Dai sistemi concorrenti a quelli distribuiti:
il problema della mutua esclusione

Roberto Baldoni

Quando si hanno processi concorrenti che accedono ad un risorsa condivisa nasce il bisogno di sincronizzarli in modo tale che tale risorsa sia assegnata ad un processo alla volta. Questo problema va sotto il nome di mutua esclusione. Dal punto di vista astratto il problema puo' essere formulato come segue. Ci sono N processi ognuno dei quali ripete la seguente sequenza di passi di programma

<non in sezione critica>
<trying protocol>
sezione critica
<exit protocol>
<non in sezione critica>

Una volta uscito dall'exit protocol il processo puo' rientrare infinite volte nel trying protocol. Questo problema e' stato definito per la prima volta da Dijkstra nel 1965 il quale fornisce anche una soluzione (istanzia cioé il trying protocol e l'exit protocol) in un modello di sistema che noi chiameremo concorrente. In un sistema concorrente esiste uno "scheduler" centralizzato che permette ad un solo processo alla volta di entrare in esecuzione e quindi di evolvere secondo il suo codice. Di fatto quindi lo scheduler esegue una linearizzazione di tutte le istruzioni elementari effettuate dai vari processi. La linearizzazione creata dalla singola esecuzione dell'algoritmo è chiamata "schedule". Il modello di Dijkstra si basa sulle seguenti assunzioni:

(D1) i processi comunicano leggendo e scrivendo variabili condivise

(D2) <u>la lettura e la scrittura di una variabile è una azione atomica</u>

Nessuna assunzione viene fatta sulla velocità dei processi e sul tempo che i processi ci mettono ad eseguire una singola azione atomica. Questa e' la tipica situazione di più processi che condividono una parte della stessa memoria.

Le proprietà di correttezza del problema di mutua esclusione da rispettare sono le seguenti (formulazione originale di Dijstra 1965):

a) Mutua esclusione (ME): due processi non possono essere nelle loro sezioni critiche contemporaneamente

b) No deadlock (ND): in una esecuzione di N processi dove un processo rimane bloccato nella sua trying section, ci sono uno o più processi che riescono ad entrare ed a uscire dalla loro sezione critica

E' anche interessante aggiungere altre proprietà al problema:

No Starvation (NS) nessun processo può rimanere bloccato per sempre nella sua trying section.

Da notare che NS ⇒ ND.

Algoritmo di Dijkstra (1965)

Quando il processo i vuole entrare in sezione critica prova prima ad impostare k al suo id i, passando quello che si chiama il ciclo di sentinella. Il valore di k ci dice chi e' l'ultimo processo pj che e' uscito dal ciclo della sentinella. Da notare che ciò non significa necessariamente che pj è l'unico processo che supera il ciclo della sentinella.

Esempio: Consideriamo infatti un sistema di 3 processi, p1, p2 e p3. Tutti e tre i processi iniziano ad eseguire la loro trying section (k e' inizialmente e' settato ad un valore diverso da 1,2,o3). P1 entra nel ciclo della sentinella (esegue il test sull' if) e poi viene interrotto dallo scheduler che mette in esecuzione p2. Anche p2 entra nel ciclo e viene bloccato dallo scheduler. Infine lo scheduler attiva p3 che esegue con successo tutto il ciclo della sentinella (settando il valore di k a 3). A questo punto lo scheduler sveglia prima p1 e poi p2, quindi quando tutti e tre i processi sono usciti dal ciclo della sentinella k ha assunto il valore 2.

Nel secondo ciclo (for) si controlla se ci sono stati passaggi concorrenti attraverso il primo ciclo ed in questo caso si deve eseguire una ulteriore scrematura per permettere ad un solo processo di entrare in sezione critica tra quelli passati per il ciclo della sentinella.

Esempio: riconsideriamo l'esempio precedente. Diversi casi possono accadere una volta p1, p2 e p3 superano il ciclo della sentinella. Analizziamo i due casi estremi particolarmente significativi

Caso 1) Se grazie alle attivazioni dello scheduler un processo, per esempio p1, riesce ad eseguire con successo il ciclo for (cio' significa che p2 e p3 sono stati bloccati dallo scheduler prima di eseguire l'istruzione 6), allora entrerà nelle sua CS indipendentemente dal fatto che k è stato settato da p2.

