



UNIVERSITÀ
di **VERONA**

Dipartimento
di **INFORMATICA**

Deadlock

Anno Accademico 2020-2021

Graziano Pravadelli

Definizione

Sequenza di utilizzo dei processi che utilizzano risorse

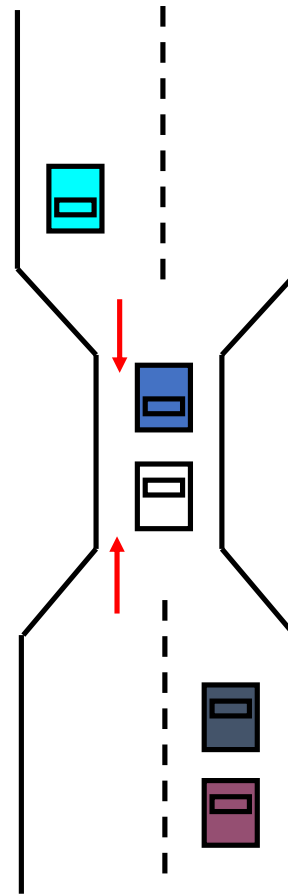
- Richiesta
 - Se non può essere immediatamente soddisfatta, il processo attende
- Utilizzo
- Rilascio

DEADLOCK

Un insieme di processi è in *deadlock* quando ogni processo è in attesa di un evento che può essere causato da un processo dello stesso insieme

Esempio di deadlock

Traffico solo in una direzione



Ponte è la risorsa condivisa

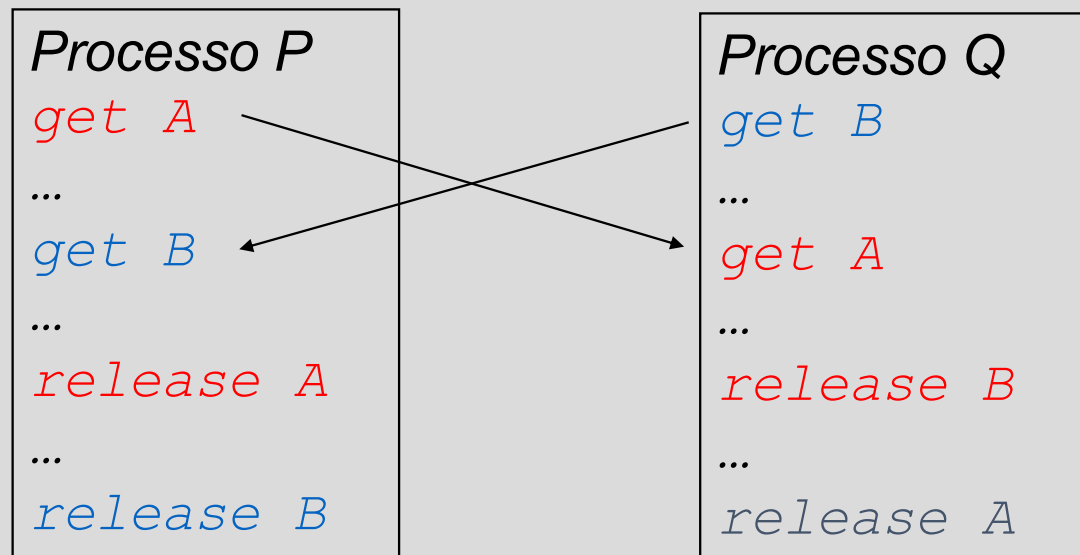
Se c'è deadlock, si può risolvere mandando indietro una macchina (preemption + rollback)

