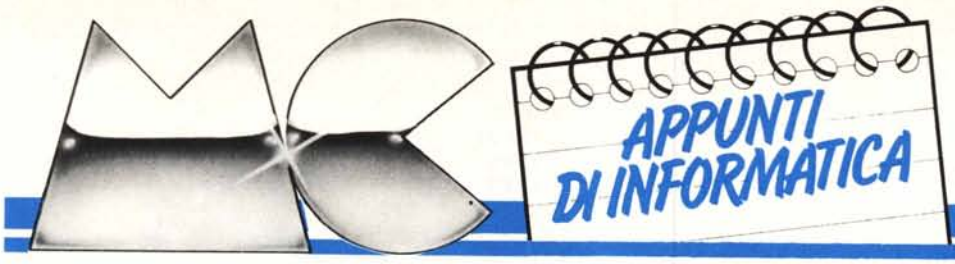


di Andrea de Prisco

Concorrenza, semafori, monitor

Dopo aver discusso lo scorso mese le principali modalità di multiprogrammazione di un computer, questo mese analizzeremo il problema della concorrenza di più processi che accedono a risorse condivise, sia questa un'unità di ingresso uscita, una struttura dati o semplicemente una variabile. Se non si prendono le dovute cautele, infatti, possono succedere veramente cose strane: procediamo con ordine...

Prologo

Per chi non ci ha seguito lo scorso mese, facciamo un attimino il punto della situazione, quantomeno circa la terminologia che useremo nel resto dell'articolo.

Innanzitutto un processo è, in parole povere, un «programma che gira» o per essere più precisi, un processo è quanto descritto dal programma mantenuto in memoria e che il processore «processa». Si introduce questa nuova definizione per indicare qualcosa di effettivamente dinamico, nonché capace di provocare il verificarsi di eventi. Di contro, un programma è semplicemente un insieme di istruzioni che descrivono qualcosa, quindi per sua natura è statico.

Detto questo, possiamo introdurre il concetto di stato (di un processo). Nei sistemi multiprogrammati infatti più programmi possono essere mantenuti in memoria, e con i meccanismi mostrati lo scorso mese possono essere eseguiti in parallelismo reale o simulato dal o dai processori di cui il calcolatore in considerazione dispone. A causa di alcuni eventi, abbiamo visto, un determinato processo può trovarsi in tre diversi stati: in stato di esecuzione, in stato di pronto o in stato di sospeso.

Il primo caso riguarda il processo effettivamente eseguito dal processore nell'istante che stiamo considerando. In stato di sospeso vanno invece quei processi che dovendo attendere passivamente un evento esterno (ad esempio l'arrivo di un dato dall'unità a dischi), liberano la CPU che così può dedicarsi a elaborare dell'altro. Infine in stato di pronto vanno quei processi che hanno ottenuto ciò che aspettavano e sono pronti per ripartire non appena la CPU si libera rilasciando il processo attualmente in esecuzione.

Le risorse condivise

Se i vari processi che devono essere eseguiti in parallelo sono completamente indipendenti l'uno dall'altro quanto esposto lo scorso mese non fa una grinza. Ciò però è davvero difficile che si verifichi: infatti già le sole periferiche di ingresso uscita rappresentano un punto di contatto (e naturalmente di potenziale collisione) per i vari processi. Immaginate che due processi cerchino di scrivere contemporaneamente qualcosa su disco. Dal canto suo, la periferica può esaudire solo una richiesta per volta: si rende necessario un meccanismo che in qualche modo arbitri tali accessi.

Se poi i processi da eseguire in parallelo cooperano usando strutture dati condivise l'affare si complica ulteriormente se non si prendono le opportune cautele. Cooperazione tutt'altro che rara: infatti lo scrivere oggi un sistema operativo di calcolatore, non corrisponde più a stendere un enorme programmone zeppo di routine che svolgono le funzioni più disparate. La tendenza attuale è quella di multiprogrammare un calcolatore già a livello di sistema operativo: esso stesso sarà una collezione di processi (evolventi in parallelo) che cooperano per svolgere le funzioni volute. Ad esempio avremo uno o più processi che controllano l'output su stampante o l'I/O del disco; tramite opportuni meccanismi (alcuni li vedremo subito) sarà poi possibile far comunicare più processi tra di loro per «intendersi» sul da farsi.

