

## Gestione delle Risorse

1

## Risorse

- Un sistema di elaborazione e' composto da un insieme di risorse da assegnare ai processi presenti
- I processi competono nell'accesso alle risorse
- Esempi di risorse
  - memoria
  - stampanti
  - processore
  - dischi
  - interfaccia di rete
  - descrittori di processo

2

## Classi di Risorse

- Le risorse possono essere suddivise in *classi*  
  
Esempi: byte della memoria, stampanti dello stesso tipo, etc.
- Le risorse di una classe vengono dette *istanze* della classe
- Il numero di risorse in una classe viene detto *molteplicita'* del tipo di risorsa
- Un processo non puo' richiedere una specifica risorsa, ma solo una risorsa di una specifica classe
- Una richiesta per una classe di risorse puo' essere soddisfatta da qualsiasi istanza di quel tipo

3

## Assegnazione delle Risorse

- Risorse ad *assegnazione statica*
  - Avviene al momento della creazione del processo e rimane valida fino alla terminazione
  - Esempi: descrittori di processi, aree di memoria (in alcuni casi)
- Risorse ad *assegnazione dinamica*
  - i processi richiedono le risorse durante la loro esistenza
  - le utilizzano una volta ottenute
  - le rilasciano quando non piu' necessarie (eventualmente alla terminazione del processo)
- Esempi: periferiche di I/O, aree di memoria (in alcuni casi)

4

## Tipi di richieste

- Richiesta *singola*:
  - si riferisce a una singola risorsa di una classe definita
  - e' il caso normale
- Richiesta *multipla*:
  - si riferisce a una o piu' classi, e per ogni classe, ad una o piu' risorse e deve essere soddisfatta integralmente

5

- Richiesta *bloccante*:
  - il processo richiedente si sospende se non ottiene immediatamente l'assegnazione
  - la richiesta rimane pendente e viene riconsiderata dalla funzione di gestione ad ogni rilascio
- Richiesta *non bloccante*
  - la mancata assegnazione viene notificata al processo richiedente, senza provocare la sospensione

## Tipi di Risorse

- Risorse *seriali* (o con accesso mutuamente esclusivo):
  - una singola risorsa non puo' essere assegnata a piu' processi contemporaneamente
  - Esempi: i processori, le sezioni critiche, le stampanti
- Risorse *non seriali*
  - Esempio: file di sola lettura

6

## Risorse prerilasciabili (preemptable)

- Una risorsa si dice *prerilasciabile* se la funzione di gestione puo' sottrarla ad un processo prima che questo l'abbia effettivamente rilasciata
- Meccanismo di gestione:
  - il processo che subisce il prerilascio deve sospendersi
  - la risorsa prerilasciata sara' successivamente restituita al processo
- Una risorsa e' prerilasciabile:
  - se il suo stato non si modifica durante l'utilizzo
  - oppure il suo stato puo' essere facilmente salvato e ripristinato
- Esempi: processore, blocchi o partizioni di memoria (nel caso di assegnazione dinamica)

7

## Risorse non prerilasciabili

- Una risorsa *e' non prerilasciabile* se la funzione di gestione non puo' sottrarla al processo al quale e' assegnata
- Sono non prerilasciabili le risorse il cui stato non puo' essere salvato e ripristinato
- Esempi: stampanti, classi di sezioni critiche, partizioni di memoria (nel caso di gestione statica)

8

## Il problema dello Stallo (Deadlock)

- Definizione di deadlock:

Un insieme di processi si trova in deadlock (stallo) se ogni processo dell'insieme è in attesa di un evento che solo un altro processo dell'insieme può provocare.

- Tipicamente, l'evento atteso è proprio il rilascio di risorse non prerilasciabili.
- Il numero dei processi e il genere delle risorse e delle richieste non è influente.

9

## Come affrontare il Deadlock

- Le situazioni di deadlock (stallo) impediscono ai processi di terminare correttamente
- le risorse bloccate in deadlock non possono essere utilizzate da altri processi

10

## Un Possibile Protocollo per Uso di Risorse

- Passi per la richiesta e l'uso di una risorsa:

1. Richiedere la risorsa
2. Usare la risorsa
3. Rilasciare la risorsa

- Se al momento della richiesta la risorsa non è disponibile, ci sono diverse alternative (attesa, attesa limitata, fallimento, . . .)