Possibile che più macchine debbano essere spostate

Starvation possibile

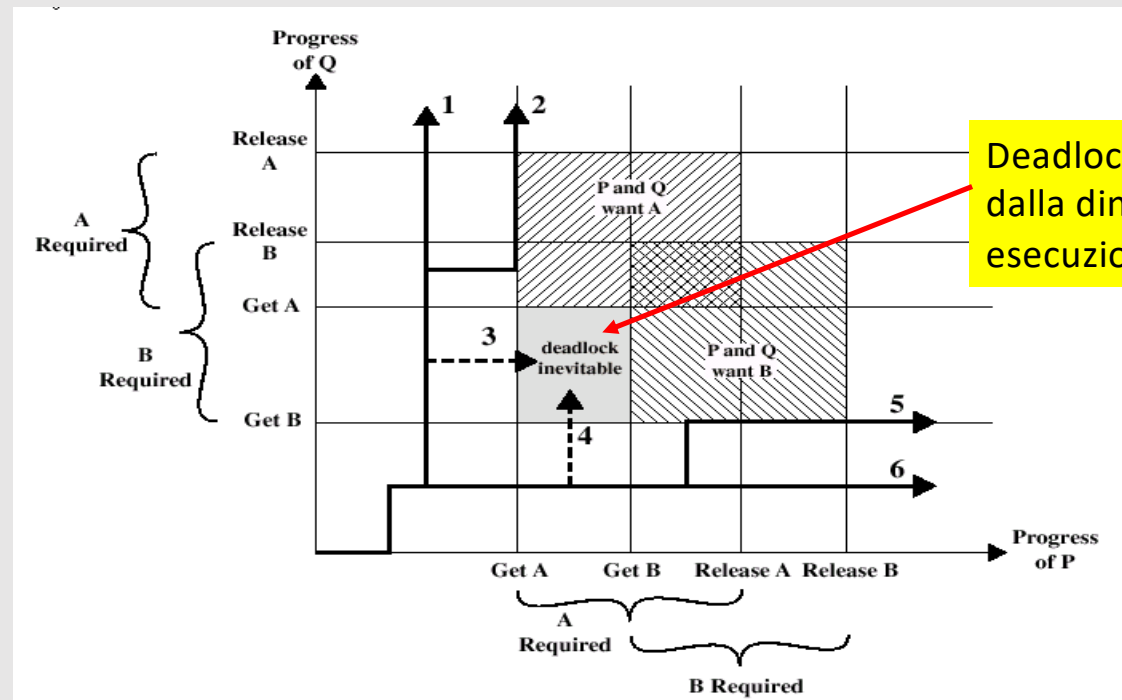
Esempio di deadlock

- P e Q devono utilizzare A e B in modo esclusivo



Esempio di deadlock – Traiettoria risorse

- 6 possibili sequenze di richiesta/rilascio



Deadlock dipende dalla dinamica della esecuzione

Condizioni necessarie

Mutua esclusione

- Almeno una risorsa deve essere non condivisibile

Hold and Wait

- Deve esistere un processo che detiene una risorsa e che attende di acquisirne un'altra, detenuta da un altro

No preemption

- Le risorse non possono essere rilasciate se non “volontariamente” dal processo che le usa

Attesa circolare

- Deve esistere un insieme di processi che attendono ciclicamente il liberarsi di una risorsa

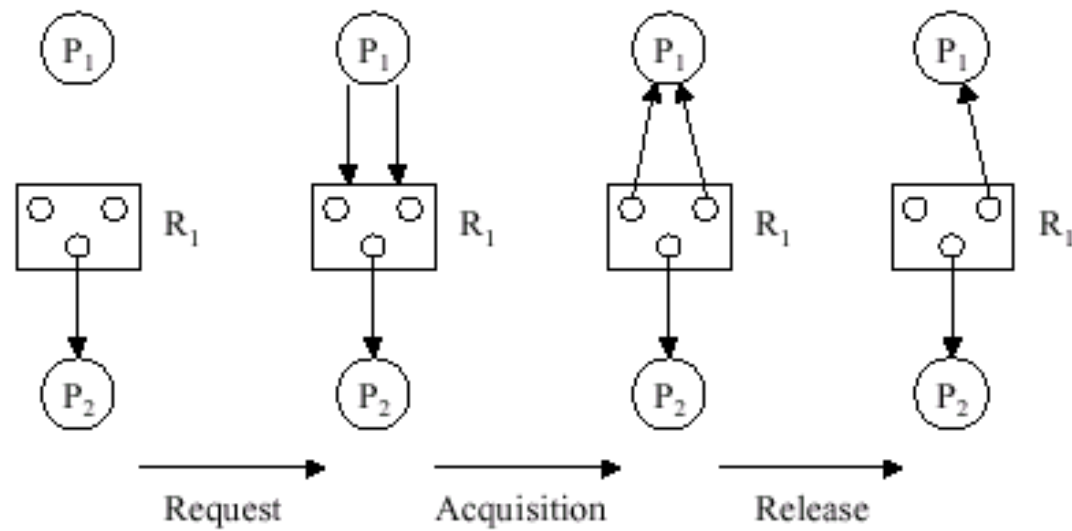
Devono essere vere contemporaneamente

- Se una non si verifica, non si ha deadlock

Modello astratto (RAG)

