di Pierluigi Panunzi

I task e l'ambiente multi-tasking

prima parte

Abbiamo parlato già alcune volte del termine task, pur senza approfondire il significato, associando tale concetto a quello di un generico programma in corso di esecuzione: vedremo in questa puntata quali e quanti concetti vengono a loro volta abbracciati ed unificati sotto il termine di task.
Prima di procedere facciamo ancora una volta mente locale non tanto sui concetti finora appresi, ma su quello che è in generale l'ambiente di esecuzione di un task. È ovvio e ben chiaro a questo punto che esiste un'enorme differenza tra l'ambiente operativo di un programma che agisce in un mondo «mono-tasking», quale quello ad esempio di un programma nell'ambito dell'MSDOS, che gira sull'8086 e viceversa un task all'interno di un ambiente multi-tasking supportato dall'80286 appunto

Nell'ambiente 8086 abbiamo il nostro bravo programma che viene eseguito «quasi» indisturbato, dal momento che nemmeno ci accorgiamo che è ad esempio interrotto parecchie volte al secondo dalla routine di «clock»: nel momento in cui l'apposito circuito hardware decide di interrompere il nostro povero programma, ecco che, già lo sappiamo, avviene tutta una serie di operazioni che permettono di sospendere l'esecuzione del programma, iniziare la routine di gestione dell'interrupt, terminarla, per poi ripassare il controllo al programma che era in attesa, senza che nulla, neanche una virgola (o un bit) risulti alterato rispetto al momento in cui il programma era stato interrotto.

La routine di interrupt deve perciò agire in modo del tutto «trasparente» per il programma in corso di esecuzione: sappiamo già che basta un registro alterato, che subito il programma rimane completamente stravolto da questo «passaggio di elefanti».

Qual è in questo caso il meccanismo che sta alla base di questa «trasparenza»? Lo diciamo solo per i neofiti per chi se lo fosse dimenticato...

Basta salvare il Program Counter (o meglio l'Instruction Pointer, IP), il Code Segment (CS) nonché il registro dei flag: salvarli per poi ripristinarli al termine della routine di interrupt (sappiamo a tal proposito che basta l'istruzione IRET).

Ma gli altri registri?! Per i neofiti e gli smemorati di cui sopra ricordiamo che basta ed è necessario (cioè obbligatorio!) salvare nello stack tutti e soli quei registri che vengono usati dalla routine di interrupt e chi meglio della routine stessa sa quali sono i registri che userà: ecco dunque un bel fiorire di istruzioni di PUSH all'inizio della routine di interrupt al quale corrisponde un'analoga serie di istruzioni POP (lo dobbiamo dire?!) con i registri posti in ordine inverso subito prima della fatidica istruzione IRET.

Questo meccanismo è ovviamente generale ed indipendente dal numero di routine di interrupt che possano essere attivate successivamente nel tempo: l'importante è che tutti i registri usati dalla routine vengano preventivamente salvati, altrimenti il tutto non funzionerà più.

Nel caso poi che ad occhio e croce sappiamo che la routine utilizza tutti i registri, ecco che ci può venire incontro un'utilissima istruzione (PUSHA, che sta per «PUSH A11»), la quale salva tutti i registri nello stack, contrapposta all'istruzione POPA («POP A11») che viceversa li ripristina in ordine inverso, in entrambi i casi ottenendo un risparmio notevole di istruzioni e soprattutto di esecuzione: mentre in generale una PUSH di un registro richiede 3 cicli di clock, la PUSHA (che salva rispettivamente AX, CX, DX, BX, l'SP originario, BP, SI e DI), invece di 8x3=24 cicli di clock, ne richiede appena 17.

Analogamente 8 istruzioni di POP (a 5 clock l'una) richiederebbero 40 clock contro gli appena 19 della POPA. Fin qui per quanto riguarda il nostro unico programma...

In un ambiente multi-tasking, invece, come dice il nome abbiamo un numero imprecisato di singoli task (da due, nel caso più favorevole a qualche centinaio se non di più...) che dovranno girare «contemporaneamente»: se il sistema è anche multi-processing allora è possibile che effettivamente due task possano fisicamente essere eseguiti nello stesso istante, mentre nel caso di un solo processor ecco che la «contemporaneità» è solo apparente, fallace, ma quel tanto che basta da ingannarci abilmente.

