1 Lezione del 11-11-25

1.1 Prevenzione dinamica, deadlock detection

Visto l'algoritmo del banchiere, celebre algoritmo di deadlock *avoidance*, veniamo allo studio della deadlock detection.

Nell'esempio più semplice di deadlock *detection*, dove esiste una singola istanza di ogni tipo di risorsa, vogliamo mantenere un grafo di allocazione delle risorse. Periodicamente, invocheremo un algoritmo che analizza il grafo per trovare un ciclo. Se esiste un ciclo, esiste un deadlock (dai fatti in 15.0.1). Ricordiamo adesso che trovare cicli in un grafo di n nodi richiede un'algoritmo $O(n^2)$.

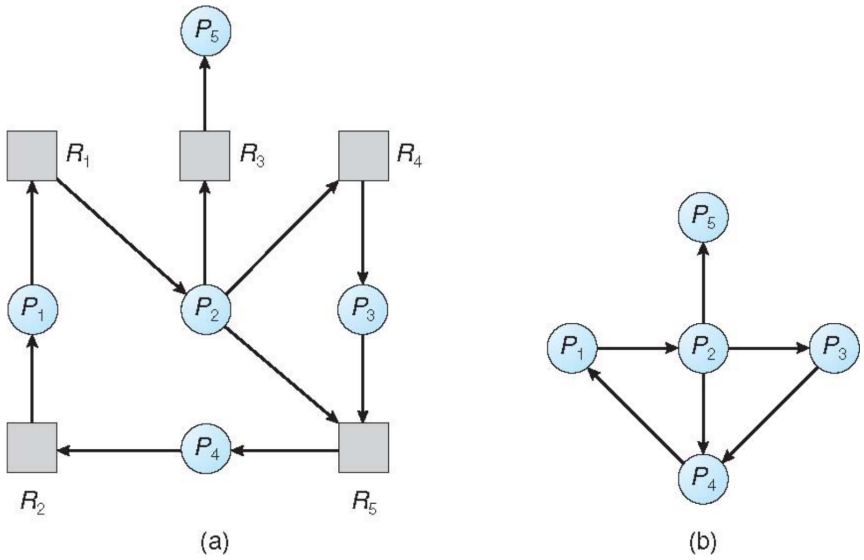
In ogni caso, quando si rileva un deadlock sarà opportuno invocare un qualche altro algoritmo, detto di deadlock *recovery*, per risolvere il deadlock.

1.1.1 Grafo di attesa

Il grafo che manteniamo in un algoritmo di deadlock detection non è propriamente un grafo di allocazione, ma un cosiddetto grafo di **attesa** (o grafo *wait-for*).

In questo caso l'informazione che vogliamo allocare non è l'attesa sulle *risorse*, ma sui processi che competono per risorse. La conversione da grafo di allocazione a grafo di attesa è banale: basta sostituire le triple freccia-risorsa-freccia con singole frecce dirette nello stesso senso.

Questo è meglio spiegato dal seguente grafico, che mostra un grafo di allocazione a sinistra, e a destra il corrispondente grafo di attesa:



Resource-Allocation Graph Corresponding wait-for graph

Il grafo di attesa è meno espressivo del grafo di allocazione, ma contiene comunque tutte le informazioni necessarie a fare deadlock detection. Infatti, se il grafo di allocazione contiene cicli, il grafo di attesa conserva tali cicli.

1.1.2 Algoritmo di deadlock detection

Potremmo interrogarci su quando è conveniente invocare un determinato algoritmo di deadlock detection.

Abbiamo introdotto in 16.1 che questo avrà necessariamente complessità $O(n^2)$ (in quanto deve scansionare un grafo per cicli).