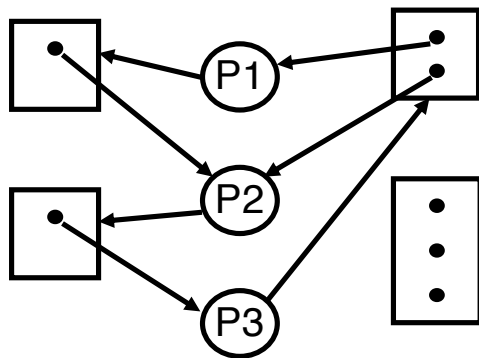
- RAG = *Resource Allocation Graph*: $G(V,E)$
 - V = nodi
 - Cerchi = processi (CPU, I/O, memoria)
 - Rettangoli = risorse
 - Nei rettangoli vi sono tanti “ • ” quante sono le istanze della corrispondente risorsa
 - E = archi
 - Da processi a risorse: processo richiede risorsa
 - Da risorse a processi: processo detiene risorsa

RAG - esempio



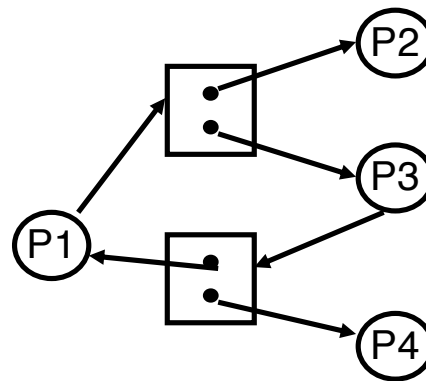
- $V = \{\{P_1, P_2\}, \{R_1\}\}$
- $E_{\text{iniziale}} = \{(R_1, P_2)\}$ $E_{\text{finale}} = \{(R_1, P_1), (R_1, P_2)\}$

RAG e deadlock



DEADLOCK

Se il RAG non
contiene cicli, non
ci sono deadlock



NO DEADLOCK

Se contiene cicli:

se si ha una sola
istanza per
risorsa allora si
deadlock

se ci sono più
istanze, dipende
dallo schema di
allocazione

Gestione dei deadlock - alternative

Prevenzione statica

- Evitare che si possa verificare una delle quattro condizioni

Prevenzione dinamica (*avoidance*) basata su allocazione delle risorse

- Mai usata poiché richiede conoscenza troppo approfondita delle richieste di risorse

Rivelazione (*detection*) e ripristino (*recovery*)

- Permettere che si verifichino deadlock
- Prevedere metodi per riportare il sistema al funzionamento normale

Algoritmo dello struzzo

- Non fare nulla, i deadlock sono rari e gestirli costa troppo

Prevenzione statica

- Obiettivo:
 - Impedire che si verifichi una delle 4 condizioni che devono essere vere contemporaneamente perché si verifichi un deadlock

Prevenzione statica

- Mutua esclusione
 - E' irrinunciabile per certi tipi di risorsa
 - Non possiamo toglierla

Prevenzione statica

- Hold and wait
 - Soluzioni
 - Un processo alloca all'inizio tutte le risorse che deve utilizzare
 - Un processo può ottenere una risorsa solo se non ne ha altre
 - Problemi
 - Basso utilizzo delle risorse
 - Possibilità di *starvation* (richiesta di molte risorse molto "popolari")
 - Conoscenza del numero di risorse richieste?

Prevenzione statica

- No preemption
 - Soluzioni
 - Un processo che richiede una risorsa non disponibile deve cedere tutte le altre risorse che detiene
 - In alternativa, può cedere risorse che detiene su richiesta di un altro processo
 - Problemi
 - Fattibile solo per risorse il cui stato può essere facilmente “ristabilito” (CPU, registri, semafori, file)
 - Non per stampanti, nastri, ...

