di Pierluigi Panunzi

Multi-tasking in Time-sharing con il Turbo Pascal

seconda parte

La scorsa puntata abbiamo fornito i primi rudimenti circa la programmazione in Multi-tasking in ambiente Turbo Pascal: prima di proseguire dunque lungo questa strada, tenendo magari sotto mano il numero precedente di MC, ricordiamo che l'oggetto della nostra analisi è una serie di «unit» scritte in TP (dal redattore...) relative all'implementazione di un ambiente Multi-tasking semplificato, che può girare su qualunque personal su cui giri il TP stesso, senza richiedere per forza la presenza di un 286 o meglio di un 386

I processi

Ricollegandoci idealmente con quanto detto la scorsa puntata, ricordiamo dunque cosa si intende con tale termine: più rigorosamente, almeno per quello che ci servirà nel prosieguo, un processo non è altro che una procedura che deve essere eseguibile all'infinito e cioè deve essere dotata al suo interno di un loop infinito, dal quale non si dovrà e potrà mai uscire, pena il blocco totale del computer.

In particolare il «processo» avrà una parte iniziale in cui si inizializzeranno le variabili locali, dopodiché sarà presente un loop infinito, che a scelta del programmatore avrà aspetti «formali» differenti.

La scelta, per quel che riguarda il loop infinito, è ricaduta sul gruppo di istruzioni

```
repeat

[corpo del processo]

until false;
```

Che potrebbe essere tranquillamente sostituita con il seguente gruppo di istruzioni:

```
while true do begin

[corpo del processo]

end;
```

Caldamente sconsigliate, invece, sono forme legate a definizioni di «label» e ad istruzioni di tipo «goto etichetta», se non altro per un doveroso rispetto verso i programmatori seri in Pascal...

Comunque, come è facile vedere andando a disassemblare un programmino di prova dotato di loop infinito (con la «repeat... until», con la «while... do» ed anche con la «goto...»), si ha che in tutti i casi la codifica in Assembler risulta perfettamente identica e perciò in teoria non c'è alcuna controindicazione ad usare una forma rispetto all'altra, a parte gusti o idiosincrasie personali.

Ecco che dunque un processo dovrà essere implementato nel seguente modo (si veda la figura 1):

— indicazione del nome del processo;
— definizione di **const, type, var, procedure** e **function** locali al processo;
— inizializzazione delle variabili locali;
— ciclo infinito di istruzioni costituenti il corpo del processo considerato.

Sottoliniamo ancora una volta il fatto che i processi devono poter girare all'infinito e questa, oltre che una condizione necessaria per poter eseguire un certo processo in multitasking, diventa pure una caratteristica che deve possedere un certo insieme di routine per poter diventare un processo.

Cerchiamo di spiegarci meglio con un paio di esempi:

— l'**output su video** di un carattere in una certa posizione **non** è un processo, in quanto è una funzione che non può e non deve essere eseguita all'infinito (si deve mostrare un certo carattere solo una volta nello schermo e non certo infinite volte): è dunque facile passare a tale funzione un parametro che rappresenterà il carattere da inviare in output;
— l'**attesa della pressione di un tasto** può essere viceversa realizzata come un processo in quanto è facile rendersi conto che la funzione «scansione della tastiera alla ricerca di un tasto premuto» può anche durare all'infinito, nel caso in cui l'operatore non avesse alcuna intenzione o necessità effettiva di premere un tasto: l'importante (e questo sarà automaticamente realizzato) è che il computer non rimanga lì, senza fare assolutamente niente, in attesa della pressione di un tasto;
— il **colloquio con un altro computer** attraverso la linea seriale è ancora una

volta un processo in quanto si tratta di una funzione che prevede il controllo della linea seriale istante per istante, la memorizzazione dei byte ricevuti e viceversa la bufferizzazione ed il successivo invio di quelli da trasmettere.

Gli esempi a questo punto potrebbero essere veramente tanti, e tra l'altro si può vedere che in realtà non è nemmeno detto che alcune funzioni che non sono processi possano essere adattate a diventarlo: infatti la procedura di output verso il video potrebbe diventare un processo (e cioè una routine che viene eseguita all'infinito) non appena ci si accorge che tale routine potrebbe restare lì a ciclare su se stessa, in attesa di un byte da mostrare sullo schermo.