11

## Allocazione delle risorse

Possiamo usare i semafori per sincronizzare i processi che accedono alle risorse

Esempio:

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

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

(a)

```
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)

12

## Allocazione di più risorse

Se ogni processo usa varie risorse potremmo usare più mutex, uno per ogni risorsa. Ma come usare correttamente i mutex?

Esempio:

```
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)

13

## Allocazione di più risorse (cont.)

- La soluzione (a) è sicura: non può portare a deadlock
- La soluzione (b) non è sicura: può portare a deadlock
- La soluzione (b) richiede l'accesso al codice di tutti i programmi che accedono alle risorse

14

## Ulteriori Problemi

- I programmi che accedono alle risorse potrebbero appartenere a diversi utenti
- Con decine, centinaia di risorse (come quelle che deve gestire il kernel stesso), determinare se una sequenza di allocazioni è sicura non è semplice
- Sono necessari dei metodi per
  - riconoscere la possibilità di deadlock (prevenzione)
  - riconoscere un deadlock
  - la risoluzione di un deadlock

15

## Condizioni necessarie per il deadlock

Un deadlock puo' verificarsi *solo* se le seguenti quattro condizioni sono vere

### 1. Mutua esclusione:

Le risorse coinvolte devono essere seriali

### 2. Assenza di prerilascio:

Le risorse non sono prerilasciabili

### 3. Hold&Wait:

le richieste devono essere bloccanti e un processo che ha richiesto ed ottenuto delle risorse puo' chiederne altre

### 4. Attesa circolare

Esiste un sottoinsieme di processi  $\{P_0, P_1, \dots, P_n\}$  tali che  $P_i$  è in attesa di una risorsa che è assegnata a  $P_{i+1 \text{ mod } n}$

Le condizioni 1-4 sono necessarie: se anche solo una di queste condizioni manca, il deadlock NON puo' verificarsi

16

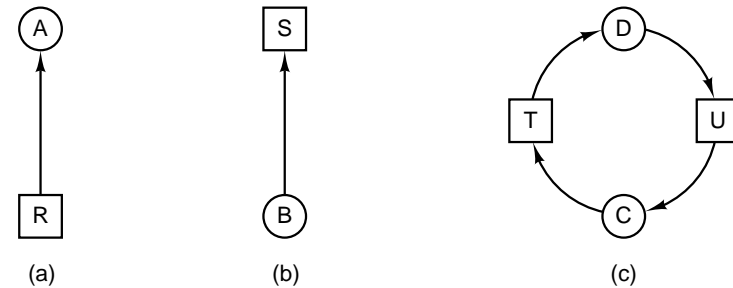
## Grafo di allocazione risorse

Le quattro condizioni si modellano con un grafo orientato, detto *grafo di allocazione delle risorse*: Un insieme di vertici  $V$  e un insieme di archi  $E$

- $V$  è partizionato in due tipi:
  - $P = \{P_1, P_2, \dots, P_n\}$ , l'insieme di tutti i processi del sistema.
  - $R = \{R_1, R_2, \dots, R_m\}$ , l'insieme di tutte le risorse del sistema.
- *archi di richiesta*: archi orientati  $P_i \rightarrow R_j$
- *archi di assegnamento (acquisizione)*: archi orientati  $R_j \rightarrow P_i$

Uno *stallo* è un ciclo nel grafo di allocazione risorse.

17

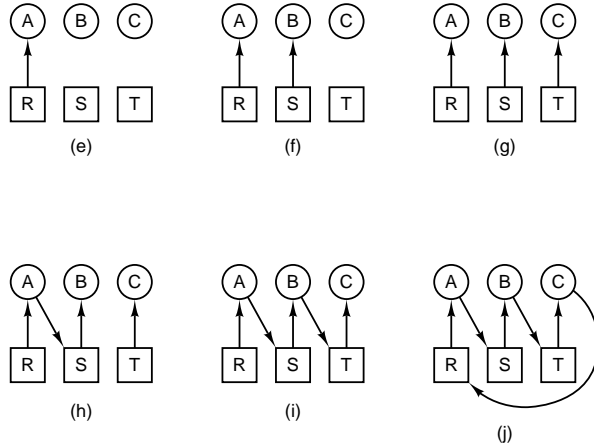


- (a) acquisizione di una risorsa R da parte del processo  
(b) richiesta di S da parte di B  
(c) situazione di stallo

## Grafo di allocazione risorse (cont.)

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

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



18

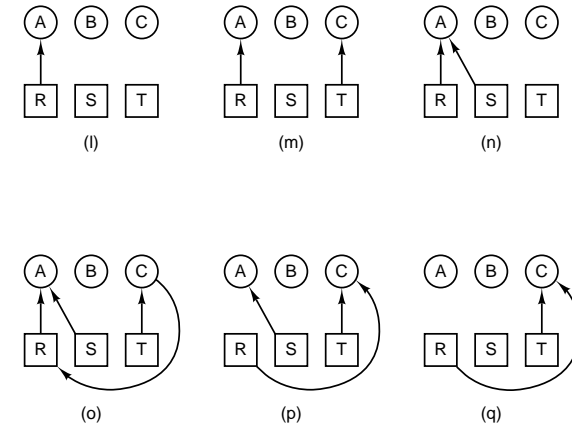
## Principali fatti

- Se il grafo non contiene cicli  $\Rightarrow$  nessun deadlock.
- Se il grafo contiene un ciclo  $\Rightarrow$ 
  - se c'è solo una istanza per tipo di risorsa, allora deadlock
  - se ci sono più istanze per tipo di risorsa, allora c'è la possibilità di deadlock

