1 Lezione del 06-11-25

1.0.1 Cicli su grafi di allocazione

Ricordiamo i fatti di base sul rilevamento di deadlock attraverso l'ispezione dei **grafi di allocazione**.

- Se il grafo non contiene cicli, abbiamo detto non c'è possibilità di deadlock;
- Se il grafo contiene cicli, dobbiamo discriminare sulla presenza di *istanze multiple* di risorsa:
  - Se si ha una sola istanza per tipo di risorsa, allora si ha deadlock;
  - Se si hanno più istanze per tipo di risorsa, allora c'è la possibilità, ma non la sicurezza, di avere deadlock.

1.0.2 Metodi di gestione dei deadlock

Ricordiamo quindi i metodi che avevamo previsto per gestire i deadlock:

- Potremmo assicurarci che il sistema non entri mai in uno stato di deadlock: questo è il modello di **deadlock prevention** (*prevenzione statica*).

  Questo è l'approccio usato dalla maggior parte dei sistemi operativi, che preferiscono lasciar eseguire i processi senza attivarsi in maniera dinamica per gestire eventuali deadlock. Di contro, si spera (e si progettano sistemi in maniera tale) che tali situazioni non si verifichino;

- Potremmo permettere al sistema in normale operazione di trovarsi in stato di deadlock, rilevare tale deadlock e iniziare delle procedure di recupero: questo è il modello di **deadlock prevention** e **avoidance** (*prevenzione dinamica*).

  Questo modello è più usato nei sistemi embedded, dove si possono fare ipotesi più stringenti sui processi in esecuzione.

1.0.3 Prevenzione dinamica

Vediamo quindi nel dettaglio la prevenzione dinamica di deadlock. In questo caso dobiamo permettere al sistema di avere a disposizione alcune informazioni a priori:

- Nel caso più semplice vogliamo che ogni processo dichiari il numero massimo di risorse di ogni tipo di cui ha bisogno;
- Un **algoritmo** di prevenzione statica deve essere messo in piedi, e deve eseguire in maniera dinamica per prevenire situazioni di attese circolari. Il più celebre di questi algoritmi è l'*algoritmo del banchiere*;
- Lo **stato** dell'allocazione di risorse è definito dal numero di risorse disponibili ed allocate, e dal numero massimo di risorse che i processi possono richiedere (che abbiamo detto il S/O deve sapere).

1.0.4 Stato sicuro

Definiamo cosa intendiamo per **safe state** (o *stato sicuro*) del sistema.

Ogni volta che un processo richiede una risorsa disponibile, il S/O deve decidere se l'allocazione immediata di tale risorsa lascia il sistema in uno stato sicuro.

Il sistema si dice in stato sicuro se esiste una sequenza $\{p_0, p_1, ..., p_n\}$ di tutti processi tale che per ogni p_i, le risorse che p_i può richiedere possono essere allocate con le risorse correntemente disponibili più le risorse allocate ai p_j con $j < i$.

Questo significa che:

- Se p_i ha bisogno di risorse e queste sono disponibili, non c'è problema;
- Se p_i ha bisogno di risorse e queste non sono disponibili, può:
  1. Aspettare che tutti i p_j terminino, liberando le risorse di cui ha bisogno;
  2. Eseguire quando i p_j terminano e le risorse sono libere, allocando e quindi liberando nuovamente le sue risorse;
  3. Quando p_i termina liberando le sue risorse, p_{i+1} può eseguire, e così via.

Possiamo corredare i fatti di base di 15.0.1 con alcuni altri fatti, riguardanti lo stato safe:

- Se il sistema è in stato safe, non c'è possibilità di deadlock;
- Se il sistema non è in stato safe, c'è possibilità, ma non la sicurezza, di avere deadlock.

La deadlock *avoidance* corrisponde esattamente a evitare che il sistema arrivi a stati unsafe.