Caso 2) Se tutti e tre i processi non riescono a completare il ciclo for, ritorneranno alla linea 3 (goto statement). A questo punto mentre p1 e p3 si bloccheranno nel ciclo della sentinella, p2

sarà l'unico processo in grado di superarlo e quindi di portare con successo l'ingresso in sezione critica poiché p2 sarà l'unico processo ad eseguire il ciclo for.

Questo algoritmo soddisfa ME, ND ma non NS.

```
Shared variables
        x[1,..n]: array of Boolean, initially all false
        y[1,..n]: array of Boolean, initially all false
        k: integer in range 1,..N, initially any value in its range
Local variables
        j:  integer in range 1,..N
repeat
    1       NCS
    2       y[i]:= true                     % inizio trying protocol %
    3       x[i]:= false
    4       while k≠i do                    % ciclo della sentinella %
    5           if not y[k] then k:=i
    6       x[i]:= true
    7       for j:= 1 to n do
    8           if i≠j and x[j] then goto 3  % fine trying protocol %
    9       CS
    10      y[i]:=x[i]:= false;             %  exit protocol %
forever
```

Algoritmo di Dijkstra (1965)

Prova di ME. Per contraddizione supponiamo due processi i e j in sezione critica. Poiché i è in sezione critica allora i trovò x[j]=false durante il test di linea 8. Questo implica che pi ha eseguito la linea 8 prima che pj eseguisse la linea 6, pertanto i.8 → j.6. Chiaramente i.6 → i.8 quindi si ha i.6 → j.6. Scambiando i e j e seguendo lo stesso ragionamento otteniamo j.6 → i.6. pertanto i.6 → i.6, ovvero una contraddizione. €

Prova di ND. Supponiamo per contraddizione che esiste un deadlock che coinvolge un insieme di processi D (ovvero tutti i processi in D sono bloccati nella trying e nessuno può entrare nella

propria CS). Per tutti i processi i’ in D, y[i’]=true. Prendiamo il processo i che fa l’ultima assegnazione della variabile k (linea 5). Ovvero dopo questa assegnazione k=i per sempre (assunzione di deadlock). *Questo processo i necessariamente appartiene a D* altrimenti qualche altro processo j in D troverebbe y[i]=false e quindi porrebbe a j il valore della variabile k a linea 5. A questo punto ogni processo i’ in D\{i} avrà prima o poi (eventually) x[i’]=false, di conseguenza questi processi si bloccano nel while (linee 4-5). Inoltre la variabile x[i’] sarà prima o poi falsa anche per tutti quei processi che non appartengono a D e che sono in NCS. Consideriamo ora il processo i. Questo processo salta il ciclo while (linea 4) poiché k=i ed entrerà in sezione critica poiché tutti i valori x[i’] dei processi diversi da i sono falsi. Ma allora il processo i non può appartenere a D, contraddizione all’ipotesi che i deve necessariamente appartenere a D. €

Domanda: Un processo che setta con successo la variabile k quante volte può eseguire l’istruzione “goto 3” prima di entrare in sezione critica?
Domanda: Perché NS non è soddisfatta? Fornire un particolare scenario in cui si viola NS.

Il primo algoritmo derivato da quello di Dijkstra e starvation-free è dovuto a Knut (1966).

L’algoritmo del panettiere di Lamport (1975)

Nell’algoritmo di Dijkstra si assume che le read e le write sono atomiche ovvero o si esegue una o l’altra. Inoltre poiché esiste una variabile che viene letta e scritta da tutti (variabile k), questo imporrebbe che processi accedano alla stessa memoria fisica. Se si ragiona in un sistema con singola CPU (a livello harware) a memoria unica questo modello è assolutamente realistico. Una CPU o organizza un accesso in memoria in scrittura o in lettura. Se pensiamo ad un livello di astrazione più alto per esempio un sistema distribuito classico basato su reti di calcolatori con variabili replicate o ad un sistema multiprocessore in cui i processori hanno accesso ad una memoria a banchi che ammette accessi in lettura e scrittura simultanei, questa assunzione comincia ad essere un po’ troppo forte. Potremmo infatti avere copie di una stessa variabile su diverse macchine o accessi concorrenti in lettura ed in scrittura ad una variabile condivisa o file system che permettono ad utenti di leggere e scrivere file contemporaneamente. Questo potrebbe portare una operazione di write e di read a prendere un tempo considerevole per la sua esecuzione su diverse macchine e ad essere concorrenti. Dal punto di vista teorico è oltremodo interessante chiedersi: possiamo risolvere il problema della mutua escusione senza basarci su primitive hardware sottostanti che di fatto eseguono una mutua esclusione?