20

## Grafo di allocazione risorse (cont.)

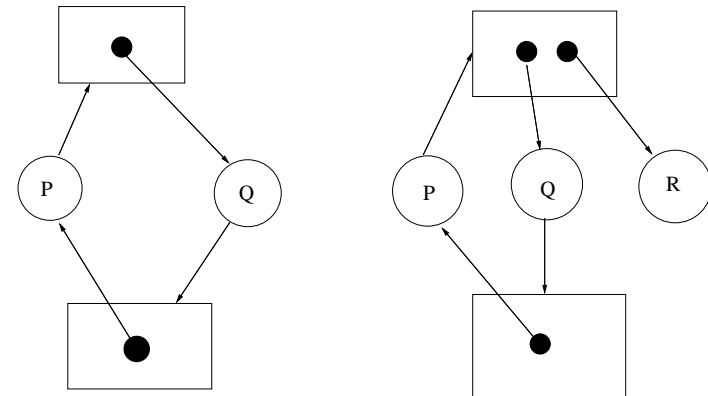
1. A requests R
  2. C requests T
  3. A requests S
  4. C requests R
  5. A releases R
  6. A releases S
- no deadlock



Nota: manca l'arco  $T \rightarrow C$  in (n) - fig. dal libro di Tanenbaum -

19

## Controesempio condizione sufficiente



**DEADLOCK**  
P e Q sono bloccati  
(non possono utilizzare/rilasciare risorse)

**NO DEADLOCK**  
Quando R rilascia la risorsa  
P si sblocca....

21

## Uso dei grafi di allocazione risorse

I grafi di allocazione risorse sono uno strumento per verificare se una sequenza di allocazione porta ad un deadlock.

- Il sistema operativo ha a disposizione molte sequenze di scheduling dei processi
- per ogni sequenza, può “simulare” la successione di allocazione sul grafo
- e scegliere una successione che non porta al deadlock.

Il FCFS è una politica “safe”, ma insoddisfacente per altri motivi.

Il round-robin in generale non è safe.

22

## Gestione dei Deadlock

- Ignorare il problema, fingendo che non esista (Molto usato).
- Permettere che il sistema entri in un deadlock, riconoscerlo e quindi risolverlo.
- Cercare di evitare dinamicamente le situazioni di stallo, con una accorta gestione delle risorse.
- Assicurare che il sistema non possa mai entrare **mai** in uno stato di deadlock, negando una delle quattro condizioni necessarie.

23

## I Approccio: Ignorare il problema

- Assicurare l'assenza di deadlock impone costi (in prestazioni, funzionalità) molto alti.
- Costi necessari per alcuni, ma insopportabili per altri.
- Si considera il rapporto costo/benefici: se la probabilità che accada un deadlock è sufficientemente bassa, non giustifica il costo per evitarlo
- Esempi: il `fork` di Unix, la rete Ethernet, ...
- Approccio adottato dalla maggior parte dei sistemi (Unix e Windows compresi): ignorare il problema.
  - L'utente preferisce qualche stallo occasionale (da risolvere “a mano”), piuttosto che eccessive restrizioni.

24

## II Approccio: Identificazione e Risoluzione del Deadlock

- Lasciare che il sistema entri in un deadlock
- Riconoscere l'esistenza del deadlock con opportuni algoritmi di identificazione
- Avere una politica di risoluzione (recovery) del deadlock

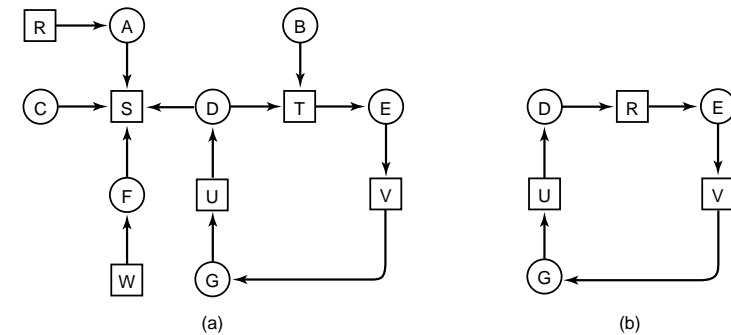
25

## Algoritmo di identificazione: una risorsa per classe

- Esiste una sola istanza per ogni classe
- Si mantiene un grafo di allocazione delle risorse aggiornato, registrando tutte le assegnazioni e le richieste di risorse
- Si usa un algoritmo di ricerca di cicli per grafi orientati
- Ad esempio: visita depth-first del grafo con complessita' nel caso peggiore  $O(n^2)$  dove  $n$ =numero nodi

