# Deadlock

Ivan Scagnetto

Università di Udine — Facoltà di Scienze MM.FF.NN.

A.A. 2007-2008

# Deadlock

- Molte risorse dei sistemi di calcolo possono essere usate in modo esclusivo.
- I sistemi operativi devono assicurare l'uso consistente di tali risorse.
- Le risorse vengono allocate ai processi in modo esclusivo, per un certo periodo di tempo. Gli altri richiedenti vengono messi in attesa.
- Ma un processo può avere bisogno di molte risorse contemporaneamente.
- Questo può portare ad attese circolari$\Rightarrow$ il deadlock (stallo).
- Situazioni di stallo si possono verificare su risorse sia locali sia distribuite, sia software sia hardware.
- È necessario avere dei metodi per prevenire, riconoscere o almeno risolvere i deadlock.

# Risorse

- Una risorsa è una componente del sistema di calcolo a cui i processi possono accedere in modo esclusivo, per un certo periodo di tempo.
- Risorse prerilasciabili: possono essere tolte al processo allocante, senza effetti dannosi. Esempio: memoria centrale.
- Risorse non prerilasciabili: non possono essere cedute dal processo allocante, pena il fallimento dell'esecuzione. Esempio: stampante.
- I deadlock si hanno con le risorse non prerilasciabili.

# Risorse

- Protocollo di utilizzo 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, fallback...)

# Allocazione di una risorsa

Si può disciplinare l'allocazione mediante dei semafori (tipica soluzione user-space): associamo un mutex alla risorsa.

```
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)

# Allocazione di più risorse

Più mutex, uno per ogni risorsa. Ma come allocarli?

```
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)

# 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.
- Non è detto neanche che i due programmi siano scritti dallo stesso utente: come coordinarsi?
- 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,
  - risoluzione di un deadlock.

# Il problema del 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 ed il genere delle risorse e delle richieste non sono influenti.

# Condizioni necessarie per il deadlock

Quattro condizioni necessarie (ma non sufficienti!) perché si possa verificare un deadlock [Coffman et al., 1971]:

1. Mutua esclusione: ogni risorsa è assegnata ad un solo processo, oppure è disponibile.
2. Hold&Wait: i processi che hanno richiesto ed ottenuto delle risorse, ne possono richiedere altre.
3. Mancanza di prerilascio: le risorse che un processo detiene possono essere rilasciate dal processo solo volontariamente.
4. Catena di attesa circolare di processi: esiste un sottoinsieme di processi $\{P_0, P_1, ..., P_n\}$ tali che $P_i$ è in attesa di una risorsa che è assegnata a $P_{i+1 \bmod n}$.

Se anche solo una di queste condizioni manca, il deadlock NON può verificarsi.
Ad ogni condizione corrisponde una politica che il sistema può adottare o no.