Oltre a D1, le assunzioni su cui si basa Lamport sono :

(L1) la lettura e la scrittura di una variabile non è una azione atomica. Uno scrittore potrebbe scivere mentre un lettore sta leggendo e nessuno (lettore o scrittore) viene notificato di tale interferenza.

(L2) Ogni variabile condivisa è di proprietà di un processo. Questo processo è l'unico che può scrivere tutti gli altri possono solo leggere (differentemente dalla variabile k dell'algoritmo di Dijkstra).

(L3) Nessun processo può emettere due scritture concorrentemente

(L4) Le velocità di esecuzione dei processi sono non correlate. In un tempo infinito ogni processo esegue infiniti step elementari mentre in un tempo finito esegue un numero finito di passi.

In questo algoritmo la trying section è divisa in due parti: la *doorway* (da linea 2 a linea 4) e il *bakery* (dal linea 5 a linea 8). Quando un processo entra nella doorway lo segnala agli altri attraverso la variabile choosing. Nella doorway il processo i legge tutti gli ultimi numeri usati dagli altri processi durante la loro ultima richiesta di accesso e definisce il numero di sequenza della sua corrente richiesta (linea 3). Le cose vanno come se un cliente entra in una panetteria portando da casa un biglietto eliminacoda in bianco. Il cliente e' in grado di leggere il numero di attesa di ogni cliente. A questo punto scrive nel proprio biglietto un valore più grande tra quelli letti. Chiaramente si deve avere ben presente che nella doorway possono esserci più processi contemporaneamente.

Uscito dalla doorway il processo i si avvicina al banco del panettiere (bakery section). A questo punto deve assicurasi che tra i processi che stanno in attesa lui è il prossimo a dover entrare nella sezione critica. Il ciclo **while** serve proprio a questo. Il processo i cicla fino a quando non e' sicuro che ogni altro processo j o (a) non è nella doorway o (b) ha un numero di coda maggiore del suo o lo ha uguale (caso in cui cui i e j hanno eseguito statement 2 concorrentemente) e j > i.

A questo punto il processo i accede alla sezione critica. Uscendo dalla sezione critica il processo i cancella dal suo biglietto di coda il numero usato scrivendo zero.

ME deriva dalla seguente proprietà: *"se un processo i è nella doorway ed un processo j è nella bakery section allora {num[j], j } < {num[i],i}"*. Quindi se due processi i e j sono nella CS contemporaneamente allora per la predecente proprietà otteniamo {num[i],i} < {num[j],j} e {num[j], j } < {num[i],i} chiaramente un assurdo. La prova formale di tale proprietà è rimandata alla parte di corso dedicata allo studio dei registri. NS è garantita dal fatto che nessun processo attende per sempre poiché prima o poi avrà il numero di attesa più piccolo.

Domanda: indicare il range della variabile num[i] nell'algoritmo del panettiere

L'algoritmo del panettiere gode inoltre della interessante proprietà seguente:

FCFS(first-Come-First-Served): Se il processo i entra nella sezione bakery prima che j entra nella doorway allora i entrerà in sezione critica prima di j.

Domanda: provare formalmente che l'algoritmo del panettiere assicura FCFS.