26

Esempio:



(a) Situazione di stallo

(b) Ciclo contenuto in (a)

## Più risorse per classe

- Abbiamo visto che se una classe puo' avere piu' istanze l'esistenza di un ciclo nel grafo di allocazione non garantisce l'esistenza di un deadlock
- Date  $m$  classi di risorse e  $n$  processi, si puo' utilizzare un algoritmo che lavora su:
  - *Esistenti*: il vettore che identifica le **risorse esistenti** per ogni classe
  - *Disponibili*: il vettore che identifica le **risorse disponibili** per ogni classe
  - *Assegnate*: la matrice  $n \times m$  di **allocazione corrente** di ogni processo.  
 $Assegnate_i$  ( $i$ -esima riga di *Assegnate*) indica le risorse allocate a  $P_i$
  - *Richieste*: la matrice  $n \times m$  delle **richieste** di ogni processo.  
 $Richieste_i$  ( $i$ -esima riga di *Richieste*) indica le risorse richieste da  $P_i$
- Invariante:  $\sum_{i=1}^n Assegnate_{ij} + Disponibili_j = Esistenti_j$  per  $j \geq 0$

27

## Operazioni sui vettori

Dati due vettori  $V = (V_1, \dots, V_m)$  e  $W = (W_1, \dots, W_m)$  definiamo

- $V \leq W$  sse  $V_i \leq W_i$  per  $i : 1, \dots, m$

Ad es.  $(1, 0, 4) \leq (2, 1, 6)$  ma  $(1, 0, 4) \not\leq (0, 1, 6)$

- $V \oplus W = (V_1 + W_1, \dots, V_m + W_m)$

$(1, 0, 4) \oplus (2, 1, 6) = (3, 2, 10)$

- $V \ominus W = (V_1 - W_1, \dots, V_m - W_m)$

$(6, 1, 4) \ominus (2, 1, 3) = (4, 0, 1)$

- (Nota: useremo  $\ominus$  per l'algoritmo del banchiere tra qualche lucido)

28

## Algoritmo di identificazione - Idea

- L'algoritmo funziona nel seguente modo:
  - si marcano (attraverso un array *Fine*) i processi che possono terminare (cioè non sono in stallo), i.e., cioè tali che il vettore di richieste è minore del vettore di risorse disponibili  $R_i \leq A$
  - le risorse dei processi marcati  $C_i$  vengono aggiunte al vettore di risorse disponibili  $A$  (il processo le rilascia)
  - se alla fine ci sono processi non marcati allora essi sono in stallo

29

## Algoritmo di identificazione - Pseudo codice

1. Per ogni  $i = 1, \dots, n$   $Fine[i] := false$  se  $Assegnate_i \neq (0, \dots, 0)$  altrimenti  $Fine[i] := true$ ;
2. Cerca un  $i$  tale che  $Fine[i] = false$  e  $Richieste_i \leq Disponibili$
3. Se esiste tale  $i$ :
  - $Disponibili = Disponibili \oplus Assegnate_i$
  - $Fine[i] = true$
  - Vai a 2.
4. Altrimenti, se esiste  $i$  tale che  $Fine[i] = false$ , allora  $P_i$  è in stallo.

L'algoritmo richiede  $O(m \times n^2)$  operazioni per decidere se il sistema è in deadlock.

30

## Uso degli algoritmi di identificazione

- Gli algoritmi di identificazione dei deadlock sono costosi
- Quando e quanto invocare l'algoritmo di identificazione? Dipende:
  - Quanto frequentemente può occorrere un deadlock?
  - Quanti processi andremo a "sanare" (almeno uno per ogni ciclo disgiunto)
- Diverse possibilità:
  - Ad ogni richiesta di risorse: riduce il numero di processi da bloccare, ma è molto costoso
  - Ogni  $k$  minuti, o quando l'uso della CPU scende sotto una certa soglia: il numero di processi in deadlock può essere alto, e non si può sapere chi ha causato il deadlock

31

## Risoluzione dei deadlock: Prerilascio

- In alcuni casi è possibile togliere una risorsa allocata ad uno dei processi in deadlock, per permettere agli altri di continuare
  - Cercare di scegliere la risorsa più facilmente "interrompibile" (cioè restituibile successivamente al processo, senza dover ricominciare daccapo)
  - Intervento manuale (sospensione/continuazione della stampa)
- Raramente praticabile

