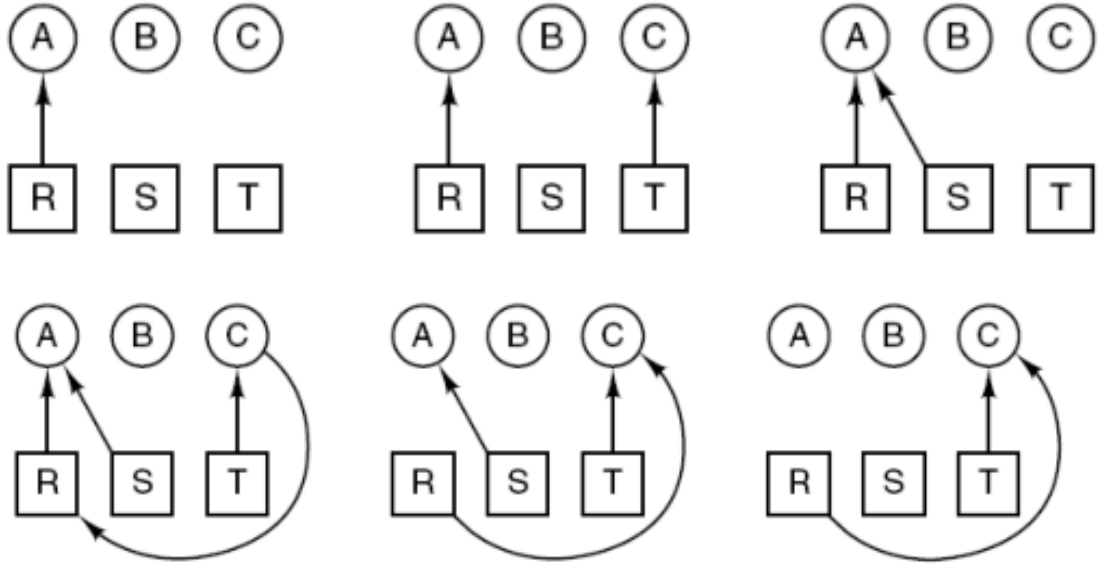
Computazione dei Processi

A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

Processo Inibito (B) ➔ Deadlock evitato

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



Operating Systems: Deadlock

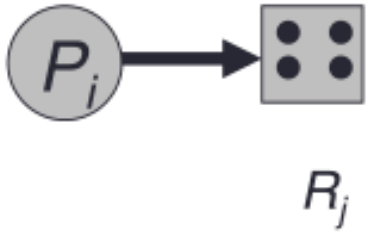
Deadlock: Grafo di Allocazione delle Risorse 1/3

- Grafo di allocazione delle risorse G (V,E):
  - Insieme di nodi V
  - Insieme di archi E
- Nodi:
  - processi del sistema $P = \{P_1, P_2, \ldots, P_n\}$
  - risorse del sistema $R = \{R_1, R_2, \ldots, R_m\}$ eventualmente con più istanze identiche
- Archi:
  - arco di richiesta: da processo a risorsa $P_i \rightarrow R_j$
  - arco di assegnazione: da risorsa a processo $R_j \rightarrow P_i$

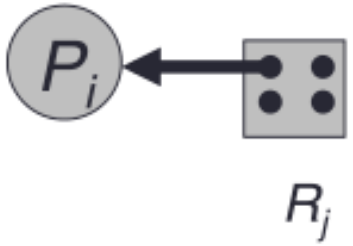
- Processo

- Tipo di risorsa con 4 istanze

- Arco di richiesta (si noti che raggiunge la risorsa)

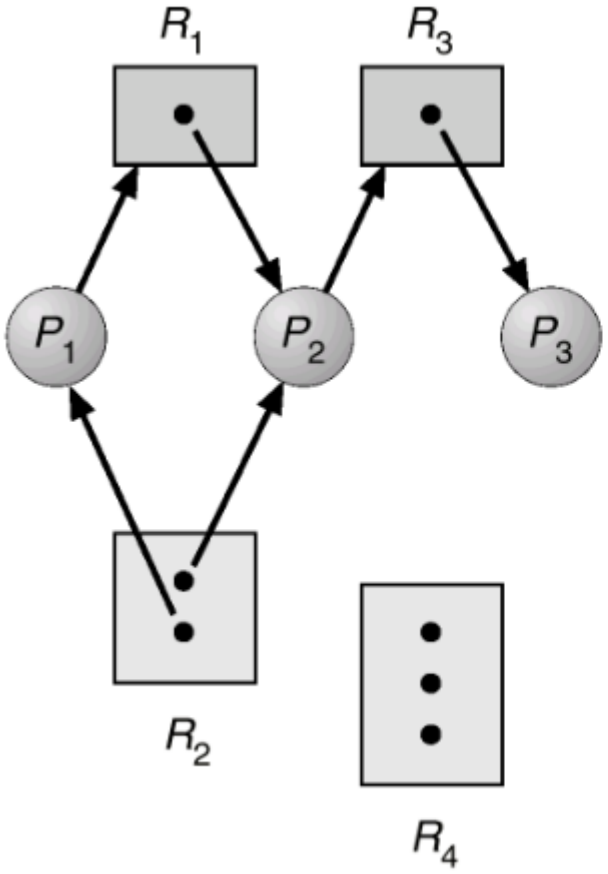


- Arco di assegnazione (si noti che parte dall'istanza)

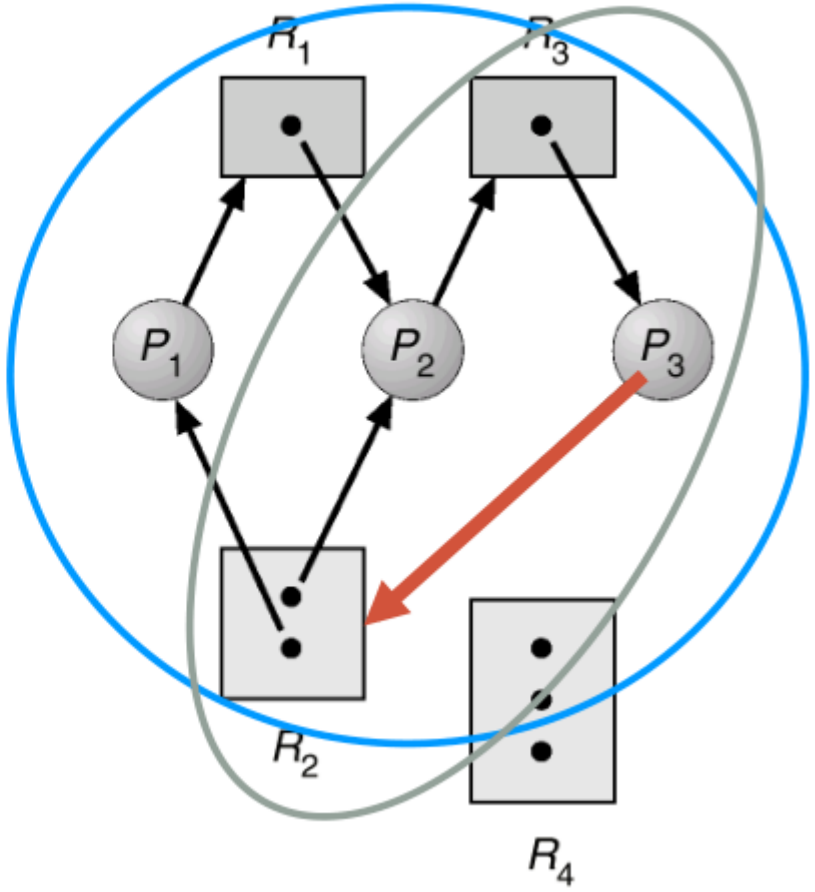


Operating Systems: Deadlock

Deadlock: Grafo di Allocazione delle Risorse 2/3



(a) No Deadlock



(b) Due cicli

Se il grafo **non** contiene cicli ⇒ nessun deadlock

Se il grafo contiene cicli ⇒

(a) se ogni tipo di risorsa ha più di un'istanza, allora si ha una possibilità di deadlock (Condizione necessaria ma non sufficiente)