Bisognerebbe allora in questo caso trovare un metodo per passare a tale processo il o i byte da mostrare (metodo che ovviamente esiste e su di esso torneremo nel seguito), senza però far rimpiangere la snellezza e la praticità d'uso di una procedura che compie la sua funzione solo quando viene chiamata.

Per concludere dunque non è detto che per forza bisogna creare dei processi a tutti i costi: al limite un paio di processi appena, ma con una serie di procedure e function ad hoc, possono già realizzare un sistemino di tutto rispetto, in quanto, non dimentichiamocelo, gira in Multi-tasking.

Mai come in questo caso è necessaria un'analisi a monte per poter decidere quali delle funzioni sono processi e quali invece saranno delle funzioni «normali».

Ancora sullo «stack locale»

Ritorniamo ancora su questo argomento per aggiungere alcune considerazioni.

In particolare ad ogni processo che girerà in ambiente Multi-tasking verrà associato (durante un'apposita fase di inizializzazione) un proprio stack locale, inteso come stack sul quale lavora il processo in esame e sul quale, come è ben noto, vengono salvati i contenuti dei registri allorché si abbia un'interruzione: da questo punto di vista il singolo processo dunque vede il proprio stack (indirizzato, come sempre e naturalmente, per mezzo della coppia di registri **SS:SP**) come una zona di memoria a lui associata (e della quale si dovrà fornire l'ampiezza) senza accorgersi dei cambiamenti attuati dallo **scheduler**, della cui opera parliamo tra breve.

Approfittiamo della situazione per dire che ad ogni processo è strettamente associato il valore **corrente** della coppia di registri di cui sopra (**SS:SP**), valore che viene memorizzato istante per istante in un vettore apposito e gestito dal supervisore stesso.

Il tutto apparirà più chiaro dopo aver visto i compiti svolti dal «kernel».

Lo scheduler

Con questo termine abbiamo indicato una parte fondamentale del sistema Multi-tasking in quanto si tratta proprio del blocco di istruzioni che supervisiona il tutto e che consente di passare da un processo all'altro ad intervalli di tempo regolari.

Abbiamo anche detto (e riportato in appositi grafici) che ogni processo ha il possesso della CPU per un intervallo di tempo (detto **slice**) che nel nostro caso è pari a circa 55 msec e cioè esattamente lo spazio di tempo tra due successivi «clock tick» e cioè due battute successive del clock, interno del computer: fin dai primi PC (e per compatibilità il tutto è stato mantenuto nei modelli superiori) infatti il clock di sistema (a partire da 4.77 MHz) veniva diviso per 4 (con un divisore hardware) e successivamente dal timer interno per un fattore pari al massimo possibile e cioè 65536, dando dunque come risultato un periodo di circa 55 msec (relativo a circa 18 Hz).

Nei modelli successivi dotati dei più disparati clock, opportune divisioni a livello hardware («prescaling» in gergo tecnico) nonché a livello software fanno sì che comunque si abbia un «clock tick» ad intervalli di 55 msec.

A partire dunque da un qualunque clock viene generato dal timer un interrupt ogni 55 msec e perciò in risposta a tale interrupt viene eseguita una routine il cui ruolo principale è di modificare dei contatori che servono a mantenere l'informazione dell'ora e della data: tali contatori non servono più laddove esiste un orologio interno, dotato di batteria in tampone, e che sovrintende alla gestione dell'ora e della data, ma ovviamente sono stati lasciati per compatibilità, anche perché usati dal DOS, che viceversa per funzionare non deve affidarsi ad un clock che in alcuni modelli non esiste proprio...

Prendendo spunto da quelli che sono i criteri alla base della programmazione in Multi-tasking con il 286 e con il 386 (si vedano a tal proposito le rubriche rispettive, sempre del medesimo autore...) ecco che anche in questo caso si è sfruttato un principio base legato alle seguenti considerazioni, che prevedono innanzitutto l'esistenza nel nostro sistema, oltre al supervisore, di «n» processi che vogliamo far eseguire «contemporaneamente».

Per passare dunque il «possesso» del microprocessore e delle sue risorse da un processo all'altro, si sfrutta un'apposita routine d'interrupt (che viene attivata ad intervalli perfettamente regolari),

```
procedure nome_del_processo;
{ definizione di costanti, tipi e variabili locali }
{ definizione di procedure e funzioni locali}
begin
 { inizializzazione variabili locali }
 repeat
  { corpo del processo }
 until false;
end;
```

Figura 1 - Questo è lo schema base secondo cui devono essere scritti i processi per poter girare correttamente nell'ambiente Multi-tasking (che ricordiamo è stato battezzato con la sigla TPMT).

secondo un meccanismo alquanto sottile, che prevede il cosiddetto «task-switching».

