26 Maggio 2023

ARCHITETTURE DI CALCOLO LEZIONE 20

Sincronizzazione dei processi

L'algoritmo di Peterson risolve il problema della sincronizzazione per soli due processi; la sua generalizzazione è l'algoritmo di Lamport (o del fornaio). Esso si fonda sul meccanismo dell'elimina-code, ovvero, prima di entrare nella sezione critica, il processo riceve un numero ed il possessore del numero più piccolo, entra nella sezione critica.
Lo schema di numerazione non assicura che ogni processo riceva un numero diverso, ma garantisce sempre la generazione di numeri in ordine crescente; ad esempio: 1,2,3,3,3,3,4,5...
Se i processi Pb e Pd ricevono lo stesso numero (a=c), si verifica il valore del loro indice (b, d), e, se b < d, allora Pb è servito prima; altrimenti sarà Pd ad essere servito per primo.

$$(a,b) < (c,d) \text{ se } a < c \text{ oppure } a == c \text{ and } b < d$$

Nell'algoritmo di Lamport i valori condivisi sono due:
- Un array di valori booleani di dimensioni n→choosing = [False]*n (inizializzata a False)
- Un vettore di numeri di dimensione n→number = [0]*n (inizializzata a 0)

Se choosing = True, ciò vuol dire che un processo sta per ricevere il numero, mentre ni è il numero assegnato al processo i-esimo
L'algoritmo si divide in:

```
while True:
   choosing[i] = True  # sta prelevando il numero
   number[i] = max(number[0], number[1],…, number [n-1]) + 1
   choosing[i] = False
   for j in range(n)
      while choosing[j]:
        pass # attende se qualche Pj sta prelevando il numero
      while number[j] != 0 and (number[j],j) < (number[i],i):
        pass
   [sezione critica]
   number[i] = 0
   [sezione non critica]
```

•Entry section: si ottiene l'accesso alla sezione critica
•Sezione critica: vengono modificate le variabili
•Exit section: termina l'accesso alla sezione critica
•Sezione non critica: vengono compiute operazioni non critiche, ovvero si opera su variabili non condivise

Spiegazione codice

- Settando choosing[i]=True, un processo dichiara di voler entrare nella sezione critica
- La variabile number[i] è pari al massimo numero tra quelli già assegnati +1
- Si setta chooosing[i]=False al termine dell'assegnazione del numero
- Vi sono, poi, due busy waiting; il processo deve controllare n volte che non vi sia alcun processo che stia ricevendo il numero (1° while) e che il j-esimo processo non abbia un numero uguale a 0 o un numero minore del proprio (2° while). È importante effettuare il controllo del 1° while per evitare di "passare avanti" ad un processo con un numero

minore (= priorità maggiore), andato in waiting a causa del termine del quanto di tempo o altra motivazione.
- Settando number[i]=0, un processo comunica di essere uscito dalla sezione critica

```
while True:
        acquisisce il lock
              sezione critica
        restituisce il lock
              sezione non critica
```

Un'altra soluzione per risolver il problema dell'accesso alla sezione critica è l'uso del lucchetto (lock). Con questo meccanismo, prima di entra nella sezione critica, bisogna acquisire il lock, e, dopo esserne usciti, bisogna rilasciare il lock.

Istruzione atomica

Un nuovo concetto da introdurre è quello dell'istruzione atomica; un'istruzione è detta atomica se completa la propria esecuzione senza interruzioni.
Per esempio:
se le istruzioni "counter+=1" e "counter -=1", nel caso del produttore/consumatore, fossero state atomiche negli algoritmi precedentemente studiati, non si sarebbe potuto verificare un interleaving delle relative istruzioni di basso livello.
Le variabili atomiche sono comunemente utilizzate nei SO e nelle applicazioni concorrenti, ma il loro uso è spesso limitato ad aggiornamenti di singoli dati condivisi, come contatori e generatori random.
Esempio:
Nel problema del produttore-consumatore con buffer limitato, se il contatore è una variabile atomica non vi sarà race condition (modifica concorrente di variabile) all'atto dell'incremento/decremento.
Una race condition, invece, si può verificare nel controllo del ciclo while; se due consumatori sono in busy waiting e un produttore produce un elemento, entrambi potrebbero venire sbloccati e procedere al consumo di quell'unico elemento. Serve, allora, un meccanismo per bloccare "atomicamente".
Sono stati, così, introdotti strumenti software (API), come i "lock mutex" (lucchetti di mutua esclusione), che permettono di "congelare" le istruzioni in blocco, e non solamente singole variabili.

```
acquire() {
   while (!available);
   /* busy wait */
   available = false;
   }
```

```
release() {
   available = true;
}
```