(b) se ogni tipo di risorsa inclusa nel ciclo ha una sola istanza, allora si ha un deadlock (Condizione necessaria e sufficiente)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

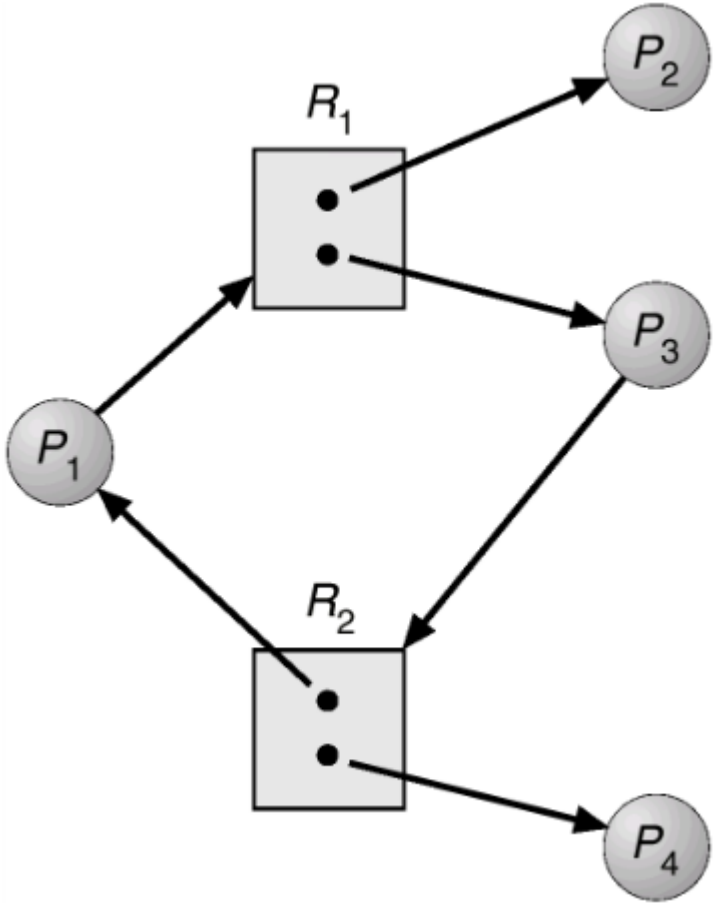
Operating Systems: Deadlock

Deadlock: Grafo di Allocazione delle Risorse 3/3

Se il grafo **non** contiene cicli ⇒ nessun deadlock

Se il grafo contiene cicli ⇒

- P_4 prima o poi cederà il possesso di R_2
- Che potrà quindi essere acquisita da P_3
- Non siamo in una condizione di deadlock



(c) Deadlock “apparente”

Operating Systems: Deadlock

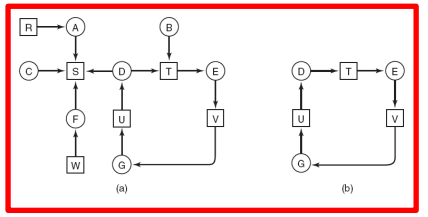
Deadlock: Metodo di Gestione

- **Ignorare** il deadlock: giustificato dalla scarsa occorrenza di questa evenienza. Richiede intervento manuale dell'utente.

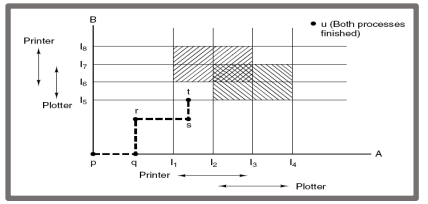
- **Prevenire** il deadlock (deadlock prevention): evitare che si verifichino tutte e quattro le condizioni necessarie. Descrivono come le risorse devono essere richieste.

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

- **Rilevare e recuperare** il deadlock (deadlock detection & recovery).



- **Evitare** il deadlock (deadlock avoidance): Il sistema operativo conosce in anticipo quali risorse un processo utilizzerà. In base a queste informazioni decide se tutte le richieste possono essere accettate senza causare stalli.



Nessun metodo è ottimale, meglio una loro combinazione a seconda della "classe di risorsa".

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Deadlock

Deadlock: Ignorare 1/2

Algoritmo dello struzzo

- Applicato dalla maggior parte dei SO odierni (e dalla JVM)
- È più economico dei metodi che prevedono di prevenire, evitare o individuare gli stalli

Linux: gestione minimale ed efficiente:

- **User Space**: processi utente possono andare in deadlock. Amministrati dall'utente. Riduzione di:
  - Mutual Esclusion ➔ Spooling
  - Hold & Wait ➔ massimizzare le risorse accessibili in contemporanea (no lock)

- **Kernel Space**: prevenzione dell'attesa circolare (lock acquisiti in ordine fisso e predeterminato).

Operating Systems: Deadlock

Deadlock: Ignorare 2/2

Algoritmo dello struzzo

- Applicato dalla maggior parte dei SO odierni (e dalla JVM)
- È più economico dei metodi che prevedono di prevenire, evitare o individuare gli stalli

Windows: i deadlock richiedono un certo grado di «bad luck», poiché richiedono che un certo numero di eventi accadano simulaneamente (cfr. Windows Hardware Developer – Deadlock Detection)

- **User Space**: processi utente possono andare in deadlock. Amministrati dall'utente.

- **Driver Verifier**: analisi statica del codice sorgente del driver, in modo da identificare eventuali deadlock potenziali (da Windows XP, 2001)

Operating Systems: Deadlock

Deadlock: Prevenire 1/5

Prevenire il deadlock (deadlock prevention): evitare che si verifichino tutte e quattro le condizioni necessarie.