Prevenzione statica

- Attesa circolare
 - Soluzione
 - Assegnare una priorità (ordinamento globale) ad ogni risorsa
 - $F: R \rightarrow N$
 - $F(R_0) < F(R_1) < \dots < F(R_n)$
 - Un processo può richiedere risorse solo in ordine crescente di priorità
 - Quindi l'attesa circolare diventa impossibile poiché:
 - Se $P_0 \rightarrow R_0 \rightarrow P_1 \rightarrow \dots \rightarrow R_{n-1} \rightarrow P_n \rightarrow R_n \rightarrow P_0$
 - Allora $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$
Impossibile!
 - Priorità deve seguire il normale ordine di richiesta (Es.: disco prima di stampante)

Prevenzione dinamica

- Le tecniche di prevenzione statica possono portare a un basso utilizzo delle risorse perché mettono vincoli sul modo in cui i processi possono accedere alle risorse

Prevenzione dinamica

Obiettivo:

- Prevenzione in base alle richieste
 - Analisi dinamica del grafo delle risorse per evitare situazioni cicliche

Requisito:

- Conoscenza del caso peggiore (bisogna conoscere il massimo numero di istanze di una risorsa richieste per processo)


Prevenzione dinamica – stato safe

Stato di una risorsa calcolato
come:

- numero di istanze allocate
- numero di istanze disponibili

Il sistema si trova in uno stato
sicuro (*safe*)

- se esiste una *sequenza safe*, ovvero
- se usando le risorse disponibili, può allocare risorse ad ogni processo, in qualche ordine, in modo che ciascun di essi possa terminare la sua esecuzione



Sequenza
safe

Una sequenza di processi (P_1, \dots, P_N) è *safe* se, per ogni P_i , le risorse che P_i può richiedere possono essere esaudite usando:

le risorse disponibili

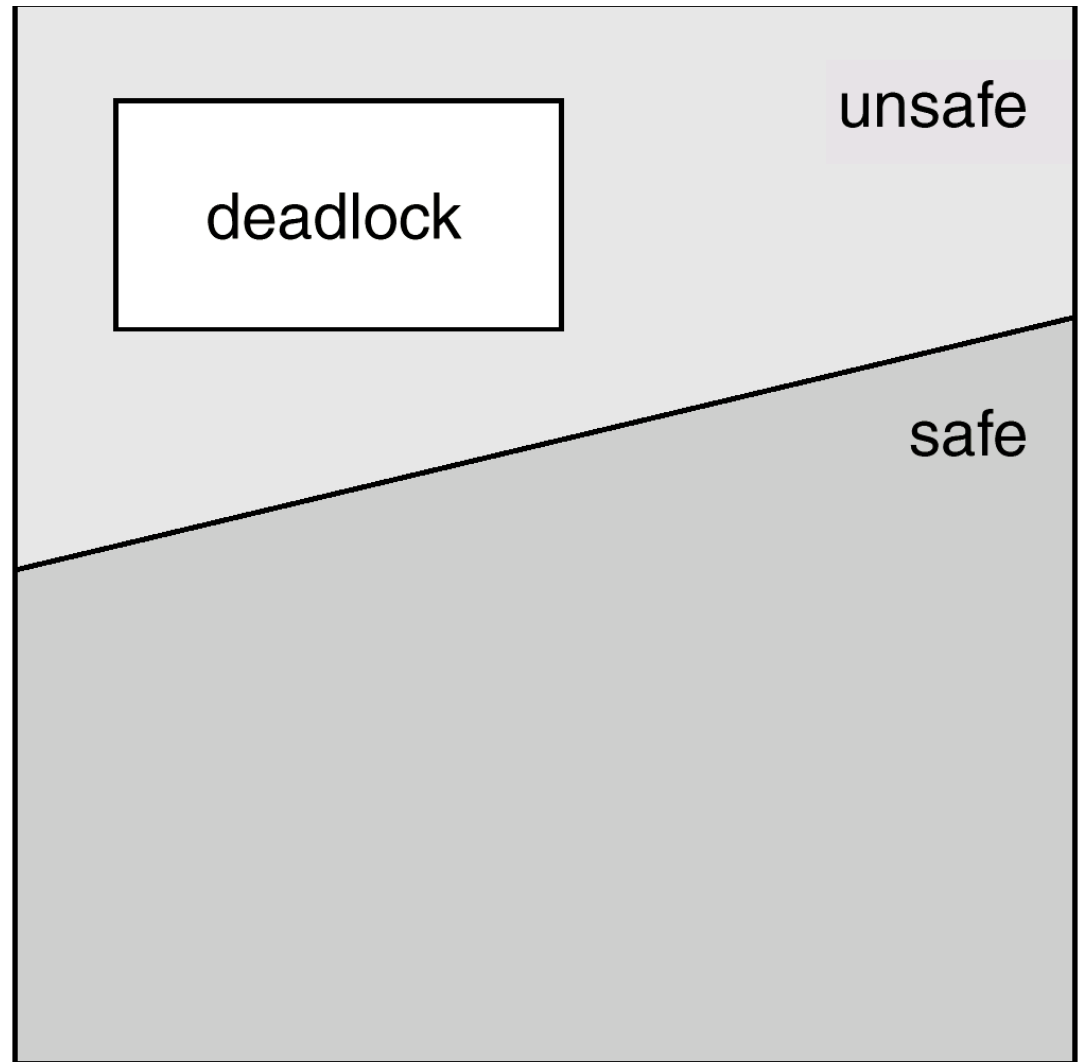
quelle detenute da $P_j, j < i$
(attendendo che P_j termini)