32

## Risoluzione dei deadlock: Rollback

- Inserire nei programmi dei *check-point*, in cui *tutto* lo stato dei processi (memoria, dispositivi e risorse comprese) vengono salvati (accumulati) su un file.
- Quando si scopre un deadlock, si conoscono le risorse e i processi coinvolti
- Uno o più processi coinvolti vengono riportati ad uno dei checkpoint salvati, con conseguente rilascio delle risorse allocate da allora in poi (*rollback*)
- Gli altri processi possono continuare
- Il lavoro svolto dopo quel checkpoint è perso e deve essere rifatto.
  - Cercare di scegliere i processi meno distanti dal checkpoint utile.
- Non sempre praticabile. Esempio: ingorgo traffico.

33

## Risoluzione dei deadlock: Terminazione

- Terminare uno (o tutti, per non far torto a nessuno) i processi in stallo
- Equivale a un rollback iniziale.
- Se ne terminiamo uno alla volta, in che ordine?
  - Nel ciclo o fuori dal ciclo?
  - Priorità dei processi
  - Tempo di CPU consumata dal processo, e quanto manca per il completamento
  - Risorse usate dal processo, o ancora richieste per completare
  - Quanti processi si deve terminare per sbloccare lo stallo
  - Prima i processi batch o interattivi?
  - Si può ricominciare daccapo senza problemi?

34

## Terzo approccio: Evitare dinamicamente i deadlock

Domanda: è possibile decidere al volo se assegnare una risorsa, evitando di cadere in un deadlock?

Risposta: sì, a patto di conoscere *a priori* alcune informazioni aggiuntive.

- Il modello più semplice ed utile richiede che ogni processo dichiari *fin dall'inizio* il numero *massimo* di risorse di ogni tipo di cui avrà bisogno nel corso della computazione.
- L'algoritmo di deadlock-avoidance esamina dinamicamente lo stato di allocazione delle risorse per assicurare che non ci siano mai code circolari.
- Lo *stato* di allocazione delle risorse è definito dal numero di risorse allocate, disponibili e dalle richieste massime dei processi.

35

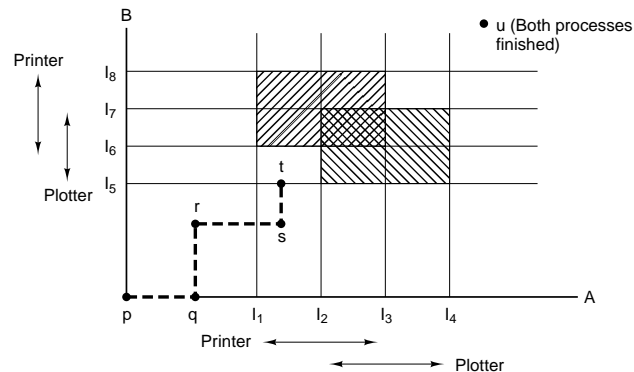
## Esempio

- Supponiamo di dover gestire due risorse, stampante e plotter, per due processi, A e B;  
Siano  $I_1, I_2, \dots, I_4$  le istruzioni di A e  $I_5, \dots, I_8$  le istruzioni di B
- Il processo A
  - richiede la stampante nell'istruzione  $I_1$  e la rilascia in  $I_3$
  - richiede il plotter nell'istruzione  $I_2$  e la rilascia in  $I_4$
- Il processo B
  - richiede il plotter nell'istruzione  $I_5$  e la rilascia in  $I_7$
  - richiede la stampante nell'istruzione  $I_6$  e la rilascia in  $I_8$

36

## Traiettorie di risorse

Vediamo in un diagramma una possibile esecuzione di  $A$  (linee orizzontali) e  $B$  (verticali)



Il S.O. deve portare  $A$  e  $B$  dal punto  $t$  al punto  $u$  senza incorrere nello stallo

37

## Considerazioni sul diagramma

- Vi sono 4 regioni: regioni sicure (senza tratteggio), regioni tratteggiate in cui il plotter o la stampante sono allocate ad entrambi i processi, regioni doppiamente tratteggiate in cui sia plotter che stampante sono allocate ad entrambi i processi (compresa tra  $I_6, I_7, I_2, I_3$ )
- Quando  $A$  passa la linea  $I_1$  ha ottenuto la stampante
- Quando  $B$  arriva alla linea  $I_5$  ha richiesto il plotter
- Se il sistema operativo fa entrare l'esecuzione nel quadrato  $I_1, I_2, I_5, I_6$  avremo un deadlock all'incrocio tra  $I_6$  e  $I_2$
- Al punto  $t$  l'unica cosa sicura da fare e' fare eseguire  $A$  fino a  $I_4$ . Dopo questo punto ogni traiettoria verso  $u$  evita lo stallo.