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

- **Mutua esclusione**: Almeno una risorsa deve poter essere acceduta da un solo processo alla volta (gli altri vengono messi in attesa)

- **Possesso e attesa**: Un processo possiede una risorsa ed è in attesa per un'altra risorsa

- **Non-preemptive**: Una risorsa posseduta da un processo non può essere rilasciata se non per spontanea volontà del processo stesso

- **Attesa circolare**: $\{P_0 , P_1 , \ldots, P_N \}$, P_0 attende una risorsa posseduta da P_1 , P_1 attende una risorsa posseduta da P_2 , …, P_N attende una risorsa posseduta da P_0

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

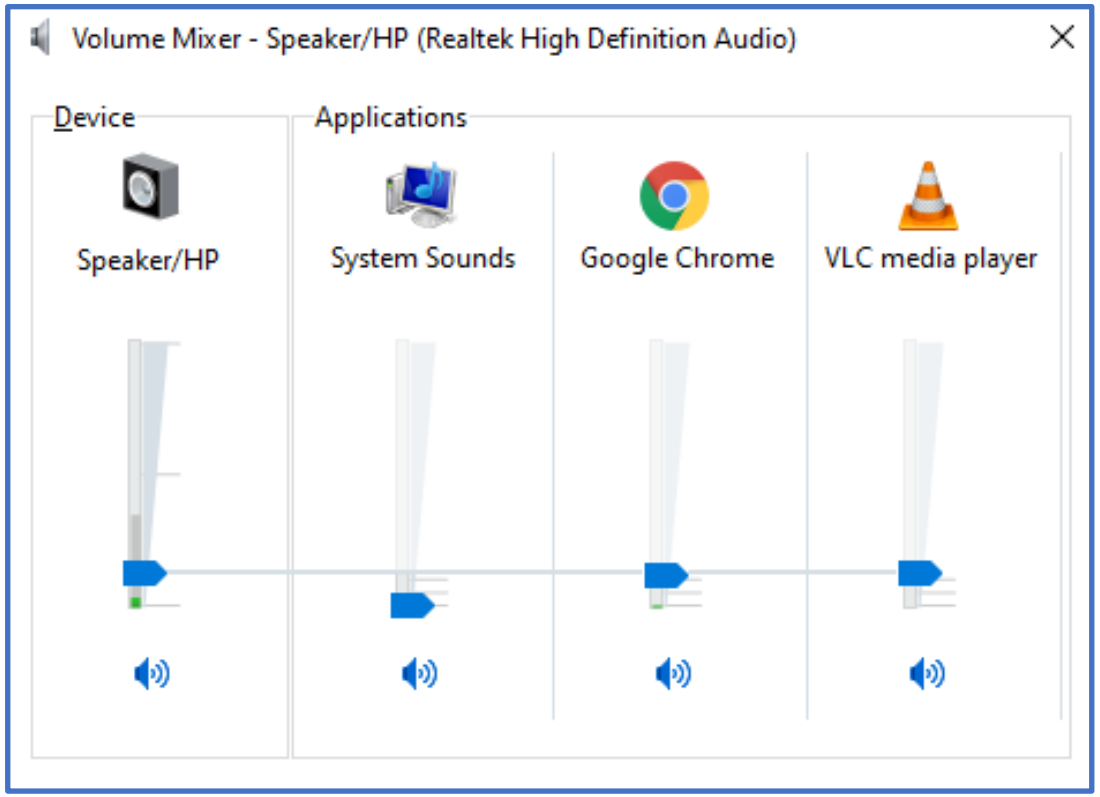
Operating Systems: Deadlock

Deadlock: Prevenire 2/5

Mutua Esclusione

- La condizione può essere invalidata rimuovendola per le risorse intrinsecamente condivisibili

- La condizione non può mai essere invalidata per le risorse intrinsecamente non condivisibili . Per questo motivo tipicamente non si cerca di prevenire i deadlock negando la condizione di mutua esclusione



Spooling del device audio

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Deadlock

Deadlock: Prevenire 3/5

Possesso ed Attesa

La condizione può essere invalidata garantendo che ogni volta che un processo chiede risorse, non possegga già qualche altra risorsa.
Due possibili tecniche:

1. Un processo chiede e ottiene tutte le risorse prima di iniziare l'esecuzione

2. Un processo che possiede alcune risorse e vuole chiederne altre deve:
- rilasciare tutte le risorse che possiede
- chiedere tutte quelle che servono (incluse eventualmente anche alcune di quelle che già possedeva

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Deadlock

Deadlock: Prevenire 4/5

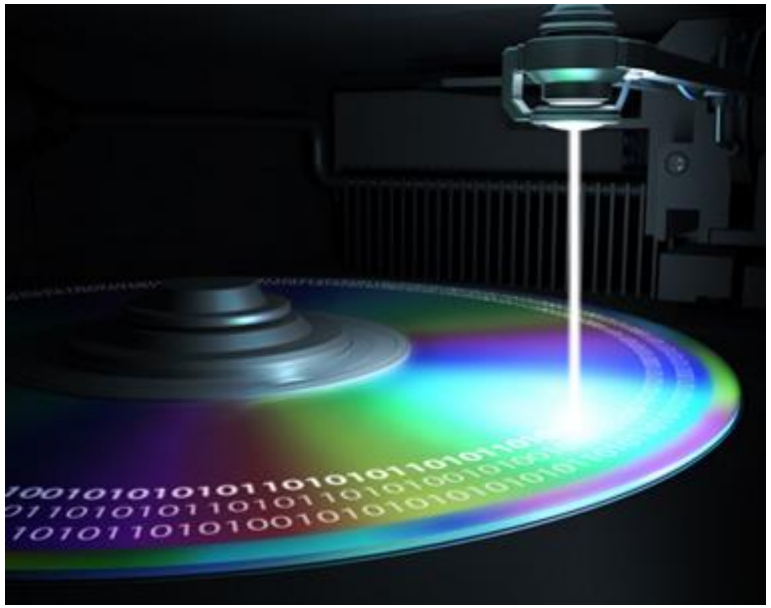
Non-preemptive

Si impone ad un processo che è in attesa di alcune risorse di rilasciarne alcune tra quelle che già possiede.
In questo modo altri processi riescono ad acquisire tutte le risorse di cui necessitano

- Applicabilità:

OK: per risorse il cui stato può essere facilmente salvato e ricaricato (es. Registri CPU e memoria centrale)

NOK: Non va bene ad esempio per nastri o stampanti, Burning



Operating Systems: Deadlock

Deadlock: Prevenire 5/5

Attesa Circolare

Per evitare l'attesa circolare, le risorse sono ordinate gerarchicamente.

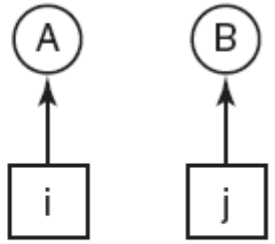
Ogni processo deve chiedere le risorse in ordine incrementale.

Un ordinamento globale univoco viene imposto su tutti i tipi di risorsa R_i

- Si implementa tramite una funzione $f(R_i) = n$
- Se un processo chiede k istanze della risorsa R_j e detiene solo risorse R_i con $i<j$,
- se le k istanze della risorsa R_j sono disponibili gli vengono assegnate
- altrimenti, il processo deve attendere
- Un processo non potrà mai chiedere istanze della risorsa R_j se detiene risorse R_i con $i \geq j$

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD-ROM drive

(a)



(b)

(a) Risorse Ordinate Numericamente.
(b) Grafo delle Risorse.