Se non esiste tale sequenza, siamo in uno stato *unsafe*

Non tutti gli stati *unsafe* sono stati di deadlock, ma da stato *unsafe* posso andare in deadlock

Spazio degli stati



Stato safe e unsafe – Esempio

- Supponiamo di avere:
 - 3 processi: P0, P1, P2
 - 12 istanze di 1 risorsa
 - 3 istanze libere
- Siamo in uno stato safe?
 - Si, esiste sequenza safe: < P1, P0, P2 >
 - P1 prende e rilascia, poi P0 infine P2
- Se ora P2 richiede 1 risorsa e gli viene assegnata, siamo ancora in uno stato safe?
 - No, si entra in uno stato unsafe!
 - Rimangono solo 2 risorse
 - P1 può eseguire, ma quando termina sono disponibili solo 4 istanze
 - P0 ne chiede 5, P2 ne chiede 6!
 - Deadlock P2 \leftrightarrow P0

	Richieste	Possedute
P ₀	10	5
P ₁	4	2
P ₂	9	2

Prevenzione dinamica

Idea:

- Utilizzare algoritmi che lasciano il sistema sempre in uno stato safe
 - All'inizio il sistema è in uno stato safe
 - Ogni volta che P richiede R, R viene assegnata a P se si rimane in uno stato safe

Svantaggio:

- L'utilizzo delle risorse è minore rispetto al caso in cui non uso tecniche di prevenzione dinamica

Prevenzione dinamica

- Due alternative:
 - Algoritmo con RAG
 - Funziona solo se c'è una sola istanza per risorsa
 - Algoritmo del banchiere
 - Funziona qualunque sia il numero di istanze

Algoritmo con RAG

Funziona solo se ho un'istanza per ogni risorsa

Il RAG viene aumentato con archi di reclamo:

- $P_i \rightarrow R_j$ se P_i può richiedere R_j in futuro
- Indicati con freccia tratteggiata

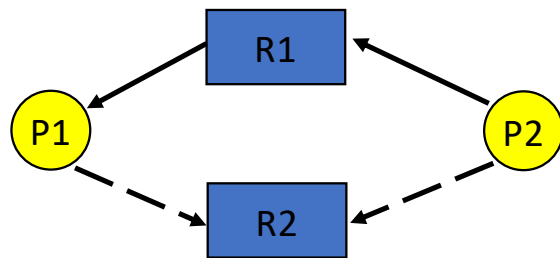
All'inizio, ogni processo deve dire quali risorse vorrebbe usare durante la sua esecuzione

Una richiesta viene soddisfatta sse l'allocazione della risorsa non crea un ciclo nel RAG

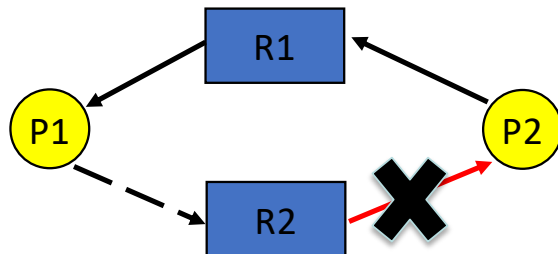
Serve un algoritmo per la rilevazione cicli