La cooperazione tra processi può avvenire in due distinti modi: ad ambiente locale o ad ambiente globale. Nel primo caso si instaura un vero e proprio traffico di messaggi: un processo spedisce un messaggio ad un altro eventualmente aspettando anche una risposta. Il nucleo del sistema operativo provvederà a dirigere tali «spedizioni» in modo, per così dire,

Figura 1

```
LDA   "X"
ADD    1
STA   "X"
```

Figura 2

```
SEI
LDA   "X"   }
ADD    1    }  Sezione Critica
STA   "X"   }
CLI
```

Figura 3

```
SEI
LOCK  <K>
LDA   "X"   }
ADD    1    }  Sezione Critica
STA   "X"   }
UNLOCK <K>
CLI
```

Figura 4

(a)

```
LOCK  <K>
10 IF K=1 THEN 10
20 K=1
30 RETURN
```

(b)

```
UNLOCK  <K>
10 K=0
20 RETURN
```

trasparente al livello dei processi. La cooperazione ad ambiente globale avviene invece tramite zone di memorie, più in generale strutture dati, condivise tra i processi: ad esempio, se il processo A deve comunicare qualcosa al processo B provvederà coi propri mezzi (li vedremo) a scrivere «il qualcosa» in una locazione di memoria, dove B si «recherà» per prelevarlo. È proprio in casi come questi che si parla di risorse condivise: nel nostro esempio, la risorsa è rappresentata dalla locazione di memoria usata per l'interazione.

Sezioni critiche

Detto questo, cominciamo a parlare dei problemi che insorgono quando degli oggetti (le risorse) sono manipolate da particolari agenti (i processi) che come abbiamo riportato sono capaci di provocare il verificarsi di eventi.

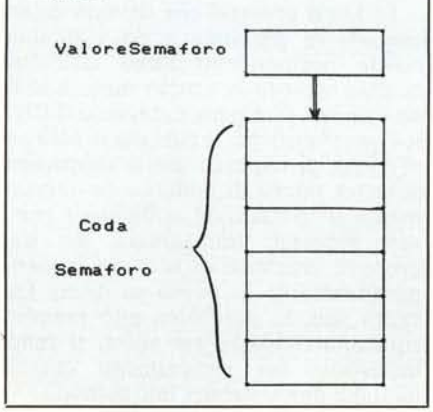
Eventi che possono benissimo essere alcune volte quantomeno indesiderati.

Come abbiamo visto lo scorso mese, nei sistemi a divisione di tempo (timesharing), la CPU veniva concessa ai singoli processi per un determinato periodo di tempo, scaduto il quale il controllo passava per un altro quanto di tempo ad un altro processo prelevato dalla lista «pronto». Scegliendo quanti di tempo sufficientemente piccoli era così possibile simulare un parallelismo anche su computer uniprocessor, o in generale su ogni processor di un computer multiprocessor. Lo scadere dei quanti di tempo, veniva segnalato da un interrupt proveniente da un orologio interno al calcolatore che svolgeva proprio tale funzione. Immaginiamo ora che due processi, P1 e P2, utilizzino una variabile comune, ad esempio X, con la quale contano qualcosa. Ciò in un generico linguaggio ad alto livello si esprime con un comando del tipo:

$$X = X + 1$$

presente in ciascuno dei due processi. Teniamo a sottolineare il fatto che la X è uno stesso oggetto visibile da due processi, non due variabili distinte con lo stesso nome. In altre parole se P1 pone $X=0$ e P2 esegue $X=X+1$, anche per P1 il valore di X sarà 1 (è stato P2 a incrementarlo). Focalizziamo la nostra attenzione su due incrementi effettuati uno da P1 e l'altro da P2. Se all'inizio X vale 0, dopo l'esecuzione di questi due comandi, ovviamente, X risulterà incrementata di due. Questo nella migliore delle ipotesi: infatti può succedere qualcosa di anormale. Vediamo perché. Del comando $X=X+1$ al processore arriverà una traduzione in linguaggio macchina sia che si tratti di interpretazione che di compilazione del programma di partenza. Ad esempio, la sequenza di operazioni corrispondente sarà tipo quella mostrata in figura 1: si pone il contenuto della variabile X nell'accumulatore, si somma 1 all'accumulatore, si scrive il contenuto dell'accumulatore nella variabile X.