➔ **I programmatori devono scrivere i programmi in modo che l'ordinamento sia rispettato**

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: Deadlock

Deadlock: Rilevare e Recuperare 1/5

Rilevare e recuperare (deadlock detection & recovery)

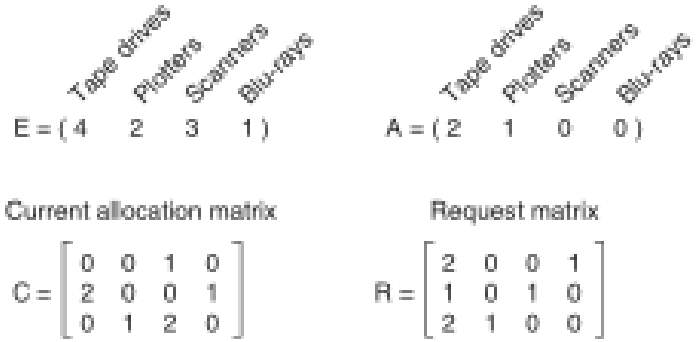
- **Rilevare** la presenza di situazioni di deadlock dopo che sono avvenute.
  - Rilevazione con **istanze singole** delle risorse
  - Rilevazione con **istanze multiple** delle risorse

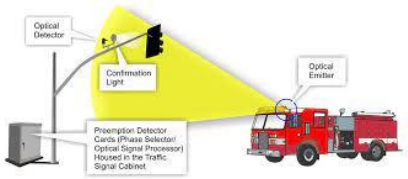
- **Recuperare** una situazione di corretto funzionamento eliminando il deadlock

  - **Preemption**: Prelazione delle Risorse

  - **Rollback**: annullare le esecuzioni effettuate da alcuni processi coinvolti nel Deadlock (utilizzo dei **Checkpoint**)

  - **Killing**: terminazione di alcuni processi coinvolti nel Deadlock





Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

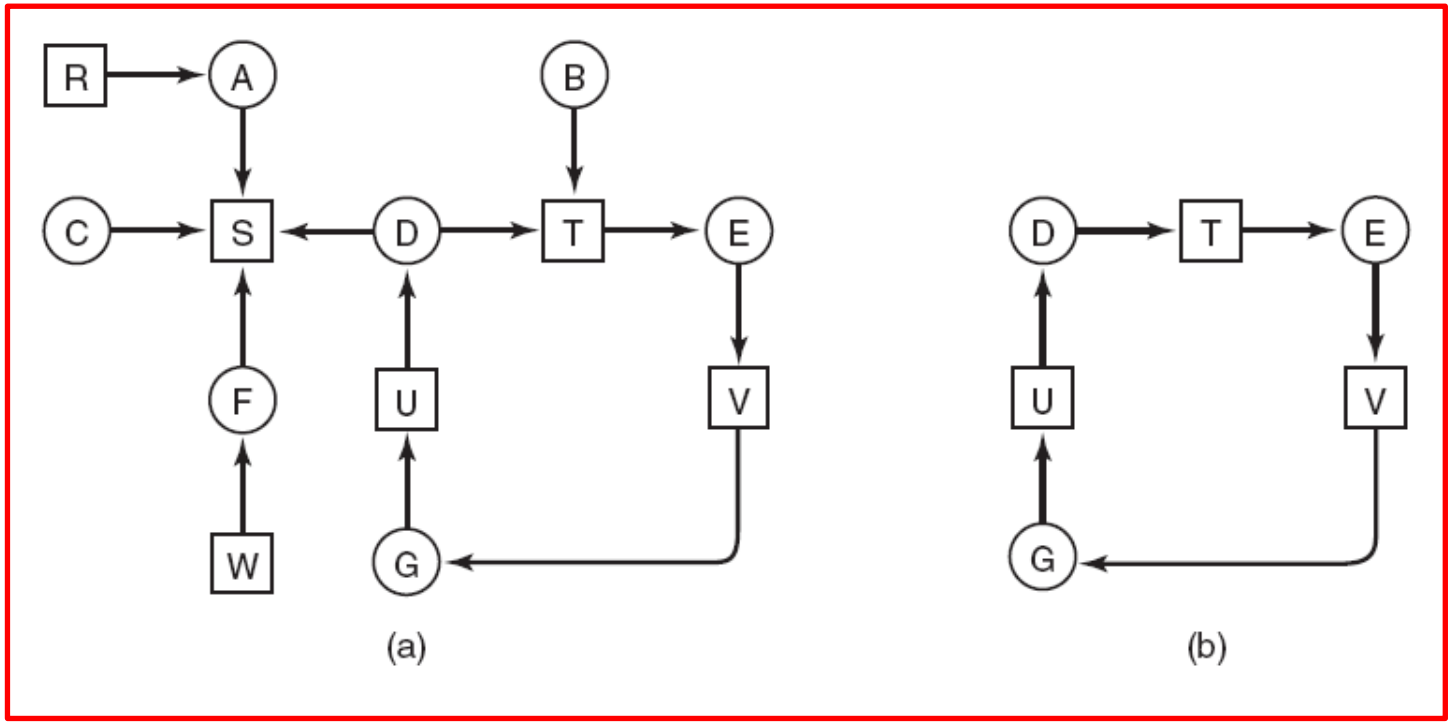
Sapienza Università di Roma

Operating Systems: Deadlock

Deadlock: Rilevare e Recuperare 2/5

Istanza Singola

Algoritmo per il rilevamento del deadlock: per ogni nodo, N nel grafico, eseguire i seguenti cinque passaggi con N come nodo iniziale.



1. Inizializzare L alla lista vuota, designare tutti gli archi come non marcato.
2. Aggiungere il nodo corrente alla fine di L, controlla se nodo ora appare in L due volte. Se lo fa, il grafico contiene un ciclo (elencato in L), algoritmo termina.
3. Da un dato nodo, vedi se ci sono archi in uscita non contrassegnato. In tal caso, andare al passaggio 4; in caso contrario, vai al passaggio 5.
4. Scegli un arco in uscita non contrassegnato a caso e segnalo. Quindi seguilo fino al nuovo nodo corrente e vai al passaggio 3.
5. Se questo è il nodo iniziale, il grafico non ne contiene nessuno cicli, l'algoritmo termina. Altrimenti, vicolo cieco. Rimuovilo, torna al nodo precedente, crea un nodo corrente, vai al passaggio 2.