Per analizzare questo meccanismo, supponiamo di trovarci «a regime» e cioè in un istante qualsiasi di funzionamento del sistema ed in particolare di voler vedere (come al rallentatore ed a partire da un istante iniziale) che cosa succede per un certo periodo di tempo.

Senza perdere di generalità supponiamo che, proprio in quell'istante iniziale, sia in corso di esecuzione uno dei processi, che indicheremo con «A»: è proprio in questa situazione che il processo vede la CPU tutta per sé, senza alcuna ingerenza dall'esterno, se non, ad esempio, a causa eventuali interrupt dal «mondo esterno» provocati ad esempio da parte della tastiera o dalla porta seriale. Eventi come questi però non fanno parte del sistema Multi-tasking, ma viceversa sono da considerarsi una normale appendice del funzionamento dei singoli processi e come tali sono gestiti normalmente.

Ecco che dunque, allo scadere del **clock tick**, deve essere eseguita la routine di servizio dell'interrupt del timer (INT 8): in questo caso sappiamo bene che proprio all'inizio della routine di servizio devono essere salvati nello stack (con istruzioni **PUSH**) tutti i registri e questo è proprio quello che viene fatto...

In particolare i registri vengono salvati nello stack «locale» del processo interrotto e ciò è di fondamentale importanza: se non fossimo in un sistema multitasking, allora le istruzioni di **POP** che si trovano subito prima dell'istruzione **IRET**, di termine della routine di servizio dell'interrupt, ripristinerebbero correttamente i registri estraendone i contenuti dallo stack (che nel nostro caso è proprio quello «locale»), dopodiché il processo che era stato interrotto riprenderebbe ad eseguire le proprie istruzioni, come se nulla fosse successo.

Il fatto di essere viceversa in un ambiente Multi-tasking comporta invece una grande differenza nel comportamento della routine di interrupt: quest'ultima, contravvenendo alle regole «conservative» ben note che prevedono il salvataggio ed il ripristino dei registri, effettua invece uno stravolgimento controllato dei registri in esame ed in particolare proprio della coppia **SS:SP** per ottenere appunto il **task-switching**.

Supponendo dunque di avere prescelto il criterio di far eseguire gli «n» processi in sequenza, uno dopo l'altro, secondo le regole del cosiddetto «roud robin», ecco che, interrotto il processo i-esimo, questo deve essere temporaneamente abbandonato, per passare il controllo al processo i+1-esimo: relativamente a questo processo il supervisore conosce esattamente (perché ad esso associata) la posizione in memoria dello stack locale.

Per ottenere il task-switching bisogna ora:

— memorizzare il valore corrente della coppia SS:SP nel vettore di cui abbiamo parlato nel paragrafo precedente (proprio in corrispondenza dell'elemento relativo al processo che è stato interrotto);

— caricare i registri SS:SP con i valori memorizzati in quello stesso vettore, ma stavolta relativi al processo da «innescare»: in tal modo ora lo stack a cui tale coppia di registri punta è proprio lo stack locale del processo da attivare, nel quale (è qui la chiave di volta) erano stati memorizzati con delle PUSH tutti i registri, proprio in occasione dell'ultima volta che quel processo era stato interrotto!

— Effettuare le POP per ripristinare i registri in esame ed eseguire la IRET: a questo punto ci troviamo dunque immersi nell'ambiente del processo i+1-esimo, proprio come se questo processo fosse stato interrotto temporaneamente da una routine di servizio di un interrupt, mentre in realtà nel frattempo sono stati eseguiti gli altri «n-1» processi...

Fondamentalmente è proprio tutto qui il nocciolo del meccanismo del task-switching, capito il quale si è veramente a buon punto.

Nel nostro caso, in particolare, come è stato più volte detto, si sono simulati per quanto possibile via software i meccanismi hardware che stanno alla base del funzionamento in modo protetto del 286 e del 386.

Visto dunque qual è il meccanismo che regola il task-switching in una situazione a regime, rimane da vedere cosa deve succedere nella fase transitoria iniziale: in questo caso infatti, riprendendo il ragionamento precedente, abbiamo che nessun processo è stato già eseguito e perciò nessuno di essi può essere materialmente interrotto dallo scheduler in quanto lo scambio dei contenuti dei registri SS:SP comporterebbe effetti disastrosi di blocco del sistema!