Figura 5 Semaforo



Poniamo il caso in cui, non appena si trasferisce il contenuto di X nell'accumulatore, l'orologio mandi l'interrupt alla CPU essendo scaduto il quanto di tempo di P1. Parte così P2 e diciamo che prima del prossimo scadere di tempo esegua anch'esso un $X=X+1$, questa volta portandolo a termine. Riparte P1 che, dove era stato interrotto, aggiunge 1 al contenuto dell'accumulatore e lo pone in X. Giusto, no?

NO!, c'è stato un errore: X al termine non contiene 2 ma 1, in quanto P1 è stato interrotto quando già aveva caricato nell'accumulatore il valore 0, al quale (dopo il risveglio) ha sommato 1 e l'ha scritto in X, assolutamente ignaro del fatto che intanto anche P2 aveva fatto lo stesso. Ciò è dovuto al fatto che per incrementare una variabile si è dovuto compiere non una sola istruzione ma tre, e nessuno ci garantiva che sarebbero state eseguite indivisibilmente. Infatti se non avessimo avuto l'interruzione giusto durante l'incremento, tutto sarebbe andato liscio ottenendo effettivamente il valore 2 in X alla fine delle due esecuzioni.

Quindi, quando si accede ad una variabile condivisa per modificarne il contenuto, è bene che non vi siano interruzioni di sorta, che come visto prima possono causare dei veri e propri fallimenti. Per fare ciò bisogna innanzitutto individuare le sezioni (che d'ora in avanti chiameremo «critiche») di programma contenenti modifiche a variabili condivise, per poi implementare opportuni meccanismi che ci garantiscano che se un processo accede ad una di queste sezioni critiche, gli altri non facciano altrettanto finché il primo non ha terminato.

Soluzioni

Nel caso di calcolatore uniprocessor che simula un ambiente multiprogrammato col meccanismo degli interrupt, un metodo abbastanza semplice per rendere indivisibile l'esecuzione di una sezione critica, consiste nel disabilitare le interruzioni con un apposito

Figura 6

(a) P <Semaforo>

```
10 IF ValoreSemaforo=Verde THEN
   ValoreSemaforo=Rosso:GOTO 30
20 <Sospendere il processo chiamante
   sulla coda del Semaforo e mandare
   un processo Pronto in esecuzione>
30 RETURN
```

(b) V <Semaforo>

```
10 IF <Esistono processi sospesi
   sulla coda del Semaforo> THEN
   <Seleziona un processo dalla coda e
   ponilo in stato di pronto>: GOTO 30
20 ValoreSemaforo=Verde
30 RETURN
```

comando al processore. Eseguita la sezione critica, un nuovo comando riabiliterà l'ascolto delle interruzioni. Eventuali interrupt ricevuti durante il «coprifuoco» non vengono persi ma restano pendenti in attesa di poter giungere a destinazione.

In figura 2 è mostrata la sezione critica discussa prima, avvolta dalle due istruzioni per il processore che settano la disabilitazione dell'interruzioni e infine cancellano tale disabilitazione. Se in tale ipotesi come prima dovesse scadere il quanto di tempo subito dopo l'LDA, la commutazione di contesto (il rilascio di P1 e l'esecuzione di P2) sarà ignorata fino all'istruzione CLI che riabilita le interruzioni quando ormai la sezione critica è terminata.