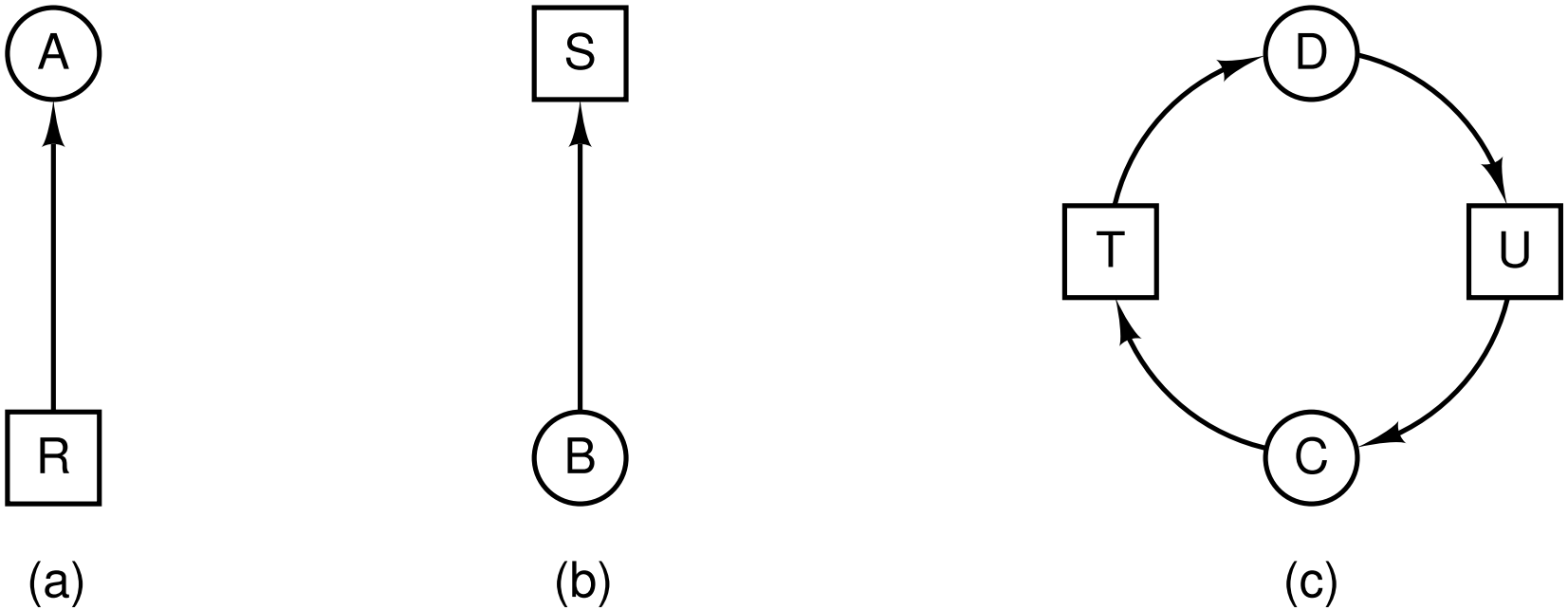
# Grafo di allocazione risorse

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:
  - $P = \{P_1, P_2, \ldots, P_n\}$, l'insieme di tutti i processi del sistema.
  - $R = \{R_1, R_2, \ldots, R_m\}$, l'insieme di tutte le risorse del sistema.
- archi di richiesta: archi orientati $P_i \longrightarrow R_j$
- archi di assegnamento (acquisizione): archi orientati $R_j \longrightarrow P_i$

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

# Grafo di allocazione risorse (cont.)



# Grafo di allocazione risorse (cont.)

A
Request R
Request S
Release R
Release S

(a)

B
Request S
Request T
Release S
Release T

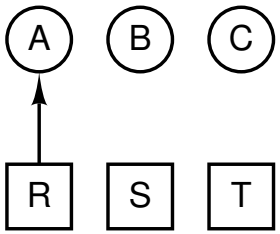
(b)

C
Request T
Request R
Release T
Release R

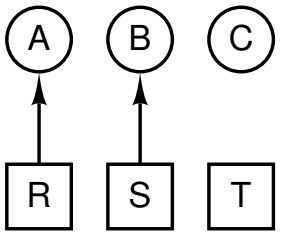
(c)

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

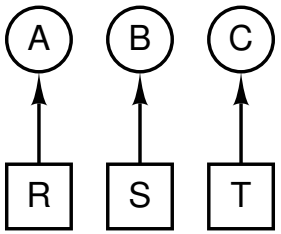
(d)



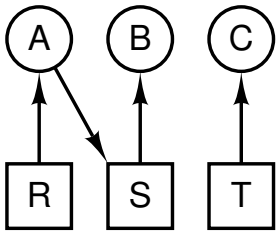

(e)



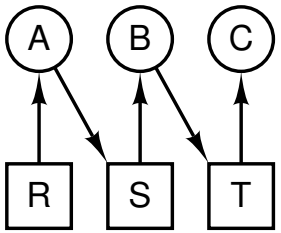

(f)



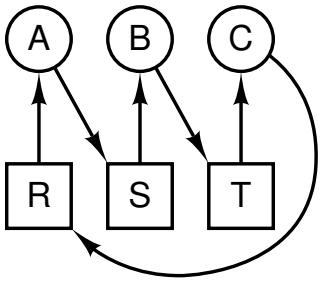

(g)




(h)




(i)



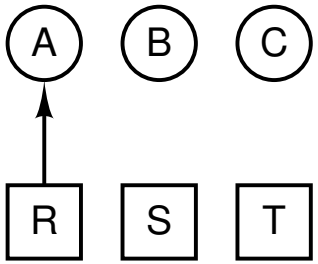

(j)