Il trucco sta nella sapiente gestione dei vari task da parte del cosiddetto «sistema operativo multi-tasking»: esistono parecchie strategie di controllo ampiamente usate, delle quali le due più semplici sono quelle legate al concetto di «priorità» da un lato e di partizione di tempo («time sharing») dall'altra.

In particolare il meccanismo «a priorità» prevede l'assegnazione ad ogni processo di una certa priorità, intesa come possibilità di utilizzare un maggior nu-

mero di risorse e per un tempo maggiore da parte di un certo task: anche qui ci sono varie strategie tra le quali una è di far eseguire le routine più veloci per parecchie volte in un'unità di tempo, lasciando alle routine più pesanti il tempo rimanente.

Questi tipi di strategia «pesata» servono in tutti quei casi in cui abbia senso privilegiare l'esecuzione di certi task rispetto ad altri: se tutto è fatto per bene allora un osservatore esterno ha l'impressione di concorrenza dei vari task ed è proprio in questo caso che si avrà un cosiddetto «scheduler» (un programma «organizzatore»), avente il compito di passare il controllo da un task all'altro, in base a criteri legati alla priorità dei singoli task ed al trascorrere del tempo.

L'altro metodo è invece quello legato più strettamente a tempo e perciò allo scandire di un clock prefissato: improrogabilmente sarà la routine di interrupt del clock (e non un programma di gestione) a far saltare da un task all'altro: il problema che nasce a questo punto è a quale task cedere il controllo ed allora in genere si utilizza un criterio di «round-robin» che prevede l'«accensione» ciclica di un task all'interno di un circolo.

Va da sé che così ad ogni task viene concesso il controllo per un lasso di tempo rigorosamente costante ed in alcuni casi viene inesorabilmente tolto, magari proprio sul più bello...

Questa strategia, come l'altra citata, possiede una serie di vantaggi ed un'immancabile serie di svantaggi, fatto che comporta la scelta di una o più strategie da attuare dinamicamente sia alterando la priorità di un processo sia alterando lo «slice» (lasso) di tempo destinato al task.

Comunque non ci addentriamo oltre in questo cammino, ma vediamo cosa comporta il tutto in termini di programmazione del 286, fatto che soprattutto ci permetterà di analizzare in dettaglio altre caratteristiche fondamentali del nostro beneamato microprocessore.

I task

Dopo aver più volte citato il concetto di «task» ed averne spiegato in modo intuitivo il significato di «singolo processo», diamo un'occhiata alle possibilità, o meglio all'impossibilità, di relazione tra più task.

In definitiva un task risulta un'entità a sé stante completamente isolata da un altro task (e questo è da tenere sempre bene a mente) ed anche se in un certo istante ci sono più task che possono essere eseguiti, solo uno di essi per volta verrà eseguito, secondo le strategie viste prima.

Il passaggio delle consegne tra un task e l'altro è detto in terminologia corrente «task switching» e può essere effettuato o tramite un interrupt (o meglio eseguendo la routine di gestione di un certo interrupt) oppure per mezzo di una «banalissima» istruzione di JMP, CALL o con una altrettanto ben nota IRET.

Anche in questo caso non è stato necessario introdurre nuove istruzioni in quanto, al solito, il passaggio tra un task e l'altro avviene in maniera trasparente per l'utente, il quale sa solo di aver effettuato una CALL o una JMP.

Tutto questo, ormai l'abbiamo imparato, ci dovrebbe far sospettare la presenza di un'ennesima struttura particolare, interna, invisibile al programma finale: questa sorta di intuizione non è

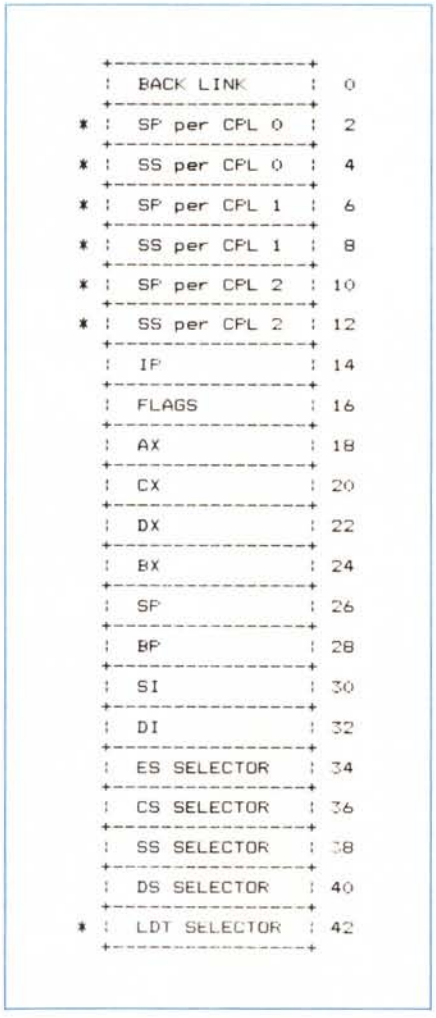

Figura 1 - La struttura di un «Task State Segment» (TSS): i campi indicati con «» vengono inizializzati alla creazione del task, mentre gli altri cambiano all'atto del «task switching».*