Nel caso di calcolatore multiprocessor, disabilitare le interruzioni del processore che sta eseguendo la sezione critica non basta in quanto P1 e P2 possono evolvere su processori diversi e anche in questo caso bisogna garantire che, quando P1 modifica una variabile condivisa, P2 non faccia altrettanto. Infatti immaginate che P1 e P2, da due processori diversi, eseguano contemporaneamente il famoso X=X+1 tradotto in linguaggio macchina sempre in figura 1. Come prima sia il caso che X all'inizio valga 0 e se due processi la incrementano di 1 alla fine dovrà valere 2. Sia P1 che P2 caricano il contenuto di X negli accumulatori dei due processor. Contemporaneamente gli sommano 1 (attenzione: ognuno nel proprio accumulatore) e ancor tutt'e due assieme riscrivono il risultato di tale somma in X: sbagliato ancora una volta, così facendo X vale 1, non 2.

Brutalmente si potrebbe oltre che disabilitare le interruzioni sul processor che in quell'istante sta eseguendo la sezione critica, bloccare anche gli accessi alla memoria da parte degli altri processor: così saremmo sicuri che nessuno ci rompa (le uova nel paniere). Fatto sta però che se sugli altri processori nessuno aveva intenzione di accedere alla stessa variabile, avremmo inutilmente provocato un arresto temporaneo del rimanente sistema che si è visto negare di colpo l'uso della memoria, senza nessun motivo.

L'idea è allora quella di suddividere le sezioni critiche in classi: due sezioni critiche appartengono alla stessa classe se manipolano le stesse strutture dati condivise. A questo punto non bloccheremo l'accesso a tutta la memoria, ma semplicemente, prima di entrare in una sezione critica, chiuderemo a «chiave» la stessa in modo da assicurarci l'esclusiva. Al termine della nostra sezione critica la lasceremo aperta in modo da permettere ad altri l'accesso. L'esempio tipico che si fa per spiegare meglio questo semplice procedimento è quello dell'appartamento coabitato da più persone dove la sezione critica è naturalmente il bagno: prima di entrare si controlla che non vi sia nessun altro, si accede alla risorsa chiudendocisi dentro e all'uscita si lascia la porta ovviamente non chiusa a chiave per permettere ad altri l'accesso.

In questo modo resteranno bloccati solo i processi che tentano di accedere alle stesse variabili condivise che sta già manipolando uno dei processi in esecuzione. In particolare vogliamo sottolineare il fatto che se su un altro processore gira un processo che non... vuole andare al bagno, questo potrà tranquillamente continuare a fare ciò che stava facendo.

Occorrono però dei meccanismi aggiuntivi con i quali accedere, chiudere o aprire sezioni critiche. Non vorremmo essere noiosi, con l'esempio del bagno: provate però a immaginare cosa accadrebbe se una persona trova la porta chiusa. La cosa più semplice è aspettare, magari, lì davanti. Non appena la porta si riapre potremo comodamente accedere.

Il primo schema di apertura/chiusura di sezioni critiche non si differenzia di molto dall'algoritmo «igienico» appena visto.

Ad ogni sezione critica è associata una chiave (che chiameremo K) che ad esempio contiene il valore 0 se è possibile entrare, 1 altrimenti. Un processo che sta per entrare in sezione critica, esegue oltre al SEI per disabilitare le interruzioni presso il suo processore (vedi fig. 3) anche l'istruzione LOCK sulla chiave K per appropriarsi l'accesso esclusivo.

Al termine, eseguito lo STA «X» l'operazione di UNLOCK sempre sulla chiave K lascerà libero l'accesso alla sezione.

In figura 4 è mostrato in linguaggio BASIC-like il corpo della procedura LOCK e quello della UNLOCK. Il loro funzionamento è assai semplice: la LOCK se K è uguale a 0 lo pone a 1 e basta (vuol dire che la sezione critica era libera quindi l'ha occupata) altrimenti cicla sulla linea 10 finché K non diventa 0 (qualcuno ha lasciato la sezione critica) per poi porlo uguale a 1 alla linea 20 avendo così acquisito l'accesso. La UNLOCK, come è facile immaginare, è molto più semplice dovendo solo rimettere K=0.