# Grafo di allocazione risorse (cont.)

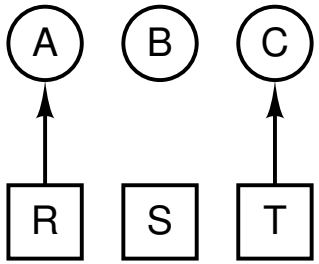
1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

no deadlock

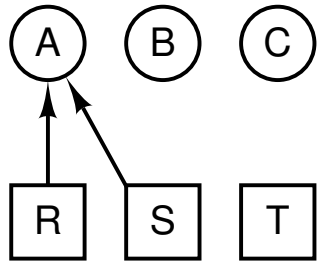
(k)



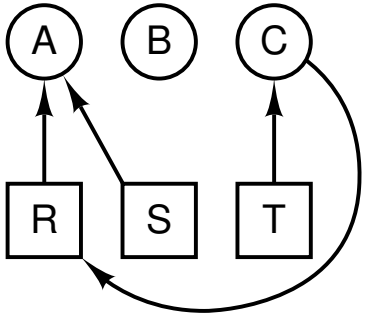

(l)



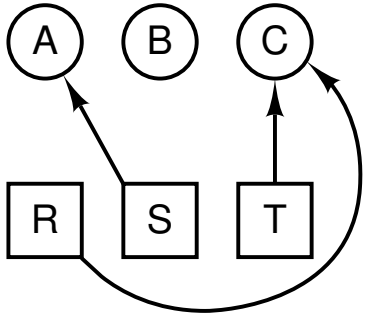

(m)



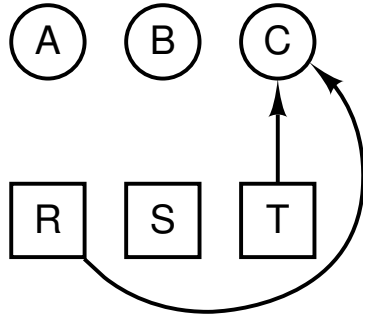

(n)




(o)




(p)




(q)

# 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.

# Uso dei grafi di allocazione delle 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.
- Per ogni sequenza può 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.

# Gestione dei Deadlock

In generale ci sono quattro possibilità:

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

# Primo approccio: ignorare il problema

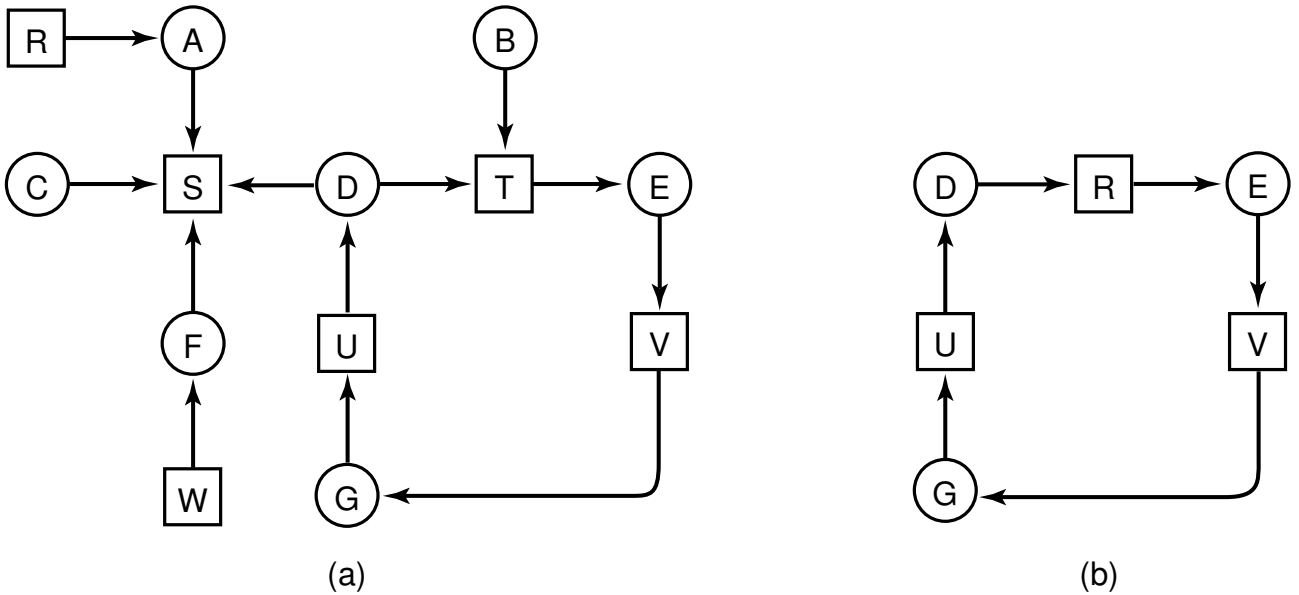
- Assicurare l'assenza di deadlock impone costi (in prestazioni, funzionalità) molto alti.
- Tali costi sono necessari in alcuni contesti, ma insopportabili in altri.
- Si considera il rapporto costo/benefici: se la probabilità che accada un deadlock è sufficientemente bassa, non giustifica il costo 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.