- Chiaramente, eseguirlo troppo spesso (come ad esempio ad ogni allocazione di risorsa, cioè nei momenti in cui il grafo di attesa cambia e si verifica la possibilità di deadlock) porterebbe ad un'overhead troppo consistente;
- Un approccio simile al precedente è quello di effettuare deadlock detection ogni volta che non si riesce a soddisfare una richiesta di allocazione di risorse. Notiamo però che questo causa esecuzioni inutili quando il sistema ha, effettivamente, finito le risorse;
- Un buon approccio è quello di valutare l'utilizzo dellla CPU, cioè quanto tempo della CPU viene effettivamente utilizzato, e quanto sprecato in attesa di risorse. A questo punto, decidiamo di mettere in esecuzione l'algoritmo di deadlock detection quando il tempo di utilizzo è minore di una certa soglia (magari il $40 - 60\%$).
- Infine, l'approccio meno drastico e con minore overhead è quello di eseguire l'algoritmo periodicamente, a prescindere dall'utilizzo CPU o dalle risorse allocate.

1.1.3 Detection in più istanze

Abbiamo fatto, in 16.1, una semplificazione per il nostro algoritmo di deadlock detection: quella di assumere che tutte le risorse esistano in singola istanza. Avevamo visto da 15.0.1 che questo significa che un ciclo è causa sufficiente per un deadlock. Vediamo come si possono implementare algoritmi di deadlock che funzionano in presenza di istanze multiple di risorsa, cioè quando questa è solo causa necessaria.

Vogliamo comportarci come nell'algoritmo del banchiere, e quindi mantenere, 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;
- Manteniamo una matrice 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 **richieste**. Se questa all'indice (i, j) vale k, allora il processo i sta richiedendo (oltre a quelle che già ha), k risorse di tipo j.

Definiamo quindi le matrici e i vettori come *Avail*, *Alloc* e *Request*.

Notiamo che queste sono in qualche modo corrispondenti alle variabili di stato viste in 15.2 (dove si è discusso l'algoritmo del banchiere), se non per il fatto che l'equazione che legava *Need* a *Max* e *Alloc* non è più presente. Questo è chiaro dal fatto che non abiamo nessuna indicazione delle risorse massime richieste da un processo, e che questo potrebbe richiederne ancora, teoreticamente all'infinito.

In verità, la matrice *Need* non è propriamente duale alla *Request*: se la *Need* mantiene uno stato noto a priori e che varia in fase di allocazione di risorse da parte di un processo, la *Request* varia nel tempo sulla base del comportamento del processo (varia sostanzialmente in fase di chiamata di primitive di allocazione).

L'algoritmo a questo punto è molto simile all'algoritmo di sicurezza del banchiere:

1. Siano *Work* e *Finish* vettori di lunghezza rispettivamente m e n. Inizializza:

   - *Work* $=$ *Avail*
   - *Finish*$[i]$ $=$ `false` se *Alloc*$[i] \neq 0$, altrimenti `true`, per $i = 0, 1, ..., n-1$. Questa condizione non equivale strettamente al fatto che il processo è terminato. Piuttosto, significa che il processo non ha risorse allocate, per cui non può in nessun modo essere parte di un ciclo di deadlock.

2. Trova un i tale che:

   - *Finish*$[i]$ $=$ `false`
   - *Request*$[i] \leq$ *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* $==$ `false` per qualche i, allora il processo i è in deadlock.

1.2 Prevenzione dinamica, deadlock recovery

Interroghiamoci quindi su cosa fare in fase di rilevamento di deadlock.

Se l'algoritmo di detection è invocato su base arbitaria, potrebbero esserci molti cicli nel grafo di attesa, e quindi potremmo non essere in grado di capire quali dei processi in deadlock hanno in qualche modo *"provocato"* il deadlock.

L'operazione fondamentale di recupero dal deadlock è quella di **rollback** del processo, cioè riportare il processo ad uno stato precedente a quando il deadlock si è verificato. Un'altra opzione, molto più brutale, è quella di abortire il processo coinvolto.

Possiamo comunque interrogarci su *quale* processo (o processi) fare rollback o abort.

- La soluzione drastica è quella di influenzare tutti i processi coinvolti nel deadlock. Questo chiaramente risolve i problemi ma è distruttivo per il sistema;

- Una soluzione più intelligente potrebbe essere quella di influenzare un processo per ogni ciclo disgiunto trovato nel grafo di attea.