Figura 7

```
Monitor MANIPOLAICS
var X:integer
Procedure Entry INCREMENTA
X=X+1:RETURN
Procedure Entry DECREMENTA
X=X-1:RETURN
Procedure Entry STAMPA
PRINT X:RETURN
```

Figura 8

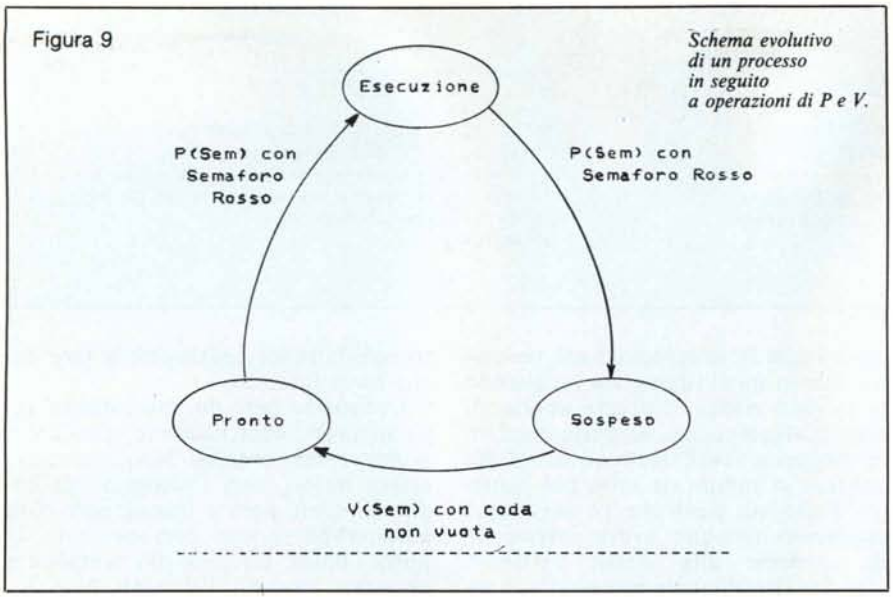
(a) Processo P1

```
10 INCREMENTA
20 ...
30 DECREMENTA
40 ...
50 ...
60 STAMPA
70 ...
80 END
```

(b) Processo P2

```
10 ...
20 INCREMENTA
30 ...
40 DECREMENTA
50 STAMPA
60 DECREMENTA
70 STAMPA
80 END
```

Figura 9



Schema evolutivo di un processo in seguito a operazioni di P e V.

Semafori

I più attenti avranno certamente notato che nell'algoritmo delle LOCK/UNLOCK c'è qualcosa che non va. Non tanto a livello di funzionamento, quanto nel fatto che un processo che trova la «porta chiusa» sta a ciclare sulla linea 10, tenendo così inutilmente occupata la CPU che non può fare dell'altro. Infatti lo diciamo e lo ridiciamo sempre, i calcolatori devono calcolare, non aspettare. Sempre.

E noi così facendo introduciamo della attesa attiva che certo non giova, specialmente dopo tutto quello che abbiamo detto lo scorso mese circa i sistemi multiprogrammati. Lì dicevamo che quando un processo comunicava con un lento dispositivo di ingresso uscita, invece di aspettare risposta attivamente, era più conveniente porlo in stato di attesa e prelevare un nuovo task (lavoro, per non usare sempre la parola processo... oh! pardon) dalla coda dei processi pronti. Associeremo quindi una coda ad ogni sezione critica in modo che il processo se non può accedervi invece di aspettare si sospende sulla relativa coda per non tenere impegnata la CPU. Simmetricamente, quando un processo lascia la sezione critica, se ci sono processi sospesi sulla coda relativa prende il primo di questi e lo pone in stato di «pronto», pronto per essere eseguito non appena si libera un processore.

Tale meccanismo è detto dei Semafori, per l'aspetto comportamentale assai simile alle ben note «primitive» stradali (forse più a quelle ferroviarie). Associamo un semaforo, come prima, ad ogni sezione critica e, guardacaso, questo sarà verde se è possibile accedervi, rosso altrimenti. La struttura dati corrispondente al semaforo è schematicamente mostrata in figura 5: abbiamo un campo Valore (che come detto assumerà Rosso o Verde a seconda del caso) e associato a questo una coda dove sospendere i task che trovano Rosso.