- Complessità $O(n^2)$ (dove n = n° di processi)

Algoritmo con RAG – esempio



- Siamo in uno stato sicuro



- Se, ad un certo punto, P2 richiede R2, la richiesta non viene accettata perché lo stato diventerebbe unsafe

Algoritmo del banchiere

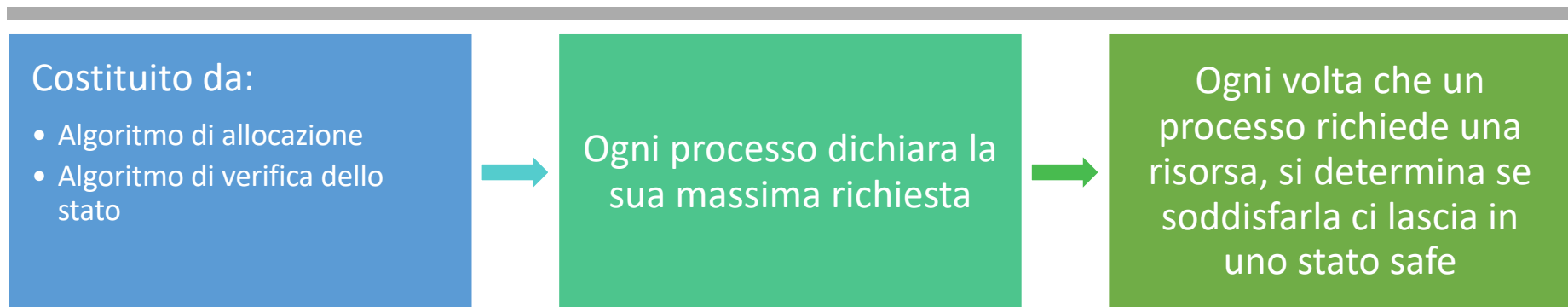
Meno efficiente dell'algoritmo con RAG...

... ma funziona con più istanze delle risorse

Idea:

- Il banchiere non deve mai distribuire tutto il denaro che ha in cassa perché altrimenti non potrebbe più soddisfare successivi clienti

Algoritmo del banchiere



```
int available[m]; /* n° di istanze di Ri disponibili */
int max[n][m]; /* matrice delle richieste di risorse */
int alloc[n][m]; /* matrice allocazione corrente */
int need[n][m]; /* matrice bisogno rimanente ovvero
                 need[i][j] = max[i][j] - alloc[i][j] */
```

Algoritmo di allocazione (P_i)

```
void request(int req_vec[]) {  
    if (req_vec[] > need[i][])  
        error(); /* superato il massimo preventivato */  
    if (req_vec[] > available[])  
        wait(); /* attendo che si liberino risorse */  
    available[] = available[] - req_vec[];  
    alloc[i][] = alloc[i][] + req_vec[];  
    need[i][] = need[i][] - req_vec[];  
    if (!state_safe()) { /* se non è safe, ripristino il vecchio stato */  
        available[] = available[] + req_vec[];  
        alloc[i][] = alloc[i][] - req_vec[];  
        need[i][] = need[i][] + req_vec[];  
        wait();  
    }  
}
```

← Richieste
del processo P_i

← “simulo”
l’assegnazione

← rollback

$O(m \cdot n^2)$

Algoritmo di verifica dello stato

```
boolean state_safe() {  
    int work[m] = available[];  
    boolean finish[n] = (FALSE, ..., FALSE);  
    int i;  
    while (finish != (TRUE, ..., TRUE)) {  
        /* cerca Pi che NON abbia terminato e che possa  
           completare con le risorse disponibili in work */  
        for (i=0; (i<n) && (finish[i] || (need[i][] > work[])); i++);  
        if (i==n)  
            return FALSE; /* non c'è → unsafe */  
        else {  
            work[] = work[] + alloc[i][];  
            finish[i] = TRUE;  
        }  
    }  
    return TRUE;  
}
```

Ho già sottratto le richieste di P_i!