  In questo caso dobbiamo però simulare lo stato ottenuto facendo rollback (o abort) di ogni processo, in modo da assicurarci che la *vittima* selezionata sia effettivamente quella che causa il deadlock.

  Un buon approccio a questo metodo è quello di definire una certa *metrica* per la scelta dei processi vittima. Ad esempio, potremmo iniziare a selezionare vittime classificando per:

  – La priorità del processo;
  – Il tempo per cui il processo ha eseguito, e il tempo rimanente fino alla terminazione (vedere 6.2.5);
  – Le risorse che il processo ha già allocato;

  - Le risorse che servono al processo per terminare;
  - Il numero di processi da terminare (preferire di terminare meno processi possibile);
  - Se il processo è interattivo o batch.

Notiamo tutte queste metodologie rischiano sempre la *starvation*: un processo potrebbe essere sempre vittima di abort o rollback, e quindi non riuscire mai a terminare.

1.3 Gestione della memoria

Veniamo quindi alla gestione della seconda risorsa più importante dopo la CPU, cioè la **memoria** principale a disposizione del calcolatore.

L'idea è di offrire a tutti i processi il loro spazio di indirizzamento locale, quindi organizzare una risorsa fisica in più risorse logiche. Vedremo poi come potrebbe essere opportuno definire meccanismi più sofisticati, come divisione di memoria fra S/O e processi, e condivisione di memoria (appunto, *memoria condivisa*) fra più processi.

Gestione della memoria e gestione della CPU hanno dei parallelismi:

- Nella CPU offrivamo più CPU virtuali mantenendo il contesto dei processi nel **PCB** (*Process Control Block*).

  Nella memoria vorremo offrire più memorie virtuali, allocando altre apposite strutture dati (*descrittori*) che mantengano il contesto relativo al singolo processo;

- In particolare, potremmo voler implementare meccanismi di *swap-in* e *swap-out* che permettono di spostare i contenuti della memoria principale nella memoria secondare (disco rigido o simili), quando la prima risulta satura. Questo processo è effettivamente parallelo al cambio di contesto CPU.

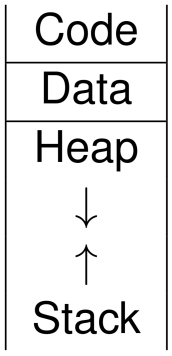
Esistono però anche differenze fra gestione della memoria e gestione della CPU:

- La gestione della CPU è atomica, cioè tutta la CPU viene allocata ad un singolo processo (a meno di approcci multiprocessore, per cui abbiamo ampiamente discusso meccanismi di sincronizzazione).

  Nella gestione della memoria, però, è fondamentale che più parti della memoria vengano allocate a processi diversi, per cui è necessario fin da subito prevedere meccanismi di *protezione* della memoria di processi dall'accesso di altri processi.

1.3.1 Immagine di un processo

Abbiamo detto che ogni processo vorrà vedere il suo *spazio di indirizzamento* privato e locale. Vediamo nel dettaglio come questo spazio potrebbe essere organizzato:



- Vogliamo mantenere un primo segmento dedicato al **codice**, cioè al programma vero e proprio in esecuzione nel processo. Si assume che questa si troverà all'inizio dello spazio di indirizzamento, e conterrà l'entry point del programma. Questa sezione conterrà inoltre i **dati in sola lettura** del programma (in quanto si assume che sia il codice che questi dovranno essere in sola lettura, come poi vedremo);
- Vogliamo quindi mantenere un segmento per i **dati** statici del programma, incluse le variabili inizializzate (**dati in lettura/scrittura**) e inizializate a 0 (**BSS**, *Block Started by Symbol*);
- Vorremo poi mantenere un segmento per l'**heap**, cioè la regione di memoria allocata in memoria dinamica;
- Infine vorremo un segmento per lo **stack**, che come sappiamo si sviluppa dall'alto verso il basso (e quindi si sviluppa a partire dal fondo allo spazio di indirizzamento).