sbagliata in quanto andremo ora a conoscere il cosiddetto «Task State Segment», il quale foneticamente è quasi uno scioglilingua, ma che è più pratico indicare con l'altrettanto corrente termine «TSS», ovviamente derivato dalle sue iniziali.

Il TSS non è altro che un particolare segmento di memoria che contiene tutte le informazioni atte ad identificare in maniera univoca ed indipendente un task: in particolare per ogni task, oltre ai suoi segmenti di codice e di dati veri e propri, viene generato appunto il TSS, che perciò rappresenta in un certo senso la carta di identità del task vero e proprio, anzi una sorta di «lasciapassare con fotografia».

Dal momento che si tratta di un segmento è lecito aspettarsi che esso abbia un suo apposito descrittore (qui gli affari si complicano, come al solito...) residente su di una «descriptor table»: in effetti è così (che intuizione!!) ed infatti, associato al TSS, abbiamo il «Task State Segment Descriptor», in gergo il TSSD, che stavolta deve essere posto nella GDT («Global Descriptor Table»), proprio perché deve essere accessibile in ogni istante.

Data l'importanza del ruolo che svolge un task, ecco che è stato introdotto un nuovo registro interno alla CPU, il «Task Register» (TR), il quale, come vedremo in dettaglio tra poco, punta istante per istante al TSSD del task in corso di esecuzione...

Uhm! qui è proprio complicato e conviene andare con calma: ricominciamo con ordine.

Dicevamo dunque che il task è una sorta di mattone dell'edificio costituito dal sistema in funzione: come tale ha bisogno da un lato di un certo numero di segmenti di codice e di dati e da un altro lato di un qualcosa che permetta di conoscere istante per istante a quale punto è arrivato il task stesso, in parole povere il suo «stato».

Sappiamo infatti che i task sono entità a sé stanti, ma che viceversa devono sottostare al meccanismo di «switching», senza che il singolo task abbia una qualsiasi sensazione di essere stato interrotto.

In pratica all'atto del «task switching» il task interrotto deve salvare il proprio stato da qualche parte in memoria, da dove poi riattingere le informazioni per ripristinare lo stato stesso nell'istante in cui viene riattivato.

Ecco che dunque abbiamo bisogno di una zona di memoria (un segmento, appunto) in cui porre tutte le informazioni utili per ridare il controllo ad un task switch-ato: a chi pensava che si poteva salvare il contenuto di tutti i registri

nello stack rispondiamo che il tutto avrebbe comportato insormontabili problemi tra i quali il minore è che la coppia SS:SP doveva poi essere memorizzata da qualche altra parte ed ancora una volta associata al task in questione.

Poi c'era l'altro problema dell'isolamento: uno stack segment non garantisce a questo livello la protezione dello stato di un task da ingerenze più o meno volute di altri task (magari a privilegio maggiore), con il risultato di sconvolgere ancora una volta tutto il sistema.

so fino a quel momento, in termini di livelli di privilegio.

Facendo perciò riferimento alla figura 1, abbiamo bisogno di una tabella (un segmento...) in cui, word dopo word, salvare tutti i registri della CPU, più qualche altra informazione (della quale parleremo fra breve); in particolare nel TSS vengono salvati:

— il cosiddetto «back link»;
— le coppie SS:SP relative ai tre livelli di CPL (0, 1 e 2);
— l'IP relativo all'istruzione da eseguire all'atto della ripresa del task;