Domanda: perchè non vale la seguente (più intuitiva) nozione FCFS:

if il processo i entra nella doorway prima di j allora i entrerà in sezione critica prima di j

---

```
Shared variable
        num[1,..n]: array of integer, initially all 0
        choosing[1,..n]: array of Boolean, initially all false
                                              %process i owns num[i] and choosing[i]%
Local variable
        j:  integer in range 1,..N
repeat
    1       NCS
    2       choosing[i]:= true
    3       num[i]:= 1+ max {num[j] : n≥ j ≥1}                        %DOORWAY%
    4       choosing[i]:= false
    5       for j:= 1 to n do begin
    6           while choosing[j]  do skip
    7           while num[j] ≠0 and  {num[j], j } < {num[i],i}  do skip          %BAKERY%
    8       end
    9       CS
    10      num[i]:=0;
forever
```

L'algoritmo del Panettiere (1979)

A questo punto nascono due problemi:

1. possiamo implementare questo algoritmo in un ambiente completamente distribuito dove cioè un processo non può direttamente leggere le variabili di un altro processo?
2. poichè le azioni di lettura/scrittura non sono atomiche e quindi possono sovrapporsi temporalmente, che valore ritornerà questa lettura?

Nella parte successiva affronteremo il primo problema mentre il secondo verrà trattato più avanti nel corso.

L'algoritmo del panettiere in ambiente distribuito Client/Server

In questo caso i processi diventano separati spazialmente (ogni processo per esempio è in esecuzione su un elaboratore distinto) e per leggere il valore di una variabile un processo deve esplicitamente inviare un messaggio ed attendere una risposta che conterrà questo valore.

```
Local variable
        j:  integer in range 1,..N; num:  integer, initially all 0;
        choosing:  Boolean, initially  false;
repeat
    1       NCS
    2       choosing:= true                         %inizio doorway%    %inizio trying%
    3       for j≠i do
    4           send num  to pj
    5           receive reply(v) from pj
    6           num:=max(v,num)
    7         num:= num+1
    8       choosing:= false                        %fine doorway%
    9       for j:= 1 to n do begin                 %inizio bakery%
    10          repeat
    11                  send choosing  to pj
    12                  receive reply(v) from pj
    13          untill  v
    14          repeat
    15                  send num to pj
    16                  receive reply(v) from pj
    17          untill v=0 or {num,i}>{v, j }
    18      end                                     %fine bakery%       %fine trying%
    19      CS
    20      num:=0;                                                     %exit protocol%
forever
Upon the arrival of a num message from process j
```

send reply(num) **to** process j

Upon the arrival of a choosing message from process j

send reply(choosing) **to** process j

Algoritmo del panettiere in un sistema distribuito non cooperante

Di fatto ogni processo si comporta da server rispetto alle variabili num e choosing di sua proprietà e risponde alle richieste di lettura di client (altri processi).

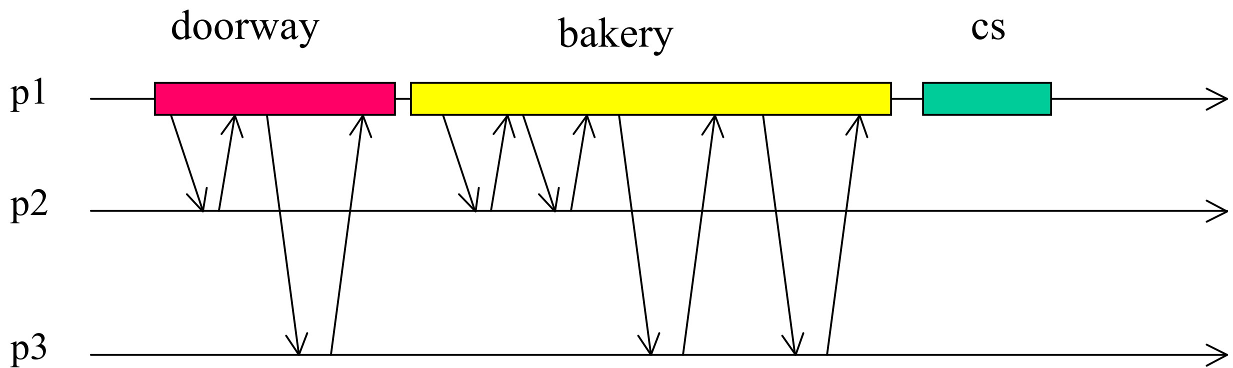
Oltre alle assunzione L1-L4, in questo ambiente si aggiungono le seguenti assunzioni:

(A1) i processi comunicano leggendo e scrivendo variabli attraverso scambio di messaggi. Il ritardo di trasmissione di un messaggio è impredicibile ma finito;

(A2) I canali di comunicazione sono affidabili. Un messaggio inviato viene ricevuto correttamente dal giusto destinatario. Inoltre non ci sono duplicazioni o messaggi spuri (ricevuti ma mai trasmessi)

Per l'assunzione A1 cambiano radicalmente le operazioni alle linee 3, 6 e 7 dell'algoritmo presentato nella precedente sezione. In questo caso un messaggio deve essere inviato ai vari processi per avere il valore della variabile corrispondente. I quali rispondono inviando il valore della variabile. Quindi il codice diventa quello riportato nella pagina seguente. Il codice della doorway va da riga 2 a riga 8 mentre quello del bakery da linea 9 a linea 20. Da notare come l'esistenza di messaggi implica l'inserimento di tread concorrenti al codice di mutua escusione in grado di rispondere alle richieste esterne. L'esecuzione di un tale algoritmo genera un message pattern simile a quello generato mostrato nella figura successiva.

Da notare che in questo algoritmo (come negli algoritmi precedenti) i processi non cooperano. Nel senso che un processo recupera i valori che gli necessitano e successivamente definisce il proprio numero di attesa. Gli altri processi cioè non hanno alcun ruolo attivo in merito se non quello di inviare un valore. Questa può essere una tecnica interessante nel caso in cui la lettura del valore non costi molto (per esempio in un sistema distribuito ad accoppiamento stretto come un multicomputer dove ogni processore è dotato di memoria privata ed i processori e le memorie sono connessi da un bus comune). Nel caso in cui i processi sono dispersi in un area locale o peggio geografica le interazioni client/server che servono per recuperare i valori rallentano notevolmente le performance dell'algoritmo.



Message pattern dell'algoritmo del bakery in ambiente distribuito non-cooperante

L'algoritmo del panettiere in ambiente distribuito peer-to-peer

Un approccio più adatto a questo tipo di ambiente è quello in cui i processi cooperano in modo esplicito per fare accedere ogni singolo processo alla sezione critica senza l'uso di un coordinatore centrale (peer-to-peer). Lo scopo di questa cooperazione è di avere dei message pattern più snelli in termini di messaggi scambiati per accesso alla sezione critica. La cooperazione tra i processi per esempio può ribaltare lo schema di scrittura del biglietto eliminacode. Si pensi ad esempio ad un cliente che entri dentro il negozio del panettiere con un numero già in tasca ed aspetti un invito esplicito da parte di tutti i clienti che individuano in lui il prossimo cliente che deve essere servito.

A questo punto il problema diventa come assicurare che il numero che porta da casa il cliente rispetta un ordine totale imposto dall'accesso in mutua esclusione della propria sezione critica. Questo può essere realizzato pensando alla variabile num come unica nel sistema di cui ogni processo ne possiede una copia ed ogni processo la aggiorna e comunica questo aggiornamento in modo che anche gli altri possano aggiornarla. Mentre nel bakery originale colui che che sta entrando nella doorway vuole leggere i valori dei biglietti eliminacode degli altri processi per scrivere il suo numero, nei sistemi distribuiti cooperanti un processo quando entra nella doorway (linea 1-3) comunica agli altri il suo numero. Chiaramente per fare ciò e rispettare un'ordinamento totale quando questo processo riceve messaggi di altri che desiderano entrare nella doorway deve mantenere in memoria il massimo numero visto. A questo punto la condizione di accesso non è computata in modo locale come nel caso precedente una volta recuperate tutte le informazioni (num e choosing) dagli altri processi, tale condizione (linea 4) è dovuta ad un messaggio di consapevole riscontro di ciascuno degli altri processi (messaggi di reply). Questi processi in piena autonomia quando vedono la richiesta di un certo nodo i essere in testa all'ordine totale (ovvero la prossima richiesta che deve essere servita al banco del panettiere) inviano un REPLY al processo i.

```
Local variable
        #replies:  integer initially set to zero
        state: in set {requesting, CS, NCS} initially set to NCS
        Q: queue of pending request {T,i} in itially set to empty
        Last_req: integer ; initially set to maxint
        Num: integer initially set to zero
begin
    1       state:= requesting
    2       num:=num+1; last_req:=num;
    3       send REQUEST(last_req) to p1..pn
    4       wait until #replies=(n-1)
    5       state:= CS
    6       CS
    7       send REPLY to any request in Q; Q:=∅; state:=NCS;  #replies:=0; lastreq:=maxint;
end

Upon the receipt of REQUEST(t) from process j
    8       num:=max(t,num)
    9       if state=CS or (state=requesting and {last_req, i } < {t,j})
    10      then insert.Q({t, j})
    11      else   send REPLY to Pj

Upon the receipt of REPLY from process j
    12. #replies++;
```