Potrebbe essere utile ripercorrere i passaggi che la nostra toolchain compie in fase di compilazione di un programma, per generare appunto un'immagine di queste sezioni da poi caricare in memoria.

- Il *compilatore* si occupa di prendere il codice sorgente del programma e tradurlo in *oggetti*. Dobbiamo ricordare che gli oggetti non contengono indirizzi veri e propri nello spazio di indirizzamento processo, ma indirizzi *simbolici*, cioò non ancora risolti e che puntano a segmenti magari adesso nemmeno definiti;
- Il *linker* (*collegatore*) si occupa di prendere gli oggetti da noi generati, e quelli già presenti nel sistema sotto forma di *librerie*, e appunto collegarli in un file eseguibile. Solo in questo passaggio i segmenti vengono preparati, e gli indirizzi simbolici vengono tradotti in indirizzi reali nello spazio di indirizzamento di processo;
- Infine, un *loader* (*caricatore*) si occuperà di prendere i segmenti preparati dal linker e caricarli in memoria (magari impostando come lettura/scrittura gli opportuni segmenti, e azzerando i segmenti di bss, ecc...). A questo punto basterà spostare il PC all'entry point del programma ed eseguire.

1.4 Rilocazione statica/dinamica

Vediamo quindi a come si può svolgere la gestione della memoria vera e propria, cioò come il caricatore può comportarsi quando gli viene fornita l'immagine di un processo.

- L'approccio più semplice è quello della rilocazione **statica**. In questo caso si prevede un caricatore *rilocante*, cioè che si occupa di prendere l'immagine del programma e allocarla in memoria a partire da un certo indirizzo, detto **base**. Chiaramente dovrà preoccuparsi di calcolare il contenuto del program counter (aggiungendo all'entry point la base), e degli indirizzi fisici del programma (ancora, aggiungendo a tali indirizzi la base).

  Il problema di tale approccio è che è limitante per quanto riguarda la rilocazione successiva dell'immagine di processo (magari a causa di uno swap-out e successivo swap-in): in questo caso saremo costretti a fare qualcosa di indicibile come ricalcolare gli indirizzi in memoria a tempo di esecuzione (praticamente impossibile), o ricaricare il processo esattamente nello stesso posto di prima (cosa che rende abbastanza inutile prevedere lo swap-out in primo luogo);

- Decidiamo quindi usare un approccio a rilocazione **dinamica**. In questo caso dobbiamo prevedere un nuovo componente hardware, detto **MMU** (*Memory Management Unit*). La MMU ha il compito di implementare una qualche funzione di traduzione da indirizzo *virtuale* a indirizzo *fisico*:

  $$\texttt{MMU\_translate}(\text{virt}) = \text{phys}$$

  In questo modo il processore può continuare a pensare ai suoi indirizzi virtuali (che sono ad esso locali), e il compito di traduzione effettiva da questi a indirizzi fisici in memoria è delegato alla MMU, così che i processi non debbano preoccuparsi di dove si trovano effettivamente i loro dati, ma possono specificare indirizzi relativi al loro spazio di indirizzamento locale.

  L'implementazione più semplice della MMU è quella parallela all'esempio della rilocazione statica appena fatta. Prevediamo infatti la presenza di un registro **base** e un registro **limite**: quello che la MMU farà sarà controllare che i registri (virtuali) forniti dal processore non cadano fuori dal limite, e quindi sommarvi il registro base. Il S/O avrà il compito di impostare tali registri per ogni processo (in fase di cambio contesto), e così avremo il comportamento della rilocazione statica senza mai dover agire sugli indirizzi fisici nel codice del programma.

  Tralasciamo per adesso le complicazioni date dal fatto che il sistema operativo stesso deve accedere alla memoria attraverso la MMU, e quindi subendo traduzioni di indirizzi (solitamente si prevede una finestra, la finestra **FM**, che permette l'accesso diretto a tutta o una parte rilocabile della memoria fisica con traduzioni identità o identità con offset).