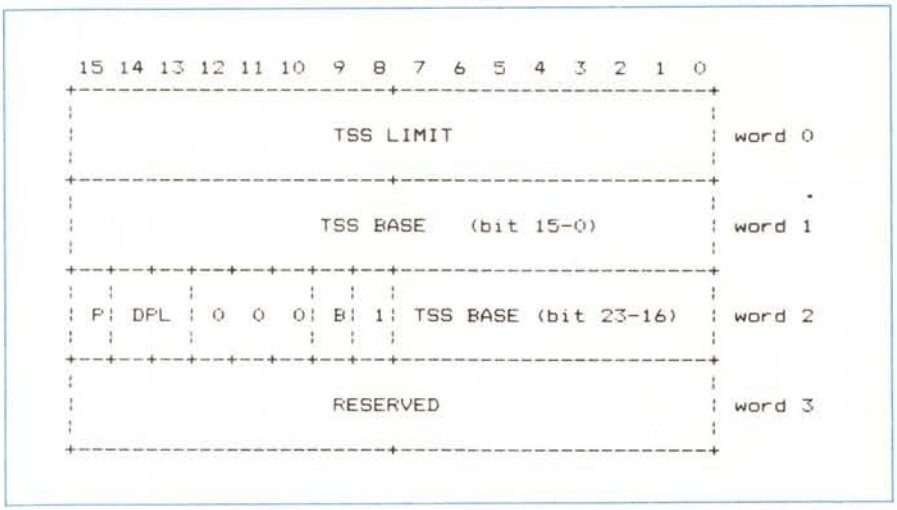

Figura 2 - La struttura di un «Task State Segment Descriptor»: per il significato dei vari campi si veda nel testo.

I solerti progettisti dell'Intel hanno dunque inventato un'altra struttura detta appunto TSS e dunque un segmento «inviolabile» di proprietà del task, inviolabile al punto che nemmeno il task stesso può andarci a scrivere!!!

Dicevamo dunque che per salvare lo stato di un task all'atto dello «switching» abbiamo la necessità di salvare il contenuto di tutti i registri, dei flag, ma ciò non basta, alla luce di quanto già conosciamo: infatti dobbiamo salvare anche l'«LDT selector» (cioè il «selector», puntatore, alla tabella dei descrittori «locali» al task) mentre sappiamo viceversa che la «GDT» è unica ed inoltre dobbiamo fare i conti (lo ricordate?!) con il fatto che all'interno di uno stesso task ci possono essere chiamate a routine aventi privilegi differenti e che perciò richiedono stack separati, uno per ogni livello di CPL.

Questo perché il «task switching» può avvenire in un qualunque istante, magari non appena è stato effettuato un salto ad un segmento di privilegio differente e non deve assolutamente perdersi la «storia» di quanto è succes-

— il Flag Register;
— tutti i registri della CPU compresi i «Segment register» attuali e cioè relativi all'istruzione che si stava eseguendo ed ai dati a cui essa fa riferimento;
— l'«LDT selector» relativo appunto alla tabella di segmenti locali e perciò appartenenti al task.

Per quanto riguarda il «back link», si tratta nient'altro che del «TSS selector» del task che è stato interrotto: a chi suonasse strana questa affermazione («Ma come, non stiamo salvando lo stato di un task interrotto?!...»), rispondiamo che bisogna pensare alla situazione a regime, in cui un task generico interrompe un altro task generico (il quale a sua volta avrà interrotto un altro task, badate bene!) ed ecco che perciò in ogni istante un task attivato saprà qual è il task che ha interrotto.

Inoltre, per maggiore chiarezza, aggiungiamo che in realtà i campi marcati con l'asterisco nella figura 1 sono stati inizializzati dal sistema operativo all'inizio dei tempi, all'atto di creazione del task stesso e non potranno in alcun caso essere modificati.

Viceversa i rimanenti campi saranno alterati in funzione del task che si va ad interrompere (nell'istante in cui il nostro task ne interrompe un altro) ed in funzione dell'istruzione che il nostro task sta eseguendo (nell'istante in cui il nostro task viene interrotto da un altro).

In gergo la parte asteriscata viene detta «statica», contrapposta alla parte «dinamica» che varia a seconda delle situazioni: la parte statica è in un certo senso quella che avevamo chiamata la «carta di identità» del task, mentre la parte dinamica fa le veci di un «super stack», lo ripetiamo, inattaccabile.

Il fatto poi che risulti inattaccabile il TSS di un task interrompente, fa sì che questo task stesso non possa alterare il «back link» del task or ora interrotto il quale ha così la certezza di essere rieseguito non appena il task interrompente è terminato.

In tutto questo è poi ovvio che, per quanto detto all'inizio, il task switching può avvenire anche a partire da un'istruzione all'interno di un task (una JMP o meglio una CALL) e proprio nel caso della CALL il «back link» rappresenta una specie di «super indirizzo di ritorno» relativo ad una «chiamata ad un altro task».