I lock mutex vengono implementati attraverso le funzioni "acquire()" e "release()", entrambe atomiche, e con la variabile booleana "available", che esprime la disponibilità del lock.
Anche questa soluzione, però, prevede una busy waiting (attesa attiva), in cui il processo, pur non compiendo azioni utili al completamento del codice, utilizza la CPU; in questo caso, i processi sono detti "spinlock".
Per risolvere la problematica degli spinlock, si può utilizzare la tecnica della sospensione, cosicché il processo, invece di rimanere in un continuo stato di running, faccia il passaggio running→ wainting→running, rendendo la CPU più efficiente. Per implementare questo sistema, si usano i semafori, introdotti da Edsger Dijkstra (inventore dell'algoritmo del

banchiere e vincitore del Turing Award nel 1972), che garantiscono la mutua esclusione e la sincronizzazione tra processi.
Il semaforo contiene una variabile intera (contatore) che può essere incrementata o decrementata esclusivamente mediante i metodi atomici acquire() e release(), chiamati anche P() [dall'olandese proberen=verificare] e V() [dall'olandese verhogen=incrementare] o wait() e signal().
Un thread (o processo) acquisisce un semaforo invocando il metodo acquire() e lo rilascia invocando il metodo release(). Quando il contatore assume un valore negativo o nullo, il semaforo si considera "rosso", ed il thread che invoca il metodo acquire viene sospeso. L'invocazione del metodo release da parte di un thread può risvegliare un altro thread precedentemente sospeso nel semaforo.
I semafori possono essere usati in caso di:

- **Busy waiting**
  Si crea un semaforo s (variabile intera);

```
wait(S) {
   while(S <= 0)
      ;//attesa attiva
   S--;
}
```

```
signal(S) {
   S++;
}
```

-Se s<=0, si entra in uno stato di waiting;
-Se s>0, si entra nella sezione critica e si decrementa s
-Se si vuole uscire dalla sezione critica, si incrementa s
La variabile s va inizializzata ad un intero positivo, che indicherà a quanti processi è concesso entrare nella zona critica

- **Mutua esclusione**

Semaforo s = 1

Codice di **T1**
...
s.acquire()
A
s.release()
...

Codice di **T2**
...
s.acquire()
B
s.release()
...

Per implementare la mutua esclusione con un semaforo s, s deve essere inizializzato a 1.
Es. Dati due thread T1e T2, i quali eseguono rispettivamente un blocco di istruzioni A e B, si vuole garantire la mutua esclusione dell'esecuzione delle istruzioni A e B (se T1 sta eseguendo A, allora T2 non può eseguire B, e viceversa), usando un semaforo (s), come nella grafica a sinistra.

Cosa avviene?
Supponendo che il thread T1 arrivi per primo, esso esegue l'operazione acquire(), portando s a 0. Se dovesse arrivare T2 prima che T1 esegua release(), T2 non potrebbe entrare nella sezione critica poiché il semaforo è "rosso" (s=0). T1, quindi, esegue A e poi la funzione release (), incrementando s di 1; ora il semaforo è di nuovo "verde" e T2 può entrare nella sezione critica.

- **Sincronizzazione**

Semaforo s = 0

Codice di **T1**
...
...
A
s.release()
...

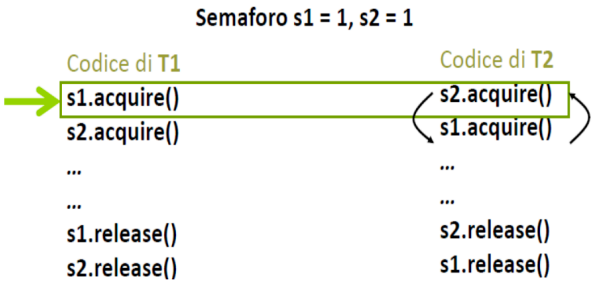
Codice di **T2**
...
s.acquire();
B
...
...

Es. Dati due thread T1e T2, i quali eseguono rispettivamente un blocco di istruzioni A e B, si vuole garantire l'ordine di esecuzione (sincronizzazione), prima di A e poi di B, usando un semaforo (s), come nella grafica a destra.
Cosa avviene?

Supponendo che T2 arrivi per primo, questo non potrà eseguire la funzione acquire() poiché il semaforo è 0, al contrario del thread T1, che non ha alcuna funzione da eseguire prima del blocco di istruzioni A. Quindi, T1 esegue A e poi release(), incrementando s di 1; ora T2 può eseguire acquire() e, successivamente, B.