Sono arrivato in fondo, senza trovare finish[i]=FALSE e need[i][] <= work[]

L'ordine non ha importanza. Se + processi possono eseguire, ne posso scegliere uno a caso, gli altri eseguiranno dopo, visto che le risorse possono solo aumentare

Algoritmo del banchiere - esempio

- 5 processi: P_0, P_1, P_2, P_3, P_4
- 3 risorse:
 - A (10 istanze) B (5 istanze) C (7 istanze)
- Fotografia al tempo T_0 :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Algoritmo del banchiere - esempio

- Al tempo T_0 il sistema è in stato safe:
 - $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ è una sequenza safe
- Al tempo T_1 , P_1 richiede (1,0,2)
 - Request_1 \leq Available? $(1,0,2) \leq (3,3,2) \Rightarrow$ OK
 - Nuova situazione:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	7 5 3	2 3 0	7 4 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Algoritmo del banchiere - esempio

- Siamo ancora in uno stato safe? Verifichiamolo:
 - $work = (2,3,0)$
 - $i=0$ $finish=FALSE$, $need[0] = (7,4,3) > work$
 - $i=1$ $finish=FALSE$, $need[1] = (0,2,0) < work$
 - $work = work + (3,0,2) = (5,3,2)$
 - $finish[1] = TRUE$
 - $i=2$ $finish=FALSE$, $need[2] = (6,0,0) > work$
 - $i=3$ $finish=FALSE$, $need[3] = (0,1,1) < work$
 - $work = work + (2,1,1) = (7,4,3)$
 - $finish[3] = TRUE$
 - ...
 - Sequenza safe finale: $\langle P1, P3, P4, P0, P2 \rangle$

Algoritmo del banchiere - esempio

- Al tempo T_2 , P_0 richiede (0,2,0)
 - $\text{Request}_0 \leq \text{Available}$? $(0,2,0) \leq (2,3,0) \Rightarrow \text{OK}$
 - Nuova situazione

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 3 0	7 5 3	2 1 0	7 2 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Algoritmo del banchiere - esempio

- Siamo ancora in uno stato safe? Verifichiamolo:
 - $work = (2,1,0)$
 - $i=0$ $finish=FALSE$, $need[0]=(7,2,3) > work$
 - $i=1$ $finish=FALSE$, $need[1]=(0,2,0) > work$
 - $i=2$ $finish=FALSE$, $need[2]=(6,0,0) > work$
 - $i=3$ $finish=FALSE$, $need[3]=(0,3,1) > work$
 - $i=4$ $finish=FALSE$, $need[4]=(4,3,1) > work$
- Stato unsafe!

Rilevazione deadlock & ripristino

Prevenzione statica e dinamica sono conservativi e riducono eccessivamente l'utilizzo delle risorse

Due approcci
alternativi:

- Rilevazione e ripristino tramite RAG
- Algoritmo di rilevazione

Rilevazione e ripristino con RAG

Funziona solo con una risorsa per tipo

Analizzare periodicamente il RAG, verificare se esistono deadlock (*detection*) ed iniziare il ripristino (*recovery*)

- Vantaggi
 - Conoscenza anticipata delle richieste non necessaria
 - Utilizzo migliore
- Svantaggio
 - Costo del recovery

Algoritmo di rilevazione

Basato sull'esplorazione di ogni possibile sequenza di allocazione per i processi che non hanno ancora terminato

Se la sequenza va a buon fine (*safe*), non c'è deadlock

Strutture dati simili ad algoritmo del banchiere:

- `int available[m];` /* n° di istanze di R_i disponibili */
- `int alloc[n][m];` /* matrice allocazione corrente */
- `int req_vec[n][m];` /* matrice delle richieste */

$O(m \cdot n^2)$

Algoritmo di detection

```
int work[m] = available[m];
bool finish[] = (FALSE,...,FALSE), found = TRUE;
while (found) {
    found = FALSE;
    for (i=0; i<n && !found; i++) {
        /* cerca un  $P_i$  con richiesta soddisfacibile */
        if (!finish[i] && req_vec[i][] <= work[]) {
            /* assume ottimisticamente che  $P_i$  esegua fino al termine
            e che quindi restituisca le risorse */
            work[] = work[] + alloc[i][];
            finish[i]=TRUE;
            found=TRUE;
        }
    }
} /* se finish[i]=FALSE per un qualsiasi i,  $P_i$  è in deadlock */
```

Se non è così
il possibile deadlock
verrà evidenziato
alla prossima
esecuzione dell'algoritmo

Rilevazione - esempio

- 5 processi: P_0, P_1, P_2, P_3, P_4
- 3 tipi di risorsa:
 - A (7 istanze), B (2 istanze), C (6 istanze)
- Fotografia al tempo T_0 :