# Secondo 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.

# Algoritmo di identificazione: una risorsa per classe

- Esiste una sola istanza per ogni classe.
- Si mantiene un grafo di allocazione delle risorse.




- Si usa un algoritmo di ricerca di cicli per grafi orientati (v. ASD).
- Costo di ogni chiamata: $O(n^2)$, dove $n =$ numero nodi (= processi+risorse).

# Algoritmo di identificazione: più risorse per classe

Strutture dati:

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Invariante: per ogni $j = 1, \ldots, m$: $\sum_{i=1}^{n} C_{ij} + A_j = E_j$

# Algoritmo di identificazione di deadlock

1. $Finish[i] = false$ per ogni $i = 1, \ldots, n$
2. Cerca un $i$ tale che $R[i] \leq A$, ossia $\forall j : R_{ij} \leq A_j$
3. Se esiste tale $i$:
   - $Finish[i] = true$
   - $A = A + C[i]$ (cioè $A_j = A_j + C_{ij}$ per ogni $j$)
   - Vai a 2.
4. Altrimenti, se esiste $i$ tale che $Finish[i] = false$, allora $P_i$ è in stallo.

L'algoritmo richiede $O(m \times n^2)$ operazioni per decidere se il sistema è in deadlock (i.e., non esistono possibili schedulazioni safe).

# 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.
- Esistono diverse possibilità:
  - si richiama l'algoritmo ad ogni richiesta di risorse: questo approccio riduce il numero di processi da bloccare, ma è molto costoso;
  - si richiama l'algoritmo 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.

# 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 (ad esempio: sospensione/continuazione della stampa).
- Il prerilascio è raramente praticabile.