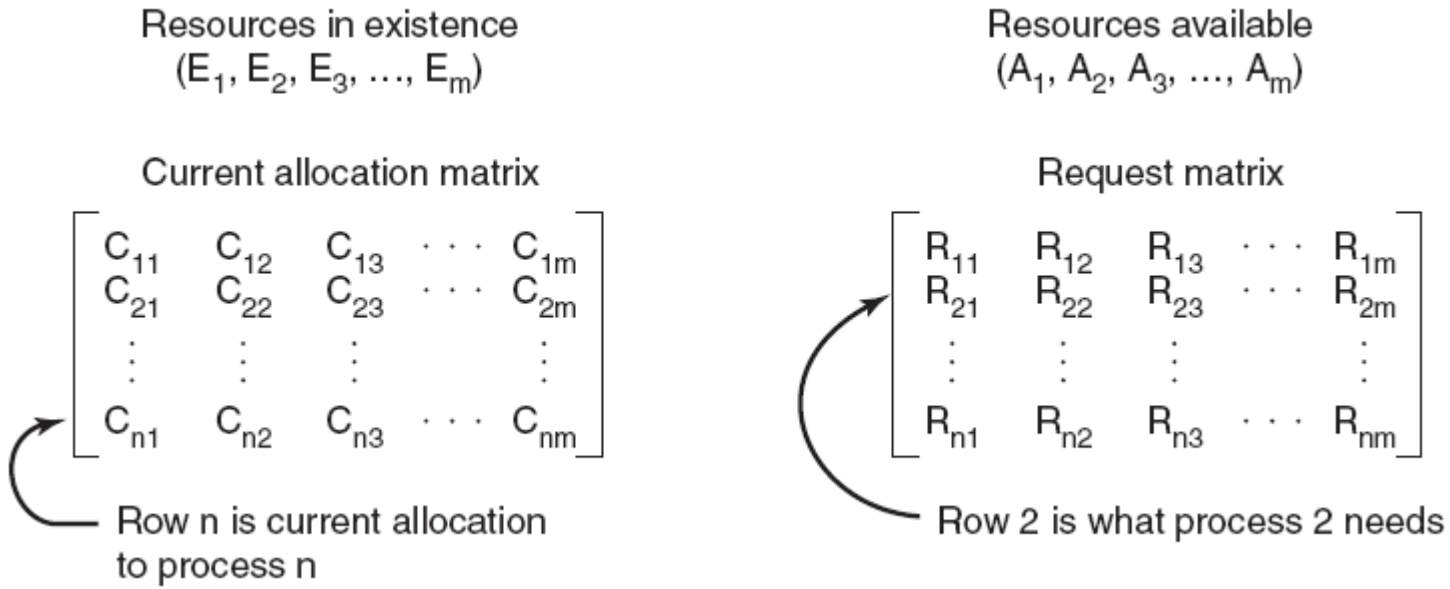
Operating Systems: Deadlock

Deadlock: Rilevare e Recuperare 3/5

Istanze Multiple

Algoritmo che presuppone lo stato peggiore: tutti i processi conservano tutte le risorse acquisite fino alla loro uscita. Man mano che l'algoritmo progredisce, i processi verranno contrassegnati, indicando che sono in grado di completarsi e quindi non sono in deadlock. Quando l'algoritmo termina, è noto che tutti i processi non contrassegnati sono in deadlock.

1. Cerca un processo non marcato, P i , per il quale l'i-esima riga di R è minore o uguale ad A.
2. 2. Se viene trovato un tale processo, aggiungere l'i-esima riga di C ad A, contrassegnare il processo e tornare al passaggio 1.
3. 3. Se non esiste tale processo, l'algoritmo termina.



Le quattro strutture dati necessarie dall'algoritmo di rilevamento del deadlock. Avendo:

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Deadlock

Deadlock: Rilevare e Recuperare 4/5

Istanze Multiple (esempio)

Per eseguire l'algoritmo di rilevamento del deadlock, cerchiamo un processo la cui richiesta di risorse può essere soddisfatta.

P_1 non può essere soddisfatto perché non è disponibile un'unità Blu-ray.

P_2 neanche può essere soddisfatto, perché non c'è uno scanner disponibile.

Fortunatamente, P_3 può essere soddisfatto, avendo

A = (4 2 3 1)

quindi viene eseguito e alla fine restituisce tutte le sue risorse, dando

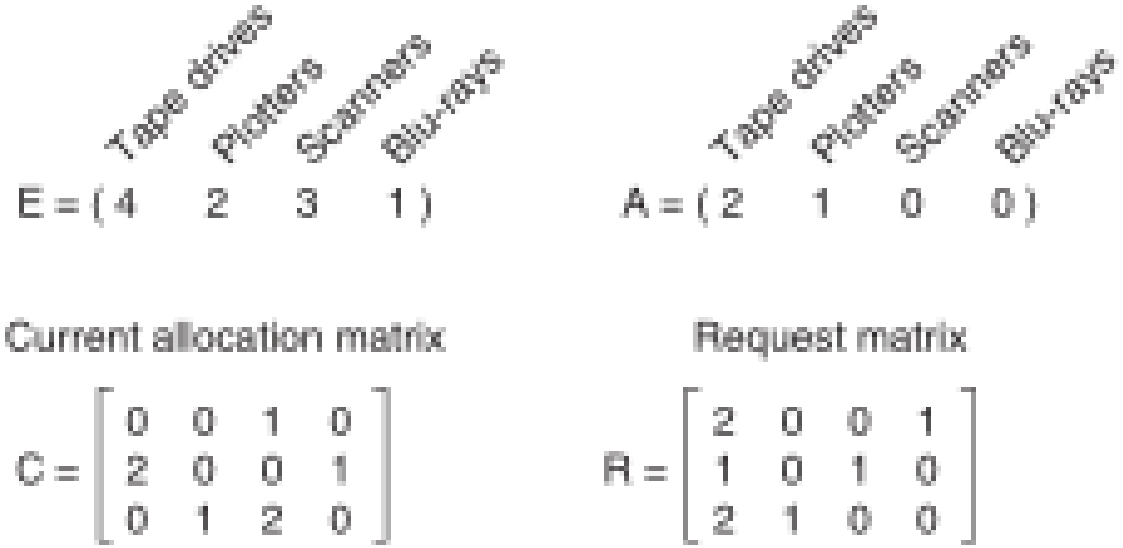
A = (2 2 2 0)

A questo punto, P_2 può essere eseguito e restituire le sue risorse, dando

A = (4 2 2 1)

Ora il rimanente P_1 può essere eseguito.

Non c'è stallo nel sistema.



Le quattro strutture dati necessarie dall'algoritmo di rilevamento del deadlock. Avendo:

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

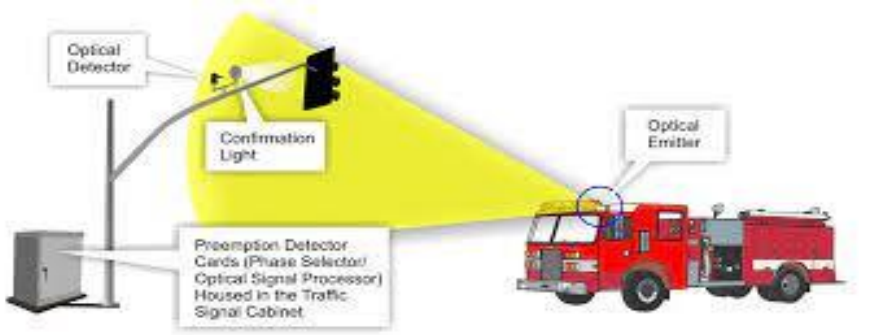
Sapienza Università di Roma

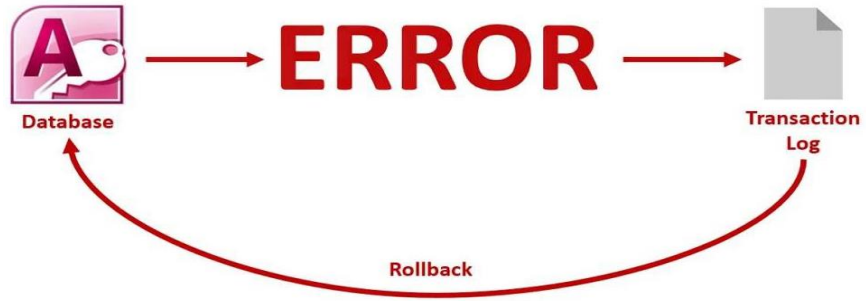
Operating Systems: Deadlock

Deadlock: Rilevare e Recuperare 5/5

Rilevare e recuperare (deadlock detection & recovery)

- **Recuperare** una situazione di corretto funzionamento eliminando il deadlock
  - **Preemption**: Prelazione delle Risorse
  - **Rollback**: annullare le esecuzioni effettuate da alcuni processi coinvolti nel Deadlock (utilizzo dei **Checkpoint**)
  - **Killing**: terminazione di alcuni processi coinvolti nel Deadlock