Algoritmo di Ricart-Agrawala

Quando il processo i ha collezionato un REPLY da ogni altro processo può entrare in CS. Per evitare violazione di ME un processo che è (1) in sezione critica o (2) in attesa ma con un numero di attesa inferiore a quello della richiesta del processo i , ritarda l'invio del messaggio di reply fino alla sua uscita dalla sezione critica. Queste richieste ritardate vengono inserite in una coda Q. Questo algoritmo è stato scritto da Ricart-Agrawala nel 1981.

Assunzione. Linea 2 e' eseguita atomicamente.

Prova di ME. (Per contradizione). Supponiamo i e j in CS contemporaneamente allora i ha inviato un REPLY message a j e viceversa. Tre casi sono possibili analizzando il message pattern associato all'algoritmo.

Caso 1. *il processo i spedisce il REPLY prima di scegliere il proprio num alla linea 2 (timestamp della richiesta).* Quindi la richiesta di i avrà necessariamente un timestamp più alto di quello della richiesta di j. Poichè i sulla ricezione della richiesta di j avrà eseguito linea 11 e prima di spedire la richiesta i eseguirà linea 2. Una volta che la richiesta di i arriva a j quest'ultimo metterà la richiesta in coda poichè o e' già nella propria CS oppure è nello stato requesting ma il timestamp associato alla sua richiesta è minore di quello di i. Di conseguenza i e j non possono essere entrambi in CS

Caso 2. identico al caso 1 invertendo i e j.

Caso 3. *i e j spediscono un REPLY all'altro dopo aver scelto il timestamp per la propria richiesta*. In questo caso la regola di ordinamento delle richieste è unica e deterministica in ogni processo. Quindi quando viene ricevuto il messaggio di richiesta, uno dei due spedisce un REPLY e l'atro metterà la richiesta in coda (test di linea 8). Di conseguenza non possono essere entrambi in CS. €

Prova di NS. Assumiamo che esista un processo i che attende indefinitivamente l'accesso in sezione critica dopo aver inviato la richiesta con timestamp num_i. Poichè ogni messaggio arriva a destinazione in un tempo finito, ciò significa che esiste un set non vuoto di processi S che non invia il messaggio di REPLY a i. Sia j un processo appartenente a S. Questo accade solo se j è in sezione critica o è in attesa e la sua richiesta ha priorità rispetto a j ovvero {num_j, j}<{num_i,i}. Nel primo caso in un tempo finito j esce dalla sezione critica e invia un REPLY a i sbloccandolo e contraddicendo l'assunzione iniziale. Nel secondo caso esiste un processo k appartenente a S\{j} che blocca j. Usando gli stessi argomenti, si definisce una sequenza di attesa di lunghezza pari alla cardinalità di S (max n processi) <i,j,k,....u> tale che un processo blocca il suo precedente nella sequenza e la richiesta del successivo ha priorità sul precedente quindi per transitività {num_u, u}<{num_i,i}. In questa sequenza l'ultimo processo u è bloccato necessariamente da un processo che lo precede nella sequenza. Sia i questo processo, quindi {num_i, i}<{num_u,u} da ciò segue l'assurdo {num_i, i}<{num_i, i}. €

<u>Domanda</u>. Consideriamo un sistema in cui al più k processi possono essere nelle loro sezioni critiche contemporaneamente (k-mutex). Come cambia il precedente algoritmo per migliorare il grado di concorrenza?