Vediamo un caso di esempio tipico secondo il modello dello stato sicuro. Poniamo di avere 2 processi (p_0 e p_1), interessati ad ottenere in maniera concorrente 2 risorse (r_0 e r_1):

- Processo p_0:

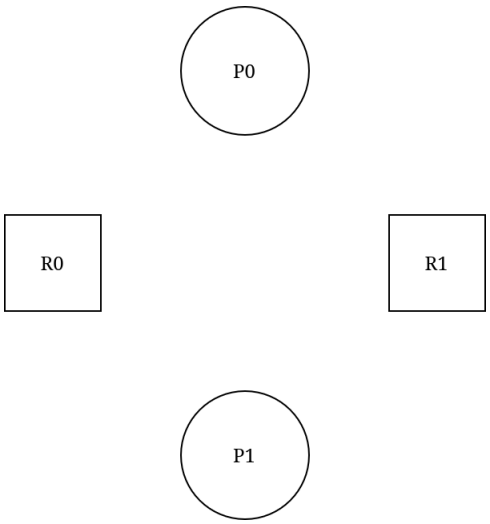
```
// p0, p1
wait(m0);
wait(m1);

// elaborazione su r0 e r1

signal(m1);
signal(m0);
```

- Processo p_1:

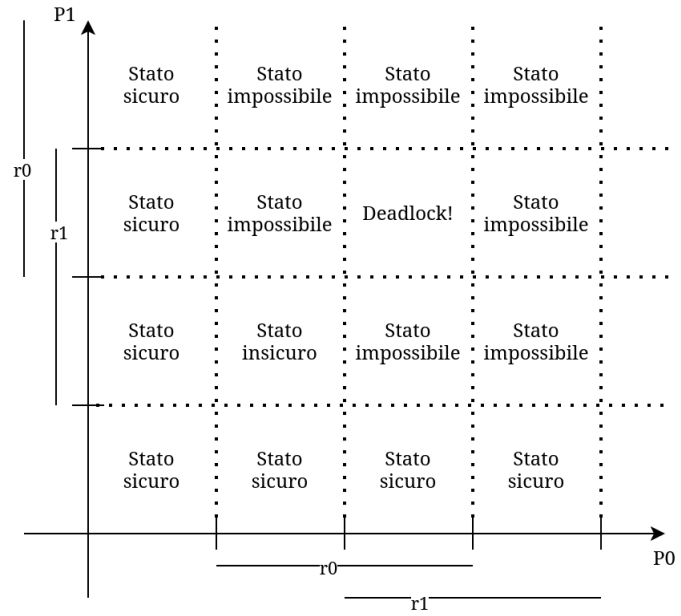
```
// p0, p1
wait(m1);
wait(m0);

// elaborazione su r0 e r1

signal(m0);
signal(m1);
```

Notiamo che i processi accedono ai mutex delle risorse in ordine inverso.

Il grafo di allocazione è il solito che abbiamo già visto:

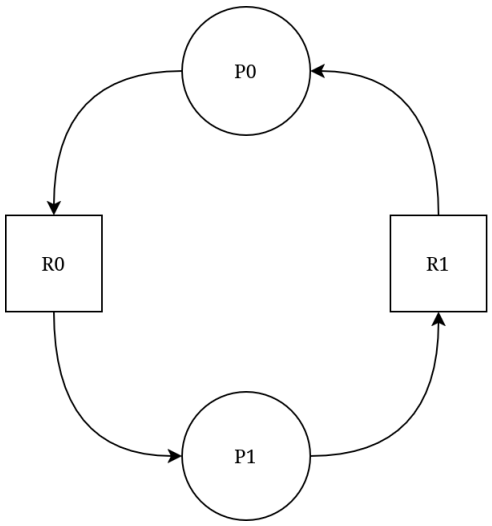


Possiamo modellizzare la stessa situazione, in evoluzione nel tempo, attraverso un grafico di questo tipo:



Dove notiamo l'evoluzione temporale (sui 2 assi) dei 2 processi, in funzione delle risorse acquisite, e gli stati corrispondenti che incontriamo. Vediamo come appena i 2 processi acquisiscono rispettivamente la risorsa r_0 e r_1, andiamo a trovarci nello stato insicuro, a cui segue un deadlock appena provano ad accedere alla risorsa seguente.

Nel grafo di allocazione questo è il deadlock:



1.0.5 Grafici di allocazione estesi

Estendiamo i grafi di allocazione, che abbiamo finora trattato in maniera abbastanza informale.