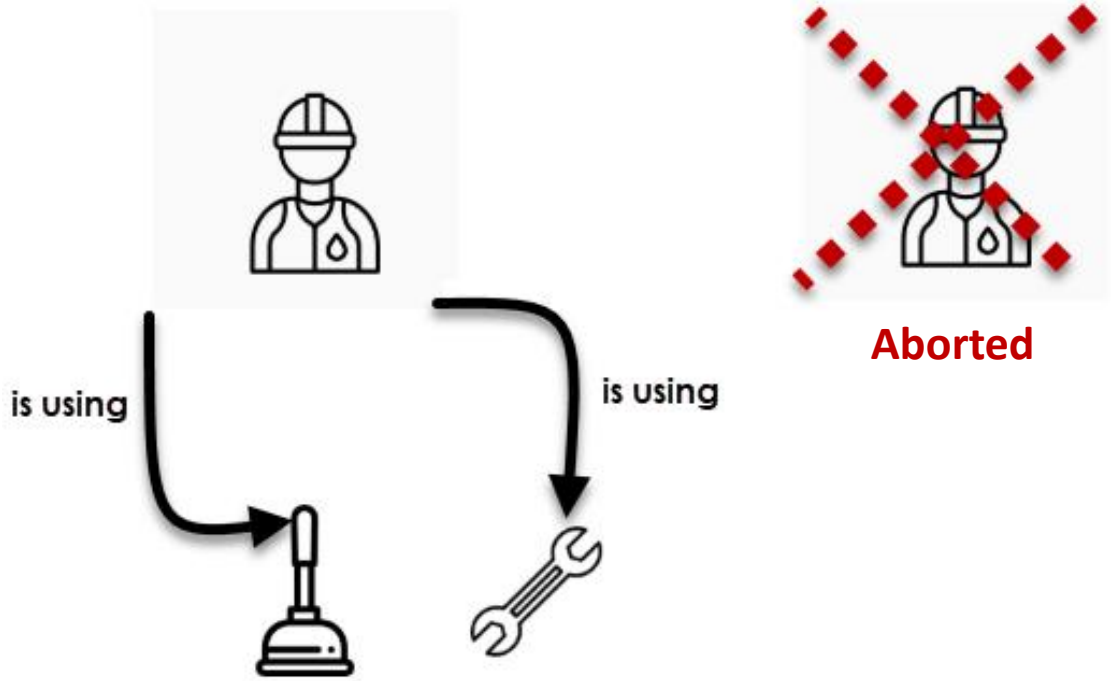



```
C:\Users\A705945>taskkill /?

TASKKILL [/S system [/U username [/P [password]]]]
         { [/FI filter] [/PID processid | /IM imagename] } [/T] [/F]

Description:
    This tool is used to terminate tasks by process id (PID) or image name.

Parameter List:
    /S    system           Specifies the remote system to connect to.

    /U    [domain\]user    Specifies the user context under which the
                           command should execute.

    /P    [password]       Specifies the password for the given user
```

Operating Systems: Deadlock

Deadlock: Evitare 1/7

- ➔ **Evitare lo Stallo**: sequenzializzare l'esecuzione dei processi (almeno quelli che potrebbero essere interessati allo stesso insieme di risorse).
- ➔ Conoscere <u>in anticipo</u> l'insieme di <u>risorse</u> usate da ciascun processo



Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Deadlock

Deadlock: Evitare 2/7

Evitare il deadlock (deadlock avoidance): Verificare a priori se la sequenza di richieste e rilasci di risorse effettuate da un processo porta al deadlock, tenendo conto delle risorse già assegnate ai processi già accettati nel sistema

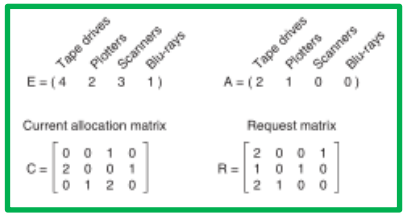
Obiettivi:

- Alto sfruttamento delle risorse
- Alta efficienza del sistema
- Semplicità di gestione

Necessità di informazioni a priori sullo stato di allocazione delle risorse e sul comportamento dei processi:

- **E** (**Existing**): max numero massimo di risorse per ogni processo
- **A** (**Available**): risorse disponibili
- **P** (**Possessed**): risorse assegnate
- richieste e rilasci futuri di risorse

Stato di allocazione
(E, A, P)
Per ogni processo



DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

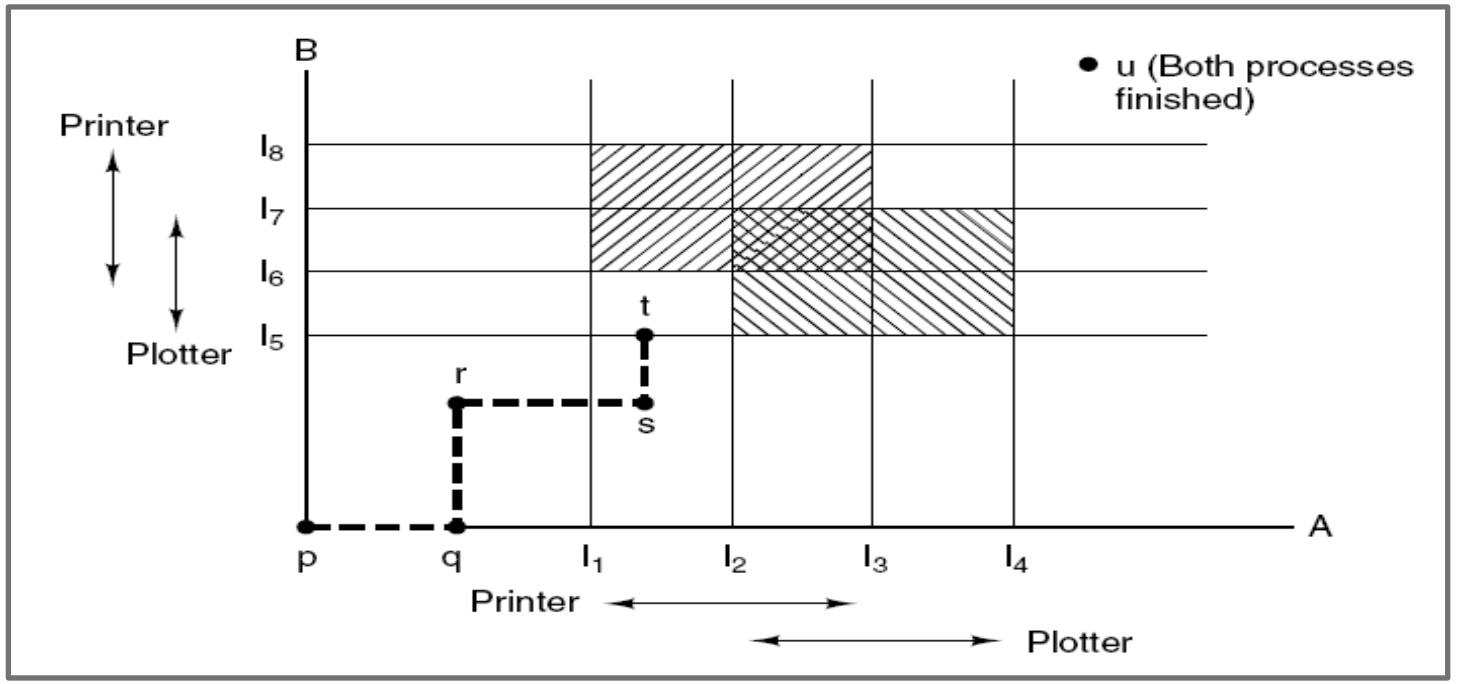
SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Deadlock

Deadlock: Evitare 3/7

Grafo di Allocazione Ogni punto del diagramma rappresenta uno stato congiunto dei due processi.