38

## Stati sicuri

- Quando un processo richiede una risorsa, si deve decidere se l'allocazione lascia il sistema in uno *stato sicuro*
- Lo stato è *sicuro* se esiste una *sequenza sicura* per tutti i processi.
- La sequenza  $\langle P_1, P_2, \dots, P_n \rangle$  è sicura se per ogni  $P_i$ , la risorsa che  $P_i$  può ancora richiedere può essere soddisfatta dalle risorse disponibili correntemente più tutte le risorse mantenute dai processi  $P_1, \dots, P_{i-1}$ .
  - Se le risorse necessarie a  $P_i$  non sono immediatamente disponibili, può aspettare che i precedenti finiscano.
  - Quando i precedenti hanno liberato le risorse,  $P_i$  può allocarle, eseguire fino alla terminazione, e rilasciare le risorse allocate.
  - Quando  $P_i$  termina,  $P_{i+1}$  può ottenere le sue risorse, e così via.

39

## Esempio

- Consideriamo tre processi che accedono ad una sola classe di risorse con 10 istanze
- Rappresentiamo il loro stato con una tabella in cui:
  - has: risorse possedute, max: massimo richiesto, free: risorse libere
- Lo stato (a) nella figura di seguito e' sicuro. Infatti la sequenza di scheduling  $B, C, A$  (stati (b - e)) e' sicura

	Has	Max		Has	Max		Has	Max		Has	Max		Has	Max
A	3	9	(a)	A	3	9	(b)	A	3	9	(c)	A	3	9
B	2	4	(a)	B	4	4	(b)	B	0	-	(c)	B	0	-
C	2	7	(a)	C	2	7	(b)	C	2	7	(c)	C	7	7
	Free: 3			Free: 1			Free: 5		Free: 0		Free: 7		Free: 0	

40

- Se ora *A* richiede e prende un'altra risorsa allora il nuovo stato (b) nella figura di seguito non e' sicuro (verificate)

- Ad esempio se lo scheduler esegue *B* (stati  $(c-d)$ ) *A* e *C* andranno in stallo

Has Max			Has Max			Has Max			Has Max		
A	3	9	A	4	9	A	4	9	A	4	9
B	2	4	B	2	4	B	4	4	B	—	—
C	2	7	C	2	7	C	2	7	C	2	7
Free: 3			Free: 2			Free: 0			Free: 4		
(a)			(b)			(c)			(d)		

### Osservazioni

- Se il sistema è in uno stato sicuro  $\Rightarrow$  non ci sono deadlock
- Se il sistema è in uno stato NON sicuro  $\Rightarrow$  possibilità di deadlock
- Deadlock avoidance: assicurare che il sistema non entri mai in uno stato non sicuro.

41

### Algoritmo del Banchiere (Dijkstra, '65)

Controlla se una richiesta può portare ad uno stato non sicuro; in tal caso, la richiesta non è accettata.

Ad ogni richiesta, l'algoritmo controlla se le risorse rimanenti sono sufficienti per soddisfare la massima richiesta di almeno un processo; in tal caso l'allocazione viene accordata, altrimenti viene negata.

Funziona sia con istanze multiple che con risorse multiple.

- Ogni processo deve dichiarare *a priori* l'uso massimo di ogni risorsa.
- Quando un processo richiede una risorsa, può essere messo in attesa.
- Quando un processo ottiene tutte le risorse che vuole, deve restituirle in un tempo finito.

42

### Algoritmo del Banchiere

- Strutture dati
  - $n$             numero di processi del sistema
  - $m$             numero di tipi di risorse
  - Disponibili*    vettore delle istanze disponibili (available) di ogni risorsa
  - Max*            matrice  $n \times m$  del numero massima di risorse richiedibili
  - Assegnate*     matrice  $n \times m$  delle risorse allocate per processo
  - Assegnate<sub>i</sub>* (*i*-esima riga di *Assegnate*) risorse assegnate a  $P_i$
  - Necessita'*    matrice  $n \times m$  delle risorse ancora richiedibili
  - Necessita'<sub>i</sub>* =  $Max_i - Assegnate_i$
  - Richieste*     matrice delle richieste considerate in un certo istante
  - Richieste<sub>i</sub>*    (*i*-esima riga di *Richieste*) richieste di  $P_i$

43

## Algoritmo del Banchiere

• Quando un processo  $P_i$  effettua una richiesta, tramite il vettore  $Richieste_i$ , l'algoritmo effettua i seguenti passi:

1. se  $Richieste_i \leq Necessita'_i$  vai al passo 2, altrimenti riporta errore ( $P_i$  ha superato il numero massimo di richieste);
2. se  $Richieste_i \leq Disponibili$  vai al passo 3, altrimenti  $P_i$  deve attendere la liberazione di altre risorse
3. Il sistema simula l'assegnamento a  $P_i$  delle risorse richieste modificando come segue lo stato di assegnamento

$$\begin{aligned} Disponibili &:= Disponibili \ominus Richieste_i; \\ Assegnate_i &:= Assegnate_i \oplus Richieste_i; \\ Necessita'_i &:= Necessita'_i \ominus Richieste_i; \end{aligned}$$

Se il nuovo stato e' **sicuro** la transazione viene completata e al processo si assegnano le risorse richieste. Altrimenti si ripristina il vecchio stato di assegnamento e  $P_i$  deve attendere la liberazione di altre risorse.