Avevamo che i cerchi rappresentano processi, e i quadrati risorse. Riguardo agli archi che li collegano, abbiamo:

- Un arco di *claim* da un processo a una risorsa simboleggia che la risorsa è desiderata dal processo, e potrebbe essere richiesta. Lo rappresentiamo come una freccia tratteggiata;
- Un arco di *claim* si trasforma in arco di *request* quando il processo effettivamente richiede la risorsa. Lo rappresentiamo come una freccia piena;
- Un arco di *assignment* (o allocazione) è diretto da una risorsa a un processo (il contrario degli archi di claim e request), e simboleggia che la risorsa è effettivamente posseduta dal processo. Lo rappresentiamo sempre come una freccia piena.

A questo punto il significato di un algoritmo di deadlock avoidance dovrebbe essere chiaro: se le transizioni in fase di allocazione di risorse consistono nella trasformazione di arco di request a un arco di assignment, allora dobbiamo assicurarci che le richieste vengano soddisfatte solo quando l'arco di assignment formato non porta alla formazione di cicli nel grafo di allocazione.

1.1 Algoritmo del banchiere

Vediamo quindi l'algoritmo più celebre di deadlock avoidance: il cosiddetto **algoritmo del banchiere**.

Le ipotesi dell'algoritmo sono:

- Esistono istanze multiple di risorse (nel caso banale, una sola istanza);
- Ogni processo deve fare *claim* a priori delle risorse massime che potrebbe usare;
- Ogni processo deve essere disposto ad aspettare per le sue risorse;
- Quando un processo ottiene tutte le risorse necessarie alla sua esecuzione, deve eseguire e restituirle in un lasso finito di tempo.

Definiamo le condizioni di stato dell'algoritmo. Sia n il numero di processi e m il numero di tipi di risorse.

- Le risorse **disponibili** saranno un vettore di interi di lunghezza m. Se la risorsa all'indice j vale k, significa che ci sono k istanze della risorsa corrispondente disponibili;
- Per ogni processo dobbiamo sapere la quantità **massima** di risorse che vorrà allocare. Rappresentiamo questo con una matrice bidimensionale $n \times m$. Se questa all'indice (i, j) vale k, allora il processo i potrà richiedere al massimo k risorse di tipo j;
- Manteniamo una matrice simile, di risorse **allocate**. Se questa all'indice (i, j) vale k, allora il processo i avrà allocato k risorse di tipo j;
- Infine, manteniamo un'altra matrice simile, di risorse **desiderate**. Se questa all'indice (i, j) vale k, allora il processo i avrà bisogno (oltre a quelle che già ha), di k risorse di tipo j;

Definiamo quindi le matrici e i vettori come *Avail*, *Max*, *Alloc* e *Need*. Viene da sé che:

$$Need[i, j] = Max[i, j] - Alloc[i, j]$$

Vediamo allora un primo algoritmo, di *sicurezza*, che controlla se il sistema si trova in uno stato sicuro.

1. Siano *Work* e *Finish* vettori di lunghezza rispettivamente m e n. Inizializza:
   - $Work = Avail$
   - $Finish[i] =$ `false` per $i = 0, 1, ..., n - 1$
2. Trova un i tale che:
   - $Finish[i] =$ `false`
   - $Need[i] \leq Work$

   se non esiste nessun i che soddisfa le condizioni, vai al passo 4;
3. Poni:
   - $Work = Work + Alloc$
   - $Finish[i] =$ `true`

   quindi vai al passo 2;
4. Se $Finish ==$ `true` per ogni i, allora il sistema è in stato sicuro.

Vediamo allora l'algoritmo del banchiere vero e proprio. Sia *Request* il vettore richiesta per il processo all'indice i. Se $Request[j] = k$ allora il processo i vuole k istanze della risorsa di tipo j.

1. Controlla che $Request[i] \leq$ Need$[i]$. Se il controllo passa, vai al passo 2. Altrimenti solleva un errore, in quanto il processo ha richiesto più risorse di quante ha dichiarato di volere;

2. Se $Request[i] \leq Avail$, vai al passo 3. Altrimenti aspetta, in quanto alcune risorse non sono disponibili e devono essere liberate dai processi precedenti;
3. Simula l'allocazione delle risorse al processo i modificando lo stato come segue:
   - $Avail = Avail - Request$
   - $Alloc[i] = Alloc[i] + Request$
   - $Need[i] = Need[i] - Request$

   A questo punto esegui l'algoritmo di sicurezza.
   - Se lo stato è sicuro, non cambiare nulla ed alloca le risorse al processo i;
   - Se lo stato non è sicuro, riporta lo stato a prima della simulazione e aspetta.