di Andrea de Prisco

Il passo successivo: i sistemi multiprocessor

Dopo la breve pausa del numero scorso, Appunti di Informatica torna sulle pagine di MC proponendo una passeggiata informatica nel mondo dei calcolatori multiprocessor dotati, ovvero, di più CPU che lavorano in parallelo per eseguire più programmi contemporaneamente

Prologo

Nelle scorse puntate abbiamo visto varie architetture di processori che permettono aumenti di performance grazie alla scomposizione e/o duplicazione di alcune componenti interne dell'architettura standard. Abbiamo visto processor paralleli, processori vettoriali, processori multi-ALU non senza segnalarvi che per sfruttare al massimo tali caratteristiche (e dunque ottenere un reale aumento della velocità di elaborazione del sistema) è necessario che i programmi siano scritti tenendo in massima considerazione l'architettura della macchina sulla quale dovranno girare. Ad esempio per i processor dotati di stadi pipeline e per quelli multi-Alu è necessario minimizzare quanto più possibile i fenomeni di dipendenza logica tra le istruzioni (vi rimandiamo ai numeri precedenti per ulteriori chiarimenti) mentre per i processori vettoriali si ha un aumento della velocità solo se nei programmi facciamo un massiccio uso di calcolo vettoriale. Mentre quest'ultima caratteristica non può, in generale, essere implementata appositamente nei nostri programmi (non possiamo inventarci calcoli di questo genere se non ne abbiamo bisogno effettivo) per quanto riguarda le dipendenze logiche lo sforzo è pressoché obbligatorio se non vogliamo annullare completamente i vantaggi offerti dai processor paralleli. Naturalmente ci aiuteranno principalmente i compilatori appositamente progettati per questi che, partendo da programmi qualsiasi scritti in linguaggi ad alto livello, forniscono codice ottimizzato non in generale ma per l'architettura della macchina su cui dovrà girare.

Il multitasking

Prima di passare ai calcolatori multiprocessor che permettono performance ancora più elevate, occorre fare a priori alcune considerazioni valide tanto per i sistemi uniprocessor che per i sistemi dotati di più CPU. Dovrebbe essere inoltre chiaro che già da un pezzo non parliamo più di sistemi personali ma di calcolatori abbastanza grossi ai quali di solito sono attaccati una nutrita serie di terminali. Tanti terminali, tante persone che utilizzano la loro postazione come un calcolatore «personale» ovvero senza curarsi affatto di ciò che stia succedendo agli altri terminali disseminati magari in altre stanze. Ogni utente avrà la sua visione del sistema operativo, lancerà applicazioni, immetterà dati e otterrà risultati direttamente sul proprio monitor. Come questo sia possibile utilizzando un solo computer centrale per tutti gli utenti, ne abbiamo già parlato in altri articoli di questa serie e lì vi rimandiamo per ulteriori chiarimenti. In questa sede faremo soltanto un piccolo riassunto su come stanno effettivamente i fatti. In sostanza, la CPU divide il suo tempo tra i vari utenti elaborando un pezzetto di programma lanciato dal terminale 1, poi un po' di programma del terminale 2 e così via, ciclicamente e effettuando tali commutazioni molto frequentemente sì da far credere ai vari utenti di dedicare tutto il tempo ad ognuno di loro. Certo i vari programmi impiegheranno tanto più tempo ad essere portati a termine quanti più sono gli utenti collegati in quel momento, ma ciò è sempre molto meglio che fare la fila davanti ad un unico terminale dove le elaborazioni sono molto più rapide.