44

## Algoritmo di verifica della sicurezza

Vogliamo decidere se uno stato e' sicuro (cioe' esiste una sequenza sicura a partire da tale stato)

L'algoritmo esegue i seguenti passi lavorando su una variabile ausiliaria  $Aux$  inizialmente uguale a  $Disponibili$ :

1.  $Fine[i] = false$  per ogni  $i = 1, \dots, n$
2.  $Aux := Disponibili$
3. Cerca un  $i$  tale che  $Fine[i] = false$  e  $Necessita'_i \leq Aux$   
se tale  $i$  non esiste vai al passo 4;
  - $Aux := Aux \oplus Assegnate_i$
  - $Fine[i] := true$
  - Vai a 3.
4. Altrimenti, se  $Fine[i] = true$  per ogni  $i$ , allora il sistema e' in uno stato sicuro.

45

## Esempio dell'algoritmo del banchiere

Consideriamo  $n = 5$  processi  $p_0, \dots, p_4$  e  $m = 3$  classi di risorse  $A, B, C$  tali che

A ha 10 istanze  
B ha 5 istanze  
C ha 7 istanze

Inoltre al tempo  $T_0$  lo stato dell'assegnamento delle risorse e'

	Assegnate			Max	Disponibili		
	A	B	C	A	B	C	
							3 3 2
p0	0	1	0	7	5	3	
p1	2	0	0	3	2	2	
p2	3	0	2	9	0	2	
p3	2	1	1	2	2	2	
p4	0	0	2	4	3	3	

46

E' uno stato sicuro. Infatti la sequenza (p1, p3, p4, p2, p0) e' safe.

Sia  $Richieste_1 = (1, 0, 2)$  la richiesta di p1 all'istante T1. Se accettata il nuovo stato sarebbe:

	Assegnate			Necessita'	Disponibili		
	A	B	C	A	B	C	
							2 3 0
p0	0	1	0	7	4	3	
p1	3	0	2	0	2	0	
p2	3	0	2	6	0	0	
p3	2	1	1	0	1	1	
p4	0	0	2	4	3	1	

E' ancora sicuro. Le possibili sequenze sicure sono:

(p1, p3, p4, p2, p0) e

(p1, p4, p3, p0, p2)

Invece la richiesta  $Richieste_4 = (3, 3, 0)$  porterebbe ad uno stato non sicuro

## Algoritmo del Banchiere (Cont.)

- Soluzione molto studiata, in molte varianti
- Di scarsa utilità pratica, però.
- È molto raro che i processi possano dichiarare fin dall'inizio tutte le risorse di cui avranno bisogno.
- Il numero dei processi e delle risorse varia dinamicamente
- Di fatto, quasi nessun sistema usa questo algoritmo

47

## Quarto approccio: prevenzione dei Deadlock

Negare una delle quattro condizioni necessarie (Coffman et al, '71)

- Negare Mutua Esclusione
  - Le risorse condivisibili solitamente non devono garantire mutua esclusione (es. file in lettura)
  - Per alcune risorse non condivisibili, si può usare lo *spooling* per simulare l'accesso simultaneo (che comunque introduce competizione per lo spazio disco)
  - In generale tuttavia vi sono risorse non condivisibile per le quali non si può negare mutua esclusione

48

## Prevenzione dei Deadlock (cont)

- Negare Hold and Wait: garantire che quando un processo richiede un insieme di risorse, non ne richiede nessun'altra prima di rilasciare quelle che ha.
  - Richiede che i processi richiedano e ricevano tutte le risorse necessarie all'inizio, o che rilascino tutte le risorse prima di chiederne altre
  - Se l'insieme di risorse non può essere allocato in toto, il processo aspetta (metodo transazionale).
  - Basso utilizzo delle risorse
  - Possibilità di starvation

49

## Prevenzione dei Deadlock (cont)

- Negare l'assenza di prerilascio
  - Si potrebbero usare dei protocolli che forzano la preemption:
    - Se un processo richiede una risorsa che non è disponibile, si rilasciano tutte le risorse attualmente in suo possesso
    - Se un processo richiede una risorsa che non è disponibile ed è assegnata ad un processo che attende altre risorse, si sottrae la risorsa a quest'ultimo e si assegna al primo processo
- Questi protocolli possono essere usati solo per risorse in cui il salvataggio/ripristino di uno stato si può fare in maniera efficiente (es. CPU)