I semafori (ovviamente quali strutture per la mutua esclusione) sono dovuti a E.W. Dijkstra, di nazionalità olandese, il quale ha anche provveduto a dare un nome alle relative primitive di accesso e rilascio della sezione critica. L'operazione P corrisponde alla LOCK di prima, V alla UNLOCK e sono ambedue mostrate in figura 6. Un processo che vuole accedere ad una sezione critica controllata dal semaforo Sem, effettuerà l'operazione P (Sem). Se il semaforo è Verde lo pone a Rosso e può accedere; nel caso contrario (linea 20, sempre di fig. 6a) la stessa P, che è un comando del sistema operativo, sospende il processo che ha eseguito la P e ne seleziona un altro dalla coda dei processi pronti, alla stessa stregua del mese scorso.

Terminata la sezione critica, il processo esegue la V, mostrata in figura 6b: se la coda del semaforo è vuota pone (linea 20) il campo valore a Verde; altrimenti (linea 10 dopo il THEN, abbiamo capovolto un po' la situazione) si prende un processo dalla coda del semaforo e lo si pone in stato di pronto. In tale caso si lascia il semaforo Rosso in modo che sia proprio questo task e nessun altro ad accedere alla sezione critica per primo, non appena un processore si libera e lo preleva per eseguirlo.

Monitor

Cosa c'è ora che non va? Anche i semafori fanno acqua?

No, non è questo che preoccupa: se usati correttamente i semafori vanno più che bene: correttamente, però. Il problema è appunto questo: fidarsi è bene, non fidarsi è meglio (degli utenti). Infatti, se da una parte è vero che con l'uso delle P e delle V si riescono a trattare facilmente le situazioni di mutua esclusione tra processi, è anche vero che se un utente usa un linguaggio di programmazione concorrente con le P e le V, e non fa molta attenzione al loro uso, può provocare più pasticci di quanti se ne sarebbero verificati senza di esse. Se per esempio dimentica di fare la V dopo la sezione critica, o viceversa, o accede in due sezioni critiche l'una dentro l'altra, facilmente si possono creare situazioni di stallo in cui tutto il sistema si paralizza, tutti i processi risultano sospesi, non vi sono processi pronti nè in esecuzione e le CPU stanno con le mani in mano.

Tradotto in altre parole, fino a quanto si tratta di una variabile condivisa tra due task, come visto prima non è assolutamente complicato trattarla adeguatamente. Se però ci sono decine di processi che interagiscono mediante qualche centinaio di sezioni critiche, non fare confusione a furia di colpi di P e V, certamente non è facile. Ecco perché qualcuno ha ben pensato di inventare una apposita struttura detta monitor (i video non c'entrano nulla, n.d.r.) che inglobando le strutture condivise rendono più facile il loro uso corretto, ovviamente senza mai dimenticare qualcosa fuori posto. Un monitor sarà allora una struttura che non fa uso esplicito di P e V, permettendo di compiere ugualmente tutte le operazioni che vogliamo sulle strutture dati condivise.

Per capire meglio, torniamo al nostro solito esempio di due processi che manipolano una variabile condivisa, la X. Immaginiamo che su questa variabile effettueremo operazioni di incremento, decremento e stampa valore (da qualche parte, non ha importanza). Praticamente le operazioni da compiere saranno quelle di definire una struttura monitor come quella di figura 7 alla quale abbiamo dato un nome, MANIPOLAICS, e una sequenza di specifiche. La prima riguarda la variabile condivisa, X nel nostro caso. Seguono delle procedure Entry (sta per ingresso nel monitor) per incrementare, decrementare o stampare il valore di X. A questo punto, se un processo vuole incrementare X basta che invochi la procedura del monitor INCREMENTA; similmente per le altre due possibilità.

Sarà il sistema stesso a garantire che una sola invocazione sia effettuata alla volta, sospendendo automaticamente i processi che eseguono procedure Entry quando già qualcun altro «sta nel monitor».

MC