Deadlock

L'uso scorretto dei semafori può portare al deadlock, situazione in cui uno o più processi (o thread) sono indefinitamente in attesa di un evento che può essere causato soltanto da uno o altri processi, che a loro volta sono in attesa (loop infinito).
In questo esempio, il deadlock si presenta qualora i due processi arrivino contemporaneamente poiché, dopo aver eseguito la prima riga di codice, i thread devono fare l'acquire() del semaforo già occupato dal thread concorrente. Risulta, così, impossibile eseguire la release, creando uno stato di stallo.



Esempio. Vi sono due thread, T1 e T2 che condividono due semafori, S1 e S2. Il T1 fa, in ordine, l'acquisizione del semaforo S1 e l'acquisizione del semaforo S2, esegue poi una serie di istruzioni (...) ed infine rilascia S1 e S2; stessa cosa dall'altra parte per T2, però in ordine
inverso (prima S2 e poi S1). Questo codice può causare un deadlock.
La cosa interessante del deadlock è che se si scrive un programma del genere e lo si lancia, potrebbe funzionare per centinaia o migliaia di volte, senza andare in deadlock, semplicemente perché non viene a verificarsi per tutto questo tempo
che i due processi vadano contemporaneamente in esecuzione. La soluzione, in questo esempio, è invertire s2. acquire() e s1.acquire() nel codice di T2.
Nell'uso dei semafori dobbiamo ricordare che l'ordine in cui si acquisiscono e si lasciano è molto importante. Una delle cose più difficili per un programmatore è riuscire ad evitare il deadlock, tant'è che spesso il deadlock non viene prevenuto a priori, ma al suo verificarsi, si prova a trovare una soluzione.

Starvation

Un altro termine che si usa nel contesto della programmazione concorrente ai sistemi operativi è "starvation", ossia un blocking indefinito che avviene quando un processo rimane in attesa potenzialmente in modo illimitato poiché potrebbe non essere mai rimosso dalla coda del semaforo. Si verifica quando un processo (thread) non riesce ad ottenere un regolare accesso alle risorse condivise e per questo non progredisce.

La tecnica dell'Aging

La tecnica dell'aging (invecchiamento) è una possibile soluzione al problema della starvation che usa un algoritmo di scheduling basato su priorità:

- Un thread "invecchia" e diminuisce la sua priorità ogni volta che riesce ad acquisire una risorsa a discapito di altri thread.
- Quando la sua priorità raggiunge una certa soglia, il thread rinuncia ad acquisire la risorsa in favore di altri thread che l'hanno acquisita meno frequentemente.

Semafori binari

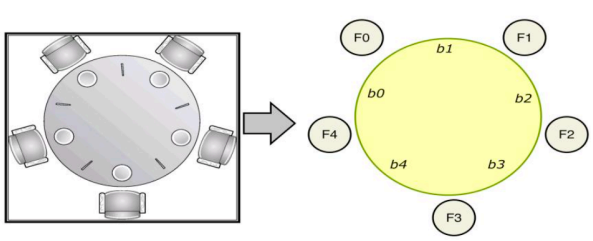
Il semaforo definito finora è chiamato semaforo contatore, in quanto il suo valore intero può spaziare su un dominio specificato. Sono detti, invece, semafori binari perché possono assumere solo due valori: 0 e 1.
I semafori binari sono sostanzialmente usati come lock nei mutex perché, mentre con un semaforo contatore si può permettere ad N processi di entrare in una parte del codice, con i semafori binari posso solo consentire o negare l'accesso.

Problema dei cinque filosofi

Ci sono cinque filosofi seduti ad un tavolo e ciascuno ha davanti a sé un piatto di riso (che supponiamo essere inesauribile). Tra un piatto ed un altro vi è una sola bacchetta, per un totale di soli 5 bastoncini, contro i 10 che sarebbero richiesti per permettere a tutti di mangiare contemporaneamente. Ciascun filosofo, per tutta la sua vita, fa solo tre cose:

- Pensare, quando è sazio;
- Mettersi in attesa di mangiare, quando ha fame;
- Mangiare.

Quando decide di mangiare, prima di poterlo fare, ha bisogno di prendere in mano due bacchette. Un filosofo può prendere solo le due bacchette che stanno rispettivamente alla sua destra e alla sua sinistra, una per volta, e solo se sono libere (non può sottrarre la risorsa "bacchetta" ad un altro filosofo che l'ha già acquisita). Finché non riesce a prendere le bacchette, il filosofo deve aspettare affamato. Quando, invece, appena dopo aver mangiato, è sazio, posa le bacchette al loro posto e si mette a pensare per un certo tempo.