50

## Prevenzione dei Deadlock (cont)

- Impedire l'attesa circolare.
  - Prima possibilita'
    - \* permettere che un processo allochi al più una risorsa:
    - \* e' una condizione troppo restrittiva
  - Seconda possibilita': *Ordinamento delle risorse*
    - \* Si impone un ordine totale su tutte le classi di risorse
    - \* si richiede che ogni processo richieda le risorse nell'ordine fissato

51

## Ordinamento delle risorse

- Supponiamo che  $R = \{R_1, R_2, \dots, R_n\}$  sia l'insieme di classi di risorse.
- Definiamo una funzione iniettiva  $f : R \rightarrow Nat$  che definisce l'ordine.
- Ad esempio,
  - $f(\text{floppy}) = 1,$
  - $f(\text{dischi}) = 5,$
  - $f(\text{stampante}) = 12$

52

- Possibili Protocolli per prevenire attese circolari
  - Ogni processo inizialmente puo' richiedere qualsiasi numero di istanze di un tipo di risorsa ad es.  $R_i$
  - Dopo di che il processo puo' richiedere solo istanze della classe  $R_j$  se  $f(R_j) > f(R_i)$   
(ad es. se un processo deve impiegare stampante e floppy deve richiedere prima un'istanza di tipo floppy e poi la stampante)
  - Alternativa: prima di richiedere un'istanza di tipo  $R_i$  il processo deve rilasciare tutte le istanze di classi  $R_j$  con  $f(R_i) \geq f(R_j)$

- Se si utilizza uno di questi protocolli non si puo' verificare attesa circolare

### Dimostrazione

- Per assurdo supponiamo che  $P_0, \dots, P_n$  siano in attesa circolare dove  $P_i$  attende una risorsa di tipo  $R_i$  assegnata a  $P_{i+1}$ .
- Poiche' il processo  $P_{i+1}$  possiede  $R_i$  quando richiede  $R_{i+1}$  deve valere che  $f(R_i) < f(R_{i+1})$  per ogni  $i$ .
- Cioe'  $f(R_0) < f(R_1) < \dots < f(R_n) < f(R_0)$ , il che e' impossibile (contraddizione)

- Teoricamente fattibile, ma difficile da implementare:
  - l'ordinamento può non andare bene per tutti
  - ogni volta che le risorse cambiano, l'ordinamento deve essere aggiornato

## Approccio combinato alla gestione del Deadlock

- I tre approcci di gestione non sono esclusivi, possono essere combinati:
  - rilevamento
  - elusione (avoidance)
  - prevenzione
 si può così scegliere l'approccio ottimale per ogni classe di risorse del sistema.
- Le risorse vengono partizionati in classi ordinate gerarchicamente
- In ogni classe possiamo scegliere la tecnica di gestione più opportuna.

53

## Blocco a due fasi (two-phase locking)

- Protocollo in due passi, molto usato nei database:
  1. Prima il processo prova ad allocare tutte le risorse di cui ha bisogno per la transazione.
  2. Se non ha successo, rilascia tutte le risorse e riprova. Se ha successo, completa la transazione usando le risorse.
- È un modo per evitare l'hold&wait.
- Non applicabile a sistemi real-time (hard o soft), dove non si può far ripartire il processo dall'inizio
- Richiede che il programma sia scritto in modo da poter essere "rieseguito" daccapo (non sempre possibile)

54

## Sommario

### Detection

- Vantaggi
  - Nessun ritardo nell'inizializzazione dei processi
  - Facilita l'intervento on line
- Svantaggi
  - Perdita della possibilità di sfruttare la preemption possibile con certe risorse

55

## Prevention

- Vantaggi
  - Richiesta unica delle risorse:  
Lavora bene con processi che effettuano una singola fase di attività, non è necessaria preemption.
  - Preemption:  
Conveniente se lo stato delle risorse può essere salvato o non ha memoria
  - Ordinamento risorse:  
Può utilizzare controlli compile time  
I problemi sono risolti fuori dal programma

## • (Prevention) Svantaggi

- Richiesta unica delle risorse:  
Inefficiente  
Ritarda l'inizio dei processi
- Preemption:  
Soggetta a restart ciclico
- Ordinamento risorse:  
Non consente la richiesta incrementale delle risorse.

## Avoidance

- Vantaggi
  - Non è necessaria preemption
- Svantaggi
  - Devono essere note le massime richieste future
  - I Processi possono essere bloccati anche a lungo.