In questo caso dunque in cui lo switch avviene tramite un'istruzione, si sa sempre e comunque l'istante in cui avverrà: ciò non accade invece nel caso di task switching per mezzo di interrupt nel quale è come se il task interrotto, nel bel mezzo del suo programma, effettuasse la chiamata al task di gestione dell'interrupt.

Comunque in entrambi i casi viene garantito, prima o poi, il ritorno al task interrotto e grazie al TSS non si perde alcuna informazione importante.

Vista dunque la funzione del TSS procediamo oltre.

Il TSS Descriptor

Facendo riferimento alla figura 2, diciamo innanzitutto che il descrittore in esame è di tipo alquanto particolare: innanzitutto abbiamo già detto che dovrà comparire all'interno della GDT, in quanto deve essere accessibile in ogni istante in cui si voglia concedere il controllo al task relativo. Dato che già conosciamo altri descrittori, possiamo subito vedere le differenze di un TSSD: innanzitutto viene identificato grazie alla particolare configurazione dei bit 0-4 dell'«Access Rights Byte», quelli che nella figura 2 abbiamo riportato come «000 B 1», laddove sul significato della «B» torneremo tra breve.

In particolare, analizzando i singoli campi abbiamo che:

— il campo «TSS LIMIT» (analogo al LIMIT che in genere indica l'ampiezza di un segmento) in questo caso deve avere un valore pari almeno a 43, in modo da contenere tutte le informazioni richieste ad un TSS. In caso contrario, al momento in cui la CPU controllerà tale valore per poter accedere alle informazioni, verrà generato un errore di «task invalido».
— Il campo «TSS BASE» ovviamente indica l'indirizzo fisico, a 24 bit, della locazione di memoria a partire dalla quale inizia il TSS.
— Il bit «P» («Present») indica se il descrittore contiene valori validi, intesi con il solito significato di «aggiornati». Ricordiamo infatti che per vari motivi legati a spostamenti di blocchi di memoria virtuale richiesti dal sistema operativo, l'indirizzo fisico del TSS potrebbe ad esempio essere variato (è ovviamente lecito aspettarsi che nel frattempo il TSS in esame sia stato salvato su disco!).

Comunque se tale bit vale «1» allora i dati sono validi.
— Il valore contenuto nel campo «DPL» («Descriptor Privilege Level») invece controlla la possibilità da parte di un task di usare il TSSD per effettuare un «task switching», così come si aveva nel caso del «call gate» di cui abbiamo parlato la scorsa puntata.
— Il primo dei bit posti a «0» subito dopo il campo «DPL» invece sta ad indicare che si tratta di un descrittore di un segmento di controllo e come tale non potrà essere caricato nei registri SS, DS ed ES. Se invece si tentasse di caricare uno di tali segmenti con il «selector» relativo ad un descriptor avente tale bit posto a «0», allora si otterrebbe la generazione di un errore, impedendo così ogni possibilità, da parte di un programma, di accedere e modificare le informazioni contenute nel segmento.

Se così non fosse, avendo per ipotesi i privilegi necessari, si potrebbe caricare ad esempio il DS con il selector relativo ad un TSS ed andare così a vedere il contenuto dei suoi 21 campi e, perché no, alterarlo: già dall'inizio avevamo detto che il TSS è una struttura inaccessibile anche al programma a cui si riferisce.

Nella prossima puntata vedremo poi che sarà solo la CPU a poter accedere alle informazioni in esso contenute, secondo meccanismi «trasparenti».
— L'ultimo campo «B» («Busy») infine indica se il task a cui si vuole dare il controllo è già stato attivato oppure no: questo è molto importante perché i task NON possono essere rientranti, come dire che un task (anche per vie traverse) non può richiamare se stesso.

Dunque, non appena un task (inattivo, «idle» e perciò con tale bit posto a «0») viene attivato, viene posto ad «1» tale bit per prevenire eventuali chiamate a tale task anche dopo successive attivazioni di altri task: ricordiamo infatti che tramite il «back link» tutti i task via via attivati ed interrotti rimangono connessi l'uno all'altro e solo al termine effettivo del task tale bit verrà resettato a «0» consentendo così ulteriori chiamate al task stesso.

Con questa ultima dolente nota (sentiamo già i lamenti dei fautori della ricorsività, "pascaliani" in testa...) terminiamo la puntata dando appuntamento alla prossima, nella quale termineremo l'analisi dei «task switching». MC