Ecco che dunque deve esistere una ben precisa temporizzazione degli eventi, seguita la quale si ha un comportamento ottimale: tale sequenza di eventi avverrà in modo automatico seguendo le indicazioni che riportiamo nel seguito ed usando in particolare due **procedure** di inizializzazione dei processi e del Multi-tasking.

```
uses {unita' che implementano il multi-tasking};

 { definizione di const, type e var del programma }

procedure { nome del processo }
 { definizione dei singoli processi }

begin   { main }
 { inizializzazioni varie }

 defineprocess(processo,stacksize,attivo);
     { per ogni processo }

 multitask;  { attivazione del multitasking }

 { eventuali routine di chiusura }

end.
```

Figura 2 - Struttura base di un sistema Multi-tasking dotato di un insieme di processi, definiti per il TPMT con la defineprocess e per il TP come delle semplici procedure (seguendo quanto citato nel testo), ed attivabile con la chiamata alla procedura multitask della quale parleremo nella prossima puntata.

La definizione di processi

Dopo aver visto cosa sono i processi e come si codificano, passiamo ora ad analizzare come si fa a comunicare al TPMT la loro presenza e le caratteristiche ad essi legate.

Ciò si ottiene con la procedura **defineprocess**, che accetta tre parametri e che viene chiamata nel modo seguente

defineprocess (processo, stacksize, attivo)

Ma vediamo il dettaglio dei parametri:

— «processo» è una variabile di tipo procedure in quanto deve contenere il nome della procedura che rappresenta il processo: entrando nei dettagli tecnici, la possibilità di passare una procedura oppure una function come parametro di un'altra procedura richiede innanzitutto l'uso della versione 5.0 (o successive) del TP e poi, dal punto di vista programmativo, l'uso della direttiva {**$F+**}, la quale forza la generazione di procedure di tipo **far**, i cui indirizzi sono composti da segment-offset invece che del solo offset;
— «stacksize» invece è una variabile di tipo integer che conterrà il numero di byte di lunghezza dello stack locale relativo al processo che si sta definendo: in questo caso il valore può essere impostato solo per tentativi, lasciando sempre attiva la direttiva {**$S+**} (controllo dello stack attivo all'ingresso di ogni procedura), aumentandolo allorché il programma si fermi per un run-time error di tipo «**stack size overflow**»;
— «attivo» è infine una variabile di tipo boolean che consente di stabilire se il processo in esame deve essere subito attivato nell'ambiente Multi-tasking oppure deve rimanere disattivo in attesa di un certo evento che lo risvegli.

A questo scopo segnaliamo che oltre al vettore contenente, per ogni processo, il valore corrente della coppia SS:SP, esiste un altro vettore «di stato» che contiene l'informazione relativa allo stato corrente del processo: attualmente nel TPMT sono stati implementati i seguenti tre stati:
— «inattivo» (valore nullo nel vettore) indicante che il processo non deve essere seguito;
— «attivo» (valore uguale ad 1) indicante che il processo è correntemente da eseguire e perciò nello slice di tempo di sua competenza potrà accedere alle risorse di sistema;
— «in attesa da mailbox» (valore uguale a 2) indicante che il processo in esame è in attesa di un messaggio in una mailbox: su questi concetti torneremo nell'immediato prosieguo in quanto ora appesantirebbero notevolmente l'esposizione.

Come primissimo assaggio di quello che vedremo nelle prossime puntate, riportiamo in figura 2 la struttura base di un sistema funzionante in Multi-tasking: in questo si notano chiamate a procedure già note ed altre ancora sconosciute. Un po' per volta le analizzeremo...

Con questi concetti terminiamo la puntata e diamo appuntamento alla prossima, dove parleremo innanzitutto di come vengono predisposti all'inizio i vari stack locali relativi a tutti i processi definiti in modo tale da consentire la «partenza» del Multi-tasking: inizieremo pure l'analisi delle singole procedure che consentono tutti questi meccanismi.

Abbiamo già detto che in alcuni casi l'analisi risulterà alquanto complessa, in quanto richiederà conoscenze sia di Assembler che di TP e per questo motivo consigliamo i lettori interessati a dare un'occhiata alle caratteristiche «interne» del Turbo Pascal, riportate in forma veramente ottima in un apposito capitolo, intitolato «Inside Turbo Pascal», del manuale d'uso.

MC