1.4.1 Memoria unica/segmentata

Un'altra distinzione ortogonale nella gestione della memoria è data da come si gestisce lo spazio di memoria fisico.

- La memoria è gestita come **unica** (o *flat*) quando si fornisce ad ogni processo una sezione contigua (o non contigua, ma comunque vista come uno o più blocchi contigui) di memoria, poi suddivisa nei vari segmenti (codice, dati, ecc...). Questo è l'approccio usato ad esempio dai sistemi a *paginazione*;
- Un approccio una volta diffuso e oggi caduto in disuso (in particolare, introdotto nell'Intel 286 e sostanzialmente rimosso a partire nell'architettura x86_64 di AMD) è quello della memoria **segmentata**. In questo caso ogni segmento viene gestito a livello hardware, per cui gli indirizzi diventano composti da due interi: il *segmento* e l'*offset* all'interno del segmento. Chiamiamo sistemi che usano questo approccio a *segmentazione*.

In quanto a pro e contro di questi approcci, abbiamo che:

- La memoria **unica** elimina i problemi di frammentazione *esterna* (ad ogni processo si alloca la memoria di cui necessita), ma introduce problemi di frammentazione *interna* (visto che solitamente la memoria si richiede in blocchi, potrebbe essere un problema riempire tali blocchi);

- La memoria **segmentata**, di contro, elimina i problemi di frammentazione **interna** (ogni segmento viene richiesto esattamente della dimensione necessaria), ma introduce problemi di frammentazione **esterna** (bisogna capire come usare lo spazio disponibile per allocare i segmenti).

  Un'approccio da usare in questi casi è quello del *compattamento* (o **deframmentazione**): periodicamente, si può analizzare la memoria in modo da spostare in un unico blocco contiguo i segmenti, in modo da massimizzare lo spazio disponibile per le immagini di nuovi processi.

1.4.2 Memoria contigua/non contigua

Vediamo un'altra distinzione ortogonale sulla gestione della memoria, in particolare relativa all'allocazione della memoria *fisica*:

- L'approccio più semplice che possiamo immaginare è quello di allocare la memoria in maniera **contigua**, cioè assicurare che ogni processo ottenga, nel suo spazio virtuale, un blocco contiguo (e opportunamente spostato di un certo offset) in memoria fisica;

- Risulta però molto più comodo concedere l'allocazione **non contigua** della memoria fisica, agendo sulla funzione di traduzione dell'MMU. Questo permette di ridurre a zero la frammentazione esterna, in quanto non c'è mai la necessità di mantenere, in primo luogo, separate e contigue le regioni allocate ai processi.

  Questo, però, introduce spesso un quanto minimo di spazio che possiamo allocare (come avevamo già introdotto), ed è l'approccio usato nella *paginazione*. Vediamo che la paginazione ha un overhead non indifferente: la realizzazione di una funzione di traduzione indirizzi che permetta l'allocazione flat e non contigua in memoria fisica richiede infatti tabelle anche consistenti in memoria, che possono essere allocate efficientemente solo usando schemi di allocazione particolari.

1.4.3 Dimensioni della memoria

Infine, vorremo distinguere sulla **dimensione** della memoria virtuale che vogliamo fornire al processo rispetto alla memoria fisica disponibile al sistema:

- Se la memoria virtuale è **minore o uguale** alla memoria fisica, saremo in condizioni di *caricamento unico*: potremo caricare l'intera immagine di processo in memoria fisica ed andare avanti;

- Altrimenti, se la memoria virtuale è **maggiore** della memoria fisica, dovremmo implementare meccanismi di *paginazione su domanda*, cioè allocare e deallocare regioni di memoria al processo su base dinamica, cioè quando questo le richiede.

1.4.4 Riassunto sulla gestione della memoria

Riassumiamo quindi i tipi di approcci possibili alla gestione della memoria, sulla base delle caratteristiche ortogonali viste finora:

- lista carattersistiche ortogonali

Vediamo quindi gli approcci possibili sulla base di tali caratteristiche: albero approcci gestione della memoria