Questo in linea generale. In pratica, in ogni sistema multitasking (come un sistema multi utente) i programmi da elaborare sono identificati da una lista di descrittori di processo i cui elementi individuano i singoli processi e sono mantenuti in un certo ordine prefissato. Questa è detta lista dei processi pronti (a partire). Una seconda lista, detta dei processi in stato di attesa, contiene i descrittori dei processi che a causa di una richiesta non soddisfabile dalla CPU non possono essere elaborati ulteriormente. Ad esempio quando un programma richiede un dato all'unità a di-

schi e questo tarda ad arrivare, come è normale che sia, una volta confrontate le velocità di una CPU con quelle di un qualsiasi dispositivo elettromeccanico. In casi del genere, il processore per non perdere tempo pone il processo attualmente in esecuzione in stato di attesa e preleva un nuovo processo dalla lista dei processi pronti per l'esecuzione. Un altro caso in cui il processore molla il processo in esecuzione, lo abbiamo già preannunciato, accade quando scade il quanto di tempo che gli doveva dedicare e passa all'elaborazione del successivo programma della lista «pronti». In questo caso, però, il processo appena lasciato non va inserito nella lista stato di attesa, ma nella lista «pronti» dal momento che potrà ripartire quando sarà nuovamente il suo turno. Manca un ultimo anello alla catena: come avviene la transizione da stato di attesa a stato di pronto? Semplice: quando un dato richiesto finalmente arriva (o più in generale, quando l'evento esterno si verifica) un interrupt da dispositivo avverte il processore di effettuare tale transazione: il processo in attesa, aggiornato dal dato mancante, è così posto in stato di pronto. Abbiamo riportato in queste pagine il diagramma di transizione di stato di un processo. I lettori più fedeli riconosceranno che è stato già pubblicato quando abbiamo parlato più approfonditamente di multitasking in queste pagine.

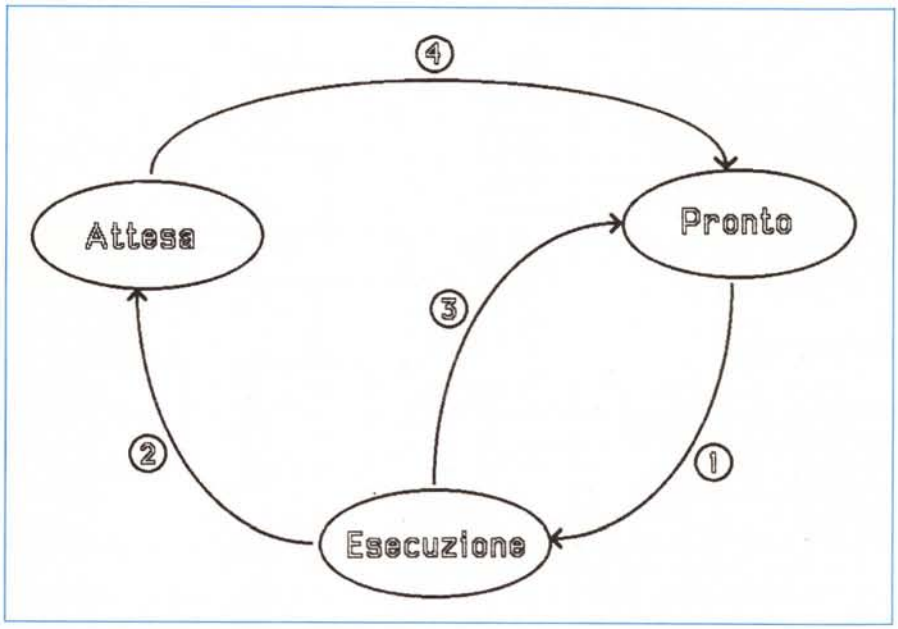

Figura 1 - Stati di un processo e relative transizioni di stato: (1) il processore è libero: un processo in stato di pronto va in esecuzione. (2) A causa di una richiesta non soddisfabile dalla CPU il processo in esecuzione è messo in stato di attesa. (3) Scade il quanto di tempo che il processore poteva dedicare al processo in esecuzione. (4) L'evento non soddisfabile dalla CPU si è verificato e il processo in stato di attesa passa in stato di pronto.