Questo è un possibile scenario di starvation dei processi e, in alcuni casi, di deadlock: se tutti i filosofi vogliano mangiare nello stesso momento e prendono tutti la bacchetta alla propria destra, nessuno riesce a mangiare (deadlock).

Soluzione: si potrebbero "proteggere" le cinque bacchette (regolarne l'acquisizione), usando cinque semafori (binari: verde o rosso), inizializzandoli a 1.

Il codice che segue è scritto in Java e definisce un array di dimensione cinque contenente in ogni posizione un semaforo.

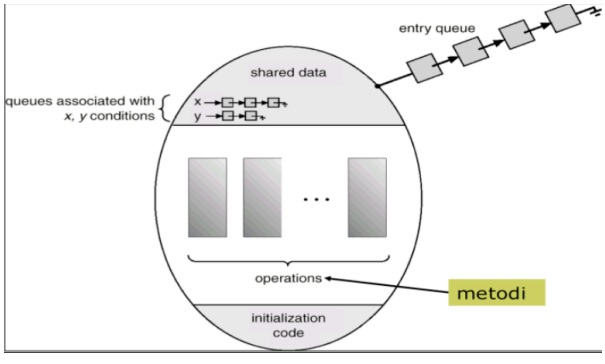
```
Semaphore bacchetta[ ] = new Semaphore [5]; //Inizialmente tutti i semafori valgono 1
Filosofo[i] eseguirà queste operazioni:
while (true) {
  bacchetta[i].acquire();
  bacchetta[(i+1) % 5]).acquire();
  …
  <mangia>
  …
  bacchetta[i].release();
  bacchetta[(i+1) % 5].release();
  …
  <pensa>
  …
}
```

Ogni filosofo per iniziare a mangiare deve acquisire la bacchetta in posizione (i) e la bacchetta in posizione (i+1), quindi per mangiare il filosofo 0 deve prendere la bacchetta 0 e la bacchetta 1 (questo metodo funziona in modo ciclico quindi dopo la bacchetta 4 c'è la bacchetta 0). Il problema di questo codice è che, se tutti quanti si trovano nello stesso momento a prendere la bacchetta dallo stesso lato, ad esempio da sinistra, poi quando vanno ad acquisire quella di destra, si bloccano perché la release si fa dopo.
Soluzione al problema dei cinque filosofi:
Tali situazioni di stallo possono essere evitate attraverso i seguenti espedienti:

- Un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (quest'operazione si deve eseguire in una sezione critica); questa soluzione si chiama l'allocazione completa.
- Si adotta una soluzione asimmetrica: un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra, invece un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra.

MONITOR

È un costrutto di alto livello che consente la condivisione sicura di dati tra più thread, inventato dai due ricercatori Hansen e Hoare negli anni '70. Per monitor s'intende un oggetto in grado di monitorare gli accessi.



•Può essere implementato come una classe che racchiude variabili ed esporta metodi che accedono a tali variabili in modo atomico.
•Le variabili del monitor possono essere utilizzate solo mediante i metodi di accesso e solo un thread per volta può usare questi metodi.
•Ad ogni istanza del monitor è associata una coda di thread in attesa di eseguire uno dei metodi di accesso.

- L'accesso in mutua esclusione alle funzioni, viene garantito attraverso un lock.
- Il lucchetto può essere posseduto da un solo thread alla volta
- Ogni metodo di accesso al monitor comincia con la chiusura del lucchetto (metodo lock()) e termina con l'apertura del lucchetto (metodo unlock()).

Un altro elemento presente nei monitor è il concetto di Condition (o coda interna), strumento con cui si può sospendere un processo all'interno del monitor, non perché sta accedendo ad una risorsa che è gia occupata ma perché lo si vuole sospendere per qualche motivo.
Questo oggetto che sembra a forma di uovo contiene una coda che prende il nome di *entry queue*, cioè la coda di accesso, su cui ci sono tutti i thread o processi in attesa di accedere al monitor, al quale si accede uno per volta. Poi ci sono i metodi, che devono essere eseguiti sempre uno per volta perché le variabili su cui lavorano sono condivise e non possono essere modificate contemporaneamente.
Poi abbiamo il codice di inizializzazione (sotto) ed i dati condivisi, ovvero variabili (sopra).
Le code associate alle conditions sono costituite da processi che, una volta entrati nel monitor, non trovano le condizioni necessarie per continuare nell'esecuzione, e vanno in uno stato di waiting.
Esempio: se un processo consumer entra nel monitor ma trova il buffer vuoto, questo va in waiting nelle code associate alle condition "buffer vuoto", finchè non si verifica l'arrivo di un producer che incrementa il buffer e, di conseguenza, risveglia il processo precedente dallo stato di waiting. La stessa situazione si verifica a parti invertite tra il processo consumer e producer, con l'unica differenza che la coda su cui il processo farà la waiting, sarà quella associata alla condizione "buffer pieno".
Esistono due tipi di code:

- la **coda di accesso** che garantisce il fatto che un solo processo alla volta possa utilizzare le funzioni presenti;
- la **coda di condition** che garantisce la possibilità dei processi non entrati di sospendersi in attesa di andare avanti.

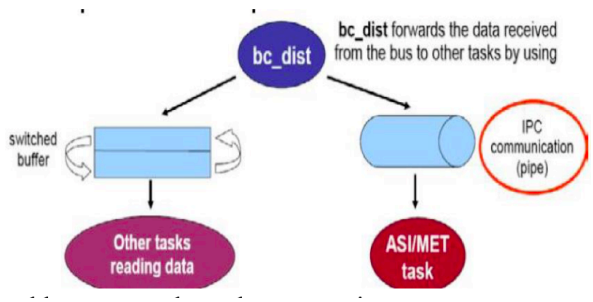
Poiché i monitor sono stati inventati da due ricercatori in modo indipendente, esistono due versioni alternative che presentano una semantica di compontamento differente in alcune situazioni.
Immaginammo di avere due processi P e Q, supponiamo che un processo P esegua x.signal(), ed esista un processo sospeso Q associato alla variabile condition x. Dopo la signal, che porta al "risveglio" di Q, occorre evitare che P e Q risultino contemporaneamente attivi all'interno del monitor:

- **Hansen**: Q attende che P lasci il monitor o si metta in attesa su un'altra variabile condition (questa è la soluzione implementata in Java).
- **Hoare**: P attende che Q lasci il monitor o si metta in attesa su un'altra variabile condition.

La soluzione di Hansen risulta essere più intuitiva ma potrebbe generare dei problemi. Infatti, consentendo al processo P di continuare, la condizione logica attesa da Q può non valere più nel momento in cui Q viene ripreso. Il problema per la soluzione di Hoare è che richiede più impegno, cioè richiede che si crei un'altra coda in cui vengono messi i processi segnalati che devono entrare con priorità maggiore rispetto a quelli che sono nell'entry.
Una soluzione di compromesso: P deve lasciare il monitor nel momento stesso dell'esecuzione dell'operazione signal, in quanto viene immediatamente ripreso il processo Q. Questo modello è meno potente di quello di Hoare, perchè un processo non può effettuare una signal più di una volta all'interno di una stessa procedura. Ad esempio, Java implementa i monitor usando la soluzione di Hansen (il thread risvegliato Q deve riacquisire il lock, concorrendo con gli altri thread che provano ad acquisire il lock).

Mars Pathfinder e inversione delle priorità

- La sonda spaziale della NASA Mars Pathfinder, nel luglio 1997, portò su Marte il robot Sojourner, destinato all'esplorazione del pianeta.
- Il Sojourner inviò alle stazioni di ascolto sulla Terra 550 fotografie e analizzò le proprietà chimiche di sedici siti in prossimità del lander.
- Tuttavia, poco dopo che ebbe iniziato il suo lavoro, cominciarono ad aver luogo numerosi reset del sistema, quindi si perdeva il tempo.
- Ciascun reset reinizializzava sia il software che l'hardware, inclusi gli strumenti preposti alla comunicazione.
- Il problema era causato da un processo ad alta priorità, che si chiamava bc_dist, che impiegava più tempo del dovuto a portare a termine il proprio compito.



-bc_dist attendeva infatti una risorsa condivisa (una pipe) utilizzata dal processo ASI/MET, a priorità più bassa, a sua volta prelazionato da processi con priorità intermedia. Il processo bc-dist rimaneva bloccato da processi con priorità minore.
-Quindi bc_dist andava in stallo, spesso per un tempo molto lungo, e il processo bc_sched, rilevando il problema, procedeva al reset per time-out.

<u>Cosa s'intende per inversione di priorità ?</u>
Una situazione in cui a causa di risorse condivise, una priorità più bassa finisce per prelevare su una a priorità maggiore.

- Il SO real-time on board (VxWorks) disponeva di una variabile globale per abilitare l'ereditarietà delle priorità, che fu modificata dagli opertori sulla Terra, ristabilendo la piena operatività del rover.