# 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 ed 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.

# Risoluzione dei deadlock: Terminazione

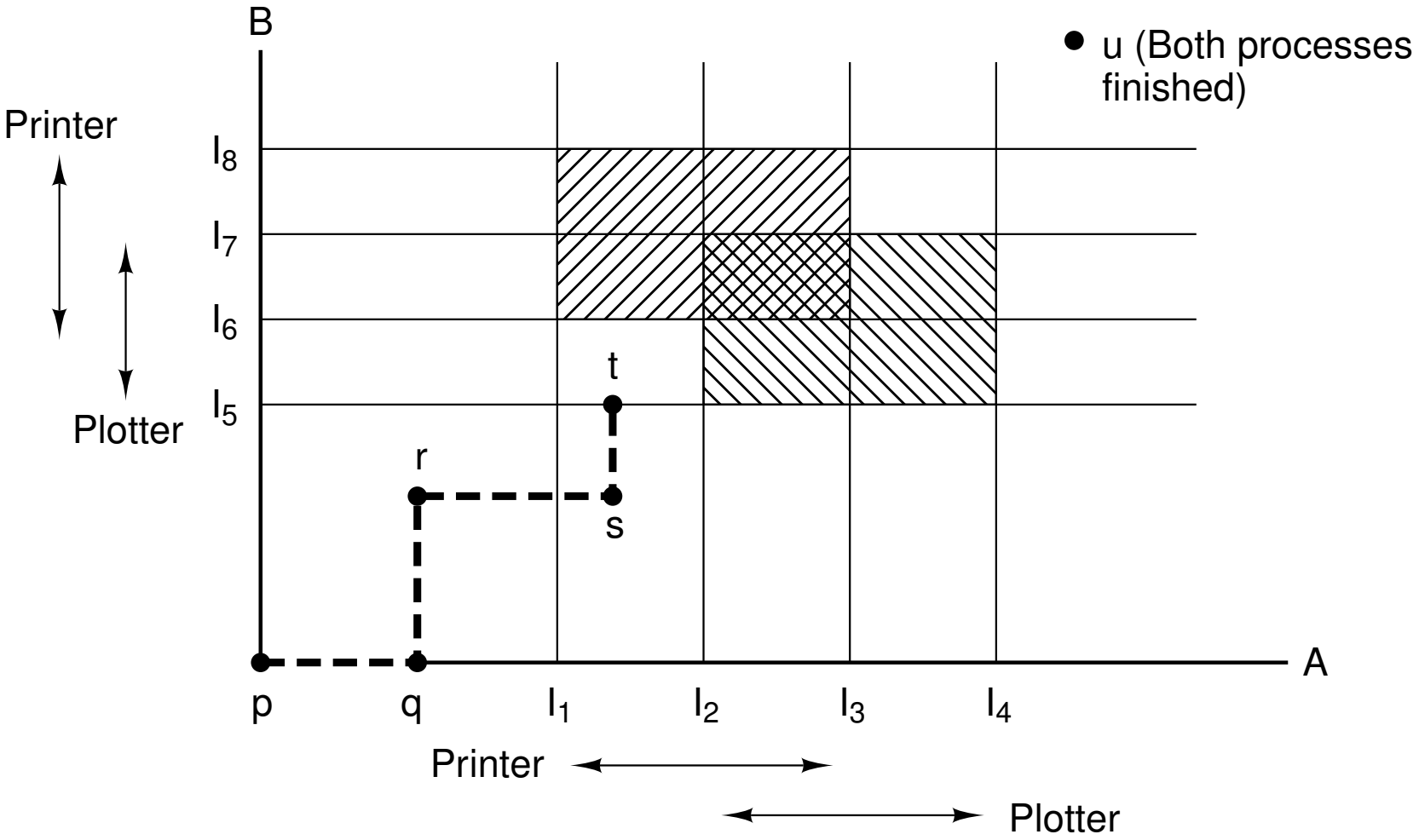
- Terminare uno (o tutti, per non far torto a nessuno) i processi in stallo.
- Equivale a un rollback iniziale.
- Se ne terminiamo uno alla volta, in che ordine?
  - Nel ciclo o fuori dal ciclo?
  - Priorità dei processi.
  - Tempo di CPU consumata dal processo, e quanto manca per il completamento.
  - 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?

# 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'algorito 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.

# Traiettorie di risorse



# 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 $\langle P_1, P_2, \ldots, P_n \rangle$ 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 $P_1, \ldots, P_{i-1}$.
  - 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.

# Esempio

## Sequenza sicura:

| | 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)

## Sequenza non sicura (lo stato (b) non è sicuro).

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 2 | 4 |
| C | 2 | 7 |

Free: 3
(a)

| | Has | Max |
|---|---|---|
| A | 4 | 9 |
| B | 2 | 4 |
| C | 2 | 7 |

Free: 2
(b)

| | Has | Max |
|---|---|---|
| A | 4 | 9 |
| B | 4 | 4 |
| C | 2 | 7 |

Free: 0
(c)

| | Has | Max |
|---|---|---|
| A | 4 | 9 |
| B | — | — |
| C | 2 | 7 |

Free: 4
(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.

# 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.

# Esempio dell'algoritmo del banchiere per risorsa singola

| | 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)

Cosa succede se in (b), si allocano 2 istanze a B?

# 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.

# Quarto approccio: prevenzione dei Deadlock

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

- Mutua Esclusione.
  - Le risorse condivisibili non hanno questo problema.
  - Per alcune risorse non condivisibili, si può usare lo spooling (che comunque introduce competizione per lo spazio disco).
  - Regola di buona programmazione: allocare le risorse per il minor tempo possibile.

# Prevenzione dei Deadlock (cont)

- Hold and Wait: garantire che quando un processo richiede un insieme di risorse, non ne richieda nessun'altra prima di rilasciare quelle che ha.
  - Necessita 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.
- Negare la mancanza di prerilascio: impraticabile per molte risorse.

# Prevenzione dei Deadlock (cont)

- Impedire l'attesa circolare.
  - Permettere che un processo allochi al più 1 risorsa: molto restrittivo.
  - 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,
    - un processo che detiene la risorsa $j$ non può mai chiedere una risorsa $i < j$, e quindi non si possono creare dei cicli.
  - 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 partizionate in classi ordinate gerarchicamente.
- In ogni classe possiamo scegliere la tecnica di gestione più opportuna.

# 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).