Il passo successivo

Scattando una ideale fotografia ad un sistema multitasking uniprocessore vedremmo dunque un processo in stato di elaborazione, tanti processi fermi in stato di attesa, altri processi altrettanto congelati ma in stato di pronto. Un vero peccato: molti processi potrebbero essere elaborati, ma restano in attesa del loro turno di CPU. Se queste però fossero più d'una, semplicemente avremmo che in ogni istante sono in elaborazione tanti processi quante sono i processori e le transizioni di stato si susseguirebbero con frequenza maggiore senza diminuire il quanto di tempo che ogni CPU dedica al processo da elaborare. Il risultato finale è che ogni utente al terminale, a parità di carico, vedrà elaborare il proprio programma con velocità tanto maggiore quanti più sono i processori disponibili. Il carico è il numero di programmi da elaborare ovvero la somma dei processi in stato di pronto più quelli in stato di attesa e di quelli in esecuzione (pari al numero di processori, come detto). È importante notare che fintantoché il carico è minore o uguale al numero di processori, un aumento del numero di CPU non accelera ulteriormente l'esecuzione di un singolo programma ma semplicemente provoherebbe che i k processori in più non saprebbero proprio cosa fare. Ridotto ai casi estremi, se abbiamo un solo programma da elaborare, per quanti processori siamo in grado di aggiungere il nostro unico programma girerà sempre alla stessa velocità. Esattamente come un gruppo di persone che devono effettuare un viaggio utilizzando un mezzo di locomozione che le porterà a destinazione: fintantoché i posti disponibili sono sufficienti, un aumento di questi non velocizza il trasferimento, se invece i posti sono meno dei passeggeri, per minimizzare il numero dei viaggi possiamo dotare del nostro mezzo di un numero maggiore di posti. Semplice, no? Nella realtà però succede che un sistema di calcolo che si rispetti ha il sistema operativo di per sé multiprogrammato, ovvero non una collezione di routine

attivate dai vari CALL dei programmi, ma un insieme di processi paralleli che collaborano per servire nel migliore dei modi le richieste degli applicativi. Avremo ad esempio un processo driver per ogni unità a dischi, per ogni stampante, per ogni terminale ecc.ecc. Così quando un processo deve accedere ad un dispositivo per scaricare qualcosa (senza dover aspettare né conferma né risposta) può delegare il compito al processo driver specifico e andare avanti senza interruzioni. Dunque vedete che anche con un solo programma da eseguire, ma con un sistema operativo multiprogrammato la presenza di un numero maggiore di processori è tutt'altro che trasparente. Nell'esempio appena riportato, avverrà che una volta delegato il compito di cui sopra, compito e proseguimento dell'elaborazione, avverranno simultaneamente con un effettivo aumento di velocità rispetto al caso uniprocessor.

L'architettura

In figura 2 è mostrato lo schema a blocchi di un sistema multiprocessor. Troviamo un certo numero di CPU, una memoria organizzata in moduli, vari dispositivi di I/O e due strutture di interconnessione denominate Crossbar. Il Crossbar non è che una estensione del bus arbitrato che permette a n utilizzatori di mettersi in comunicazione con m serventi arbitrando eventuali conflitti per l'acquisizione dello stesso servente da parte di più utilizzatori.

Rispetto al bus arbitrato la differenza sta appunto nel fatto che, in assenza di conflitti, tutti gli utilizzatori possono adoperare i serventi mentre nel caso precedente in ogni istante l'utilizzatore è sempre e comunque uno solo.

Tornando allo schema di figura 2, è conveniente che la memoria sia organizzata in modo interlacciato (indirizzi contigui in blocchi contigui) in modo da minimizzare il più possibile la probabilità di conflitto. Sappiamo infatti che se due processori tentano di accedere a due moduli distinti tali accessi avverranno in contemporanea, ma se la richiesta è per lo stesso modulo, uno dei due processori, per forza, dovrà aspettare che l'altro finisca. Ad ogni processore potremo aggiungere una piccola memoria privata atta a contenere informazioni riguardanti il processo in esecuzione sul processore in questione o addirittura parte del sistema operativo. Ovviamente gli accessi alla memoria privata non sono arbitrati in nessun modo dal momento che non v'è alcuna possibilità di conflitto: ogni processore accede solo alla propria memoria privata.