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Rilevazione - esempio

- Siamo in una situazione di deadlock? Verifichiamolo:
 - $work = (0,0,0)$
 - $i=0$ $req[0]=(0,0,0) \leq work$ OK
 $work = work + (0,1,0) = (0,1,0)$ $finish[0] = true$ P0 ✓
 - $i=1$ $req[1]=(2,0,2) \leq work$ NO
 - $i=2$ $req[2]=(0,0,0) \leq work$ OK
 $work = work + (3,0,3) = (3,1,3)$ $finish[2] = true$ P2 ✓
 - $i=3$ $req[3]=(1,0,0) \leq work$ OK
 $work = work + (2,1,1) = (5,2,4)$ $finish[3] = true$ P3 ✓
 - ...
- La sequenza $\langle P0, P2, P3, P1, P4 \rangle$ dà $finish[i] = true$ per ogni i

Rilevazione - esempio

- Supponiamo invece che P_2 richieda un'ulteriore istanza della risorsa C

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Esempio

- Siamo in una situazione di deadlock? Verifichiamolo
 - $work = (0,0,0)$
 - $i=0$ $req[0]=(0,0,0) \leq work$ OK
 $work = work + (0,1,0) = (0,1,0)$ $finish[0] = true$ P0✓
 - $i=1$ $req[1]=(2,0,2) \leq work$ NO
 - $i=2$ $req[2]=(0,0,1) \leq work$ NO
 - $i=3$ $req[3]=(1,0,0) \leq work$ NO
 - $i=4$ $req[4]=(0,0,2) \leq work$ NO
- DEADLOCK formato da P1, P2, P3, P4

Ripristino

Quanto spesso chiamare l'algoritmo di rilevazione?

- Dopo ogni richiesta
- Ogni N secondi
- Quando utilizzo della CPU scende sotto una soglia T

Cosa fare?

- Uccisione processi coinvolti
- Prelazione delle risorse dai processi bloccati nel deadlock

Uccisione processi coinvolti

Uccisione di tutti i processi

- Costoso
 - Tutti i processi devono ripartire e perdono il lavoro svolto

Uccisione selettiva fino alla scomparsa del deadlock

- Costoso
 - Invoca l'algoritmo di rilevazione dopo ogni uccisione
- In che ordine?
 - Priorità, tipi di risorse allocate, quante risorse mancavano, quanto tempo mancava alla fine, interattivo o batch, è il processo del capo?

Prelazione delle risorse

A chi toglierle e in quale ordine?

- Problema
 - Il processo che subisce la prelazione non può continuare normalmente
- Soluzione
 - Rollback in uno stato *safe*, e ripartenza da questo stato

Eventualmente *total rollback* (cioè riparto da zero)

Problema

- Starvation possibile, se tolgo le risorse sempre agli stessi processi

Soluzione

- Considerare il numero di rollback nei fattori di costo

Conclusione

Ognuno dei tre approcci visti
ha vantaggi e svantaggi

Nessuno è sempre superiore
agli altri

Soluzione “combinata”:

- Partizionare le risorse in classi
- Usare una strategia di ordinamento tra classi di risorse (priorità)
- All'interno di una classe, usare l'algoritmo più appropriato per quella classe

Conclusione

- Partizionare e ordinare le risorse in classi:
 1. Risorse interne (usate dal sistema, es.: PCB, I/O)
 2. Memoria
 3. Risorse di processo (es. File)
 4. Spazio di swap (blocchi su disco)

Conclusione

- Algoritmi specifici:
 1. Prevenzione tramite ordinamento delle risorse
 2. Prevenzione tramite prelazione
 - Un job può in genere essere “swappato”
 3. Prevenzione dinamica
 - Richiesta massima di risorse tipicamente nota a priori
 4. Prevenzione tramite preallocazione (v. hold & wait)
 - Richiesta massima di memoria tipicamente nota a priori

Conclusione

- Possibile anche la soluzione più semplice
 - STRUZZO
 - Es.: UNIX
 - In fin dei conti:
 - I deadlock si verificano poche volte!
 - La prevenzione è costosa!
 - Il recovery è costoso!
 - Gli algoritmi spesso sono sbagliati!

A. Tannenbaum