- p: nessun processo eseguito
- q: A eseguito, B comincia
- r: A ricomincia, in I_1 chiede il plotter
- …



modello per gestire due processi e due risorse, ad esempio una stampante e un plotter. L'asse orizzontale rappresenta il numero di istruzioni eseguite dal processo A. L'asse verticale rappresenta il numero di istruzioni eseguite dal processo B. In I_1 A richiede una stampante; a I_2 ha bisogno di un plotter. La stampante e il plotter vengono rilasciati rispettivamente in I_3 e I_4. Il processo B necessita del plotter da I_5 a I_7 e della stampante da I_6 a I_8 .

Operating Systems: Deadlock

Deadlock: Evitare 4/7

Algoritmi

1. Un algoritmo generico "per evitare i deadlock" esamina dinamicamente lo stato di allocazione delle risorse per accertarsi che la condizione di attesa circolare non possa mai verificarsi.
   - ➔ richiedere ad ogni processo di dichiarare il numero massimo di risorse che userà per ogni tipo
   - ➔ E iniziare ad assegnare le risorse solo se la richiesta complessiva non porterà allo stallo

2. **Algoritmo del banchiere** per singolo tipo di risorsa [Dijkstra, 1965]

3. **Algoritmo del banchiere** [Habermann, 1969] (per risorse multiple)

➔ **"stato sicuro"**

Stato Sicuro (**Safe State**): se da esso parte almeno un **cammino** che **non** porta ad un **deadlock**

➔ **Garantire che il sistema passi da uno stato sicuro ad un altro stato sicuro quando un processo chiede una nuova risorsa**

Operating Systems: Deadlock

Deadlock: Evitare 5/7

Stato Sicuro: Uno stato si dice sicuro se esiste un **ordine di pianificazione** in cui **ogni processo** può essere **eseguito** fino al completamento anche se tutti richiedessero immediatamente il numero massimo di risorse

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7
(e)

Lo stato è sicuro perché esiste una sequenza di allocazioni che consente il completamento di tutti i processi.

- B ➔ (b), (c)
- C ➔ (d), (e)
- A: può ottenre tutte le risorse di cui necessita

Operating Systems: Deadlock

Deadlock: Evitare 6/7

Algoritmo del Banchiere [Dijkstra, 1965]: estensione dell'algoritmo di deadlock detection.
Il banchiere (Sistema Operativo) sa che non tutti i clienti (processi) avranno bisogno del loro credito (risorse) massimo immediatamente, quindi riserva un numero di riorse inferiori a quelle effettivamente presenti sul sistema.

	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)

(a) Safe
(b) Safe
(c) se tutti i clienti chiedessero improvvisamente i loro prestiti massimi, il banchiere non potrebbe soddisfare nessuno di loro e ci troveremmo in una situazione di stallo.

➔ Unsafe State: deadlock non sicuro ma possible. Un proceso potrebbe avere bisogno dell'intero insiem di risorse (intera linea di credito disponibile)

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

Operating Systems: Deadlock

Deadlock: Evitare 7/7

Algoritmo del Banchiere [Habermann, 1969]:
Due matrici:

- **C** (Assigned): risorse correntemente assegnate
- **R** (Still Needed): risorse ancora necessarie per il completamento

Process	Tape drives	Plotters	Printers	CD ROMs
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	Printers	CD ROMs
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

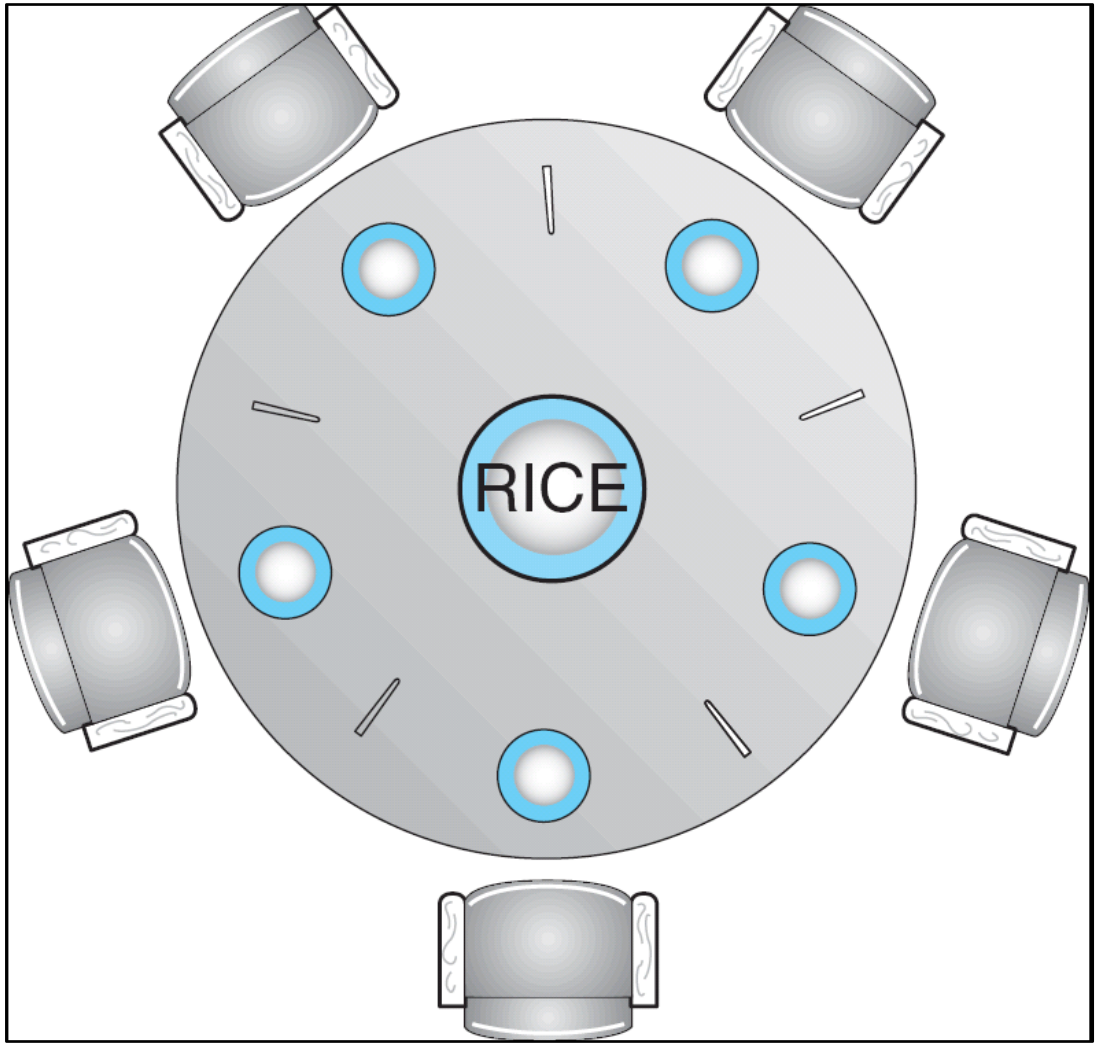
E = (6342)
P = (5322)
A = (1020)

Algoritmo di verifica della sicurezza dello stato:
1. Cerca una riga, R, le cui esigenze di risorse non soddisfatte sono tutte inferiori o uguali ad A. Se tale riga non esiste, il sistema alla fine andrà in stallo poiché nessun processo può essere completato (assumendo che i processi mantengano tutte le risorse fino alla loro uscita) .
2. Supponiamo che il processo della riga scelta richieda tutte le risorse di cui ha bisogno (il che è garantito possibile) e termini. Contrassegna quel processo come terminato e aggiungi tutte le sue risorse al vettore A.
3. Ripetere i passaggi 1 e 2 fino a quando tutti i processi sono contrassegnati come terminati (nel qual caso lo stato iniziale era sicuro) o non è rimasto alcun processo le cui esigenze di risorse possono essere soddisfatte (nel qual caso il sistema non era sicuro).