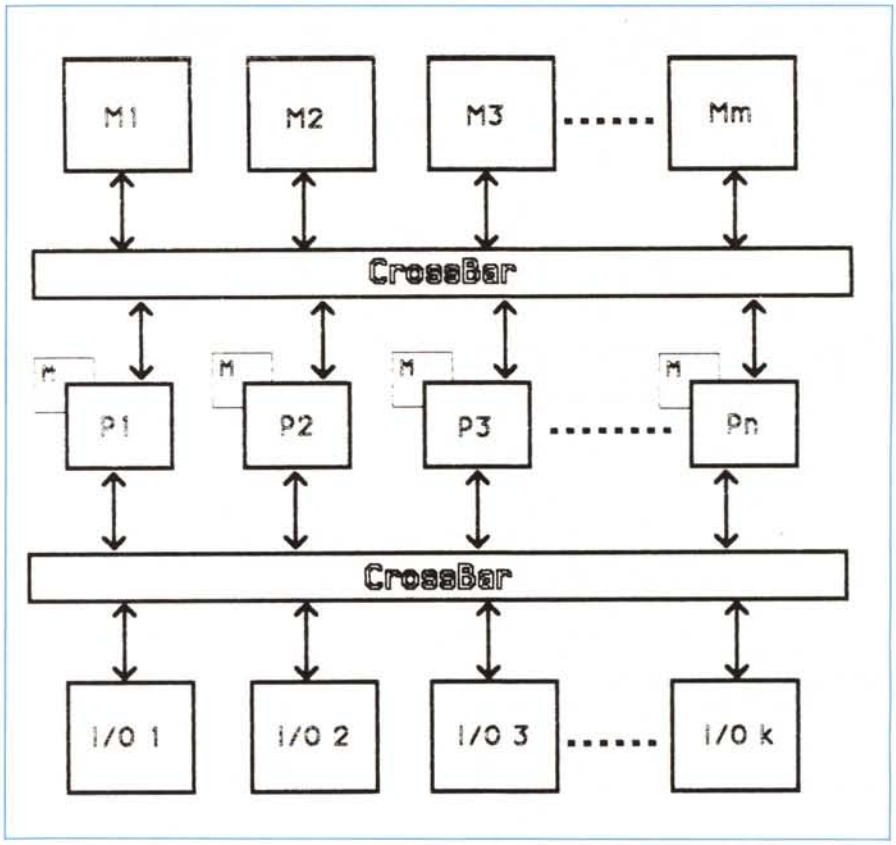

Figura 2 - Schema a blocchi di un calcolatore multiprocessor.

Due per tre uguale cinque

Il titoletto di questo paragrafo è presto spiegato. Utilizzando n processori da k mips l'uno non illudetevi di ottenere un calcolatore multiprocessore con performance reale pari a nk. Sarebbe troppo comodo. Del resto se una Ferrari corre a 300 Km orari, né attaccando due Ferrari, né costruendone una dotata di due motori identici al primo, otterremo un bolide che corre il doppio: c'est la vie!

E non azzardatevi a chiedermi il perché: non sono mica un ingegnere meccanico, né un fisico, né uno al quale la fisica piace particolarmente (specialmente quando si comincia a parlare di rotori...). Ma tornando rapidamente al nostro sistema multiprocessor, il motivo per cui due per tre non fa sei ma qualcosa di meno, è abbastanza intuitivo. Il problema nasce soprattutto dai conflitti dovuti ad accesso da parte di più processori allo stesso modulo. Abbiamo detto che in assenza di questi, utilizzando una struttura di tipo Crossbar fila tutto liscio, ma se più processori necessitano di accedere allo stesso modulo, il meccanismo di arbitraggio (ovviamente presente anche nel Crossbar) concederà l'accesso ad uno solo e gli altri dovranno aspettare.

Aspettare, che brutta parola! Tanto brutta che abbassa il risultato finale della performance quanto più alta è la probabilità di conflitto o, parimenti, quanti meno sono i moduli della nostra memoria. Certo, se fossero infiniti (o meglio, pari al numero totale di celle) avremmo che la performance reale sia pari a quella ideale (nk) ma vi assicuro che un sistema così fatto avrebbe sicuramente qualche problema di costo. Dal momento che, fissato il numero di processori, all'aumentare dei moduli di memoria utilizzati la performance reale approssima asintoticamente quella ideale, è sufficiente prevedere un numero di moduli sì grande, ma non «impossibile» per avere risultati di tutto rispetto. Capito? MC