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti
Sapienza Università di Roma

Dining Philosopher Problem

Operating Systems: IPC
Dining Philosopher Problem 1/5



I 5 filosofi a cena:

- 5 filosofi su 5 sedie
- 5 bacchette per prendere il riso posto in una zuppiera condivisa
- Un filosofo può prendere una bacchetta alla volta, solo se è libera e se è posta tra lui e un vicino
- Quando il filosofo non mangia pensa e non interagisce con gli altri
- Quando il filosofo ha due bacchette mangia, dopo di che lascia le bacchette e torna a pensare

➔ Deve essere possibile che 2 filosofi possano mangiare contemporaneamente

➔ Processi in competizione per l'accesso esclusivo a risorse in numero limitato

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: IPC
Dining Philosopher Problem 2/5


```
#define N 5                 /*number of philosophers*/
Void philosopher(int i)     /*i:philosopher number, from 0 to 4*/
{
While(TRUE){
        think();        /*philosopher is thinking*/
        take_fork(i); /*take left fork*/
        take_fork(i+1)%N;    /*take right for;% is modulo operator*/
        eat():          /*self-explanatory*/
        put_fork(i);  /*put left fork back on table*/
        put_fork(i+1)%N;     /*put right fork back on table*/
        }
}
```

Non soluzione:

- **starvation**: se tutti i filosofi prendono la forchetta sinistra, nessuno riesce a prendere la destra, nessuno mangia
- **deadlock**: tutti i filosofi sono in attesa di una azione da parte degli altri

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: IPC
Dining Philosopher Problem 3/5


```
#define N 5                 /*number of philosophers*/
typedef int semaphore;
semaphore mutex = 1;
void philosopher(int i)      /*i:philosopher number, from 0 to 4*/
{
while(TRUE){
       think();      /*philosopher is thinking*/
       down (&mutex) ;      /* entering critical section */
       take_fork(i); /*take left fork*/
       take_fork(i+1)%N;    /*take right for;% is modulo operator*/
       eat():        /*self-explanatory*/
       put_fork(i);  /*put left fork back on table*/
       put_fork(i+1)%N;     /*put right fork back on table*/
       up (&mutex)    /* exiting critical section */
       }
}
```

Soluzione non efficiente: solo 1 filosofo per volta mangia, non 2!

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: IPC
Dining Philosopher Problem 4/5


```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N–1)%N          /* number of i's left neighbor */
#define RIGHT            (i+1)%N            /* number of i's right neighbor */
#define THINKING         0                  /* philosopher is thinking */
#define HUNGRY           1                  /* philosopher is trying to get forks */
#define EATING           2                  /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
        while (TRUE) {                      /* repeat forever */
                think( );                   /* philosopher is thinking */
                take_forks(i);              /* acquire two forks or block */
                eat( );                     /* yum-yum, spaghetti */
                put_forks(i);               /* put both forks back on table */
        }
}
```

Soluzione efficiente: sono definiti N+1 semafori.

- **S[N]**: per tracciare la richiesta di risorsa condivisa di oogni filosofo;

- **mutex**: usato per mutua esclusione. Garantisce che solo un processo alla volta possa leggere/scrivere sul buffer

Ed un array di stato:

- **state[N]**: vettore per tener traccia dello stato di ciascun filosofo : THINKING, HUNGRY, EATING

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: IPC
Dining Philosopher Problem 5/5


```
void take_forks(int i)          /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}
 void put_forks(i)                      /* i: philosopher number, from 0 to N–1 */
 {
     down(&mutex);                      /* enter critical region */
     state[i] = THINKING;               /* philosopher has finished eating */
     test(LEFT);                        /* see if left neighbor can now eat */
     test(RIGHT);                       /* see if right neighbor can now eat */
     up(&mutex);                        /* exit critical region */
 }

 void test(i) /* i: philosopher number, from 0 to N–1 */
 {
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
         state[i] = EATING;
         up(&s[i]);
     }
 }
```

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Sleeping Barber Problem

Operating Systems: IPC
Sleeping Barber 1/4

Barber Shop:

- 1 barbiere
- 1 sedia da barbiere
- n sedie nella sala d'attesa
- Se le ne sedie son vuote il barbiere si siede sulla sedia da barbiere e dorme
- Se arriva un cliente lo deve svegliare
- Ogni cliente addizionale che arriva, mentre il barbiere sta tagliando i capelli:
  - cerca una sedia in sala d'attesa
  - se sono tutte piene, esce dal negozio

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: IPC
Sleeping Barber 2/4


```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */
```

Possibile Soluzione: sono definiti 3 semafori.

- **customers**: clienti in attesa;

Il cui valore viene copiato nell'intero **waiting** (in modo da poterlo leggere agevolmente);

- **barber**: se il barbiere è in attesa di clienti;
- **mutex**:usata per mutua esclusione.

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: IPC
Sleeping Barber 3/4


```
void barber(void)
{
  while (TRUE) {
        down(customers);          /* go to sleep if # of customers is 0 */
        down(mutex);              /* acquire access to 'waiting' */
        waiting = waiting - 1;    /* decrement count of waiting customers */
        up(barbers);              /* one barber is now ready to cut hair */
        up(mutex);                /* release 'waiting' */
        cut_hair();               /* cut hair (outside critical region) */
  }
}
```

barber: procedura eseguita dal barbiere

1. quando il barbiere arriva la mattina, si blocca con un down() sul semaforo customers (poiché == 0)
2. Va a dormire.
3. Resta a dormire fino a che non viene svegliato da un customer

Operating Systems: IPC
Sleeping Barber 4/4


```
void customer(void)
{
        down (&mutex);          /* enter critical region */
        if(waiting < CHAIRS) { /* if no free chair, leave */
                waiting += 1;  /* waiting customers ++ */
                up(&customers);/* wake up barber if needed*/
                up(&mutex);    /* il barber riprnde l'haircut*/
                down(&barbers);/* a dormire se barber occupato*/
                get_haircut();
        } else {
                up(&mutex); // shop full, leave
        }
}
```

customer: procedura eseguita dal barbiere

1. Acquisisce mutex, entrando nella critical region. Un secondo client che dovesse entrare non potrebbe fare niente finché il primo non avesse rilasciato mutex;
2. Verifica il numero di waiting. Se maggiore del numero di sedie va via senza haircut
3. Altrimenti incrementa waiting
4. Si mette in fila (sveglia il barbiere, se necessario)
5. Resta a dormire fino a che non viene chiamato dal barber

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma