L'interrupt dei due mondi

Il mese scorso abbiamo visto come servirsi dello spooler del DOS da un programma Pascal. La unit PRNSPOOL funziona correttamente se usata da un programma compilato per il modo reale, ma se il programma è compilato per il modo protetto, riconosce la presenza dello spooler solo in una sessione DOS di Windows, non se eseguito sotto DOS «normale». Si tratta di un esempio dei problemi che si possono incontrare quando occorre una qualche comunicazione tra il modo protetto ed il modo reale, oppure tra un programma compilato per il modo protetto ed un interrupt del DOS

di Sergio Polini

Vedremo ora una versione per il modo protetto della unit illustrata il mese scorso. Contrariamente alle consolidate abitudini della rubrica, tuttavia, cominceremo dal fondo.

Si tratterà di una unit destinata unicamente al modo protetto, che non darebbe i risultati sperati se usata in un programma compilato per il modo reale. Sarà semplice distinguerla dalla unit PRNSPOOL grazie ad un nome diverso, ad esempio PROSPOOL, ma sarebbe comunque utile un meccanismo automatico. Durante le fasi di sviluppo, io stesso mi sono troppe volte trovato a compilare «per il modo sbagliato», al punto che, per evitare errori, ho attribuito particolari nomi in codice alle diverse versioni sia della unit che del programma di prova (URMINT2F, UPMINT2F, UPMINT31 e simili).

Le specifiche DPMI comprendono anche, oltre a quelle disponibili mediante l'INT 31h, alcune funzioni accessibili mediante l'INT 2Fh. Assegnando il valore 1687h al registro AX, ad esempio, si ottengono informazioni circa la presenza e la versione di un server DPMI e, se il server è presente, l'indirizzo di una procedura che può essere chiamata per passare dal modo reale al modo protetto. Qualsiasi programma Pascal compilato per il modo protetto provvede a ciò automaticamente; quella funzione, inoltre, va chiamata in modo reale. Ai nostri fini, quindi, è più interessante l'informazione che si ottiene con lo stesso interrupt ponendo 1686h in AX: se al ritorno AX vale zero, l'interrupt è stato chiamato da un programma eseguito in modo protetto grazie ad un server DPMI, altrimenti o si è in modo reale, oppure non si è sotto DPMI.

La sezione di inizializzazione della unit (figura 1) usa questo metodo per riconoscere l'ambiente in cui viene eseguita ed eventualmente provocare l'immediato arresto del programma. Se invece si ha conferma dell'esecuzione in modo protetto, si provvede ad accertare se si ha a che fare con una implementazione DPMI a 16 o a 32 bit, assegnando quindi 16 o 32 alla variabile *DPMIBits*.

Si usa a questo scopo la funzione 0400h dell'INT 31h. Questa rende nel registro AX la versione del server DPMI e nel registro CX il tipo del processore; i primi bit del registro BX vanno invece intepretati come flag: il primo è settato se l'implementazione DPMI consente l'esecuzione di programmi a 32 bit, il secondo se si ritorna al modo reale quando viene intercettato un interrupt (invece di usare un task V86), il terzo se è supportata la memoria virtuale. Il tutto è esemplificato nel breve programma illustrato nelle figure 2 e 3.

```
var
  Reg: Registers;

begin
  Reg.AX := $1686;
  Intr($2F, Reg);
  if Reg.AX <> 0 then begin
    Writeln('Unit da compilare per il modo protetto!');
    Halt(1);
  end
  else begin
    Reg.AX := $0400;
    Intr($31, Reg);
    if (Reg.BX and 1) <> 0 then
      DPMIBits := 32
    else
      DPMIBits := 16;
  end;
  HeapError := @HeapFunc;
end.
```

Figura 1 - La sezione di inizializzazione della unit PROSPOOL. Chiamando l'interrupt 2Fh con 1686h in AX, si può verificare se il programma viene eseguito in modo protetto sotto DPMI. Se così è, la funzione 0400h dell'INT 31h ci dice se si tratta di un'implementazione DPMI a 16 o 32 bit.

Simulazione di un interrupt reale

Un programma eseguito in modo protetto può valersi delle funzioni da 0300h a 0306h dell'INT 31h per chiamare routine del modo reale non emulate direttamente dal server DPMI.

I registri ES:DI (o ES:EDI se si usa un'implementazione a 32 bit delle specifiche DPMI) devono contenere il selettore e l'offset di una struttura di dati detta *real mode call structure*, illustrata nella figura 4. I primi campi di questa sono tutti di tipo DWORD e, quindi, si possono passare al codice che verrà eseguito in modo reale anche valori a 32 bit; per contro, le implementazioni a 16 bit del DPMI ignoreranno sia i 16 bit più alti di quei campi, sia i registri FS e GS.

I campi *SS* e *SP* possono essere lasciati nulli; in questo caso, il DPMI appronta automaticamente uno stack per

```
Program ChkDPMI;
(* Da compilare per il modo protetto *)
uses
  DOS;

var
  Reg: Registers;

begin
  Reg.AX := $1686;
  Intr($2F, Reg);
  if Reg.AX = 0 then begin
    Reg.AX := $0400;
    Intr($31, Reg);
    Write('Processore ');
    case Reg.CL of
      2: Writeln('80286');
      3: Writeln('80386');
      4: Writeln('80486');
    end;
    Write('Versione DPMI: ');
    Writeln(Reg.AH, '.', Reg.AL);
    if (Reg.BX and 1) <> 0 then
      Writeln('Implementazione a 32 bit')
    else
      Writeln('Implementazione a 16 bit');
    if (Reg.BX and 2) <> 0 then
      Writeln('Ritorno al modo reale per interrupt')
    else
      Writeln('Modo V86 per interrupt');
    if (Reg.BX and 4) <> 0 then
      Writeln('Memoria virtuale supportata')
    else
      Writeln('Memoria virtuale non supportata');
  end
  else
    Writeln('Programma non eseguito sotto DPMI');
end.
```

◀ *Figura 2 - Un breve programma che mostra come utilizzare la funzione 0400h dell'INT 31h per ottenere informazioni sull'ambiente in cui vengono eseguiti programmi compilati per il modo protetto.*

Figura 3 - L'output del programma CHKDPMI (figura 2), eseguito prima sotto DOS (con il DOS Extender fornito insieme al Pascal 7.0) e poi in una sessione DOS di Windows in modo 386 avanzato. ▶

```
CHKDPMI eseguito da DOS
-----------------------

Processore 80386
Versione DPMI: 0.90
Implementazione a 16 bit
Modo V86 per interrupt
Memoria virtuale non supportata


CHKDPMI eseguito da una sessione DOS
di Windows in modo 386 avanzato
-----------------------------------

Processore 80386
Versione DPMI: 0.90
Implementazione a 32 bit
Modo V86 per interrupt
Memoria virtuale supportata
```

l'esecuzione in modo reale (circa una trentina di word; se si prevede l'uso di uno stack più ampio occorre provvedere espressamente, avvalorando di conseguenza i campi *SS* e *SP*).

Dopo la chiamata dell'INT 31h, la struttura conterrà i valori di tutti i registri – tranne *SS*, *SP*, *CS* e *IP* – come impostati dal codice eseguito in modo reale, consentendo così al codice eseguito in modo protetto di acquisire le informazioni risultanti dall'esecuzione.

È possibile anche passare parametri al modo reale attraverso lo stack; la figura 5 mostra come fare quando si usa la funzione 0301h dell'INT 31h, che consente di chiamare direttamente una procedura da eseguire in modo reale, assegnando il suo indirizzo ai campi *CS* e *IP* della *real mode call structure*. Il codice chiamato si troverà i parametri nello stack come indicato nella figura 6.

La unit PROSPOOL utilizza invece la funzione 0300h, che permette di chiamare un interrupt del modo reale il cui numero, 2Fh nel nostro caso, va posto nel registro BL. I registri ES:(E)DI conterranno il selettore e l'offset di una *real mode call structure*, ad alcuni campi della quale verranno assegnati i valori che si attribuirebbero ai corrispondenti registri se si operasse in modo reale. Il registro CX sarà zero, in quanto non passeremo parametri attraverso lo stack. Sarà zero anche il registro BH; il bit 0 di questo va azzerato per indicare che, qualora l'implementazione DPMI di cui si dispone ritorni al modo reale per eseguire l'interrupt, va disabilitata la linea A20 del processore (quella che abilita indirizzi oltre il primo megabyte); gli altri bit sono riservati e vanno azzerati.

La funzione 0300h ignora i campi *CS* e *IP* della struttura puntata da ES:EDI, in quanto l'indirizzo della routine da eseguire viene tratto dal numero dell'interrupt posto in BL. Qualora non fossero nulli i campi *SS* e *SP*, invece, si assumerebbe che essi definiscono uno stack approntato per l'esecuzione in modo reale; nel nostro caso, quindi, è importante assicurarsi che valgano entrambi zero prima di chiamare l'interrupt 31h.

Nella figura 7 è riprodotta la prima parte dell'implementazione della unit PROSPOOL; rispetto a PRNSPOOL, troviamo in più la dichiarazione di un record TRMCS (traduzione in Pascal della *real mode call structure* illustrata nelle specifiche DPMI), di una variabile *RMCS* di tale tipo, di una variabile intera *ErrInt31*, della variabile *DPMIBits* cui abbiamo già accennato e di una procedura *RealModeInt2F*.

La procedura *RealModeInt2F* viene usata per chiamare l'interrupt 2Fh mediante la funzione 0300h dell'interrupt 31h. Questa richiede che l'indirizzo di una struttura di tipo TRMCS venga assegnato alla coppia di registri ES:DI o ES:EDI, secondo che si operi in un'implementazione a 16 oppure a 32 bit delle specifiche DPMI; in concreto, si potranno assumere 32 bit solo nel caso di una sessione DOS di Windows in modo 386 avanzato, 16 bit negli altri casi (se il processore è un 80286 o comunque se si esegue il programma sotto DOS – usando così il DOS Extender fornito con il compilatore – o sotto Windows in modo standard).

La procedura quindi, dopo aver assegnato il segmento della variabile *RMCS* al registro ES, esamina il valore della variabile *DPMIBits*; se questa vale 16, si assegna al registro DI l'offset di *RMCS*, altrimenti si ricorre ad un artificio per provocare l'assegnazione al registro EDI. L'assembler incluso nel compilatore, infatti, non è in grado di generare il codice corrispondente ad istruzioni che abbiano come operandi registri a 32 bit (EDI viene rifiutato come identificatore sconosciuto). Ciò può accadere anche nella programmazione in assembler 386, in quanto è possibile definire segmenti di codice o di dati sia a 16 che a 32 bit (ciò si rifletterà nel *default bit* dei relativi descrittori, di cui avevamo discusso a dicembre); gli operandi di un'istruzione hanno una dimensione di default, che è quella indicata nella definizione del segmento di codice cui appartengono; sono quindi disponibili appositi «prefissi», mediante i quali si può ottenere che gli operandi o gli indirizzi di

Offset	Valore	Tipo
0	EDI	DWORD
4	ESI	DWORD
8	EBP	DWORD
12	Riservato	DWORD
16	EBX	DWORD
20	EDX	DWORD
24	ECX	DWORD
28	EAX	DWORD
32	Flags	WORD
34	ES	WORD
36	DS	WORD
38	FS	WORD
40	GS	WORD
42	IP	WORD
44	CS	WORD
46	SP	WORD
48	SS	WORD

Figura 4 - I campi della struttura di dati da utilizzare per le funzioni da 0300h a 0306h dell'INT 31h (real mode call structure).

```
Param1      SS : SP+8
Param2      SS : SP+6
Param3      SS : SP+4
CS          SS : SP+2
IP          SS : SP
```

Figura 6 - Lo stack come apparirà alla routine eseguita in modo reale chiamata dal codice della figura 3.

un'istruzione vengano interpretati come valori a 32 bit in un segmento di codice a 16 bit, o viceversa. Il byte 66h è il prefisso che provoca l'interpretazione non-di-default degli operandi. Poiché i segmenti di codice generati dal Pascal 7.0 sono a 16 bit, premettendo DB 66h all'istruzione MOV DI, OFFSET RMCS questa viene interpretata come MOV EDI, OFFSET RMCS. Non basta. L'offset di *RMCS* è una word, in quanto riferito ad un segmento dati che può avere come dimensione massima 64 KByte; occorre trasformare la word (16 bit) in una dword (32 bit). Durante l'esecuzione, infatti, il processore vorrà caricare 32 bit nel registro EDI; se non troverà un valore a 32 bit per l'offset, userà un valore composto dall'offset a 16 bit e dai primi 16 bit dell'istruzione successiva, con ovvi catastrofici risultati. Si rimedia aggiungendo 16 bit nulli con DW 0000h.

La procedura chiama quindi l'interrupt 31h. Se si verifica un errore, al ritorno il flag carry è settato; in questo caso, si assegna il valore –31h alla variable *ErrInt31*.

```
PUSH  Param1            ; tre parametri di tipo WORD
PUSH  Param2
PUSH  Param3
MOV   AX, SEG RealModeCallStructure
MOV   ES, AX
MOV   EDI, OFFSET RealModeCallStructure
MOV   CX, 3             ; 3 sono le word passate allo stack
MOV   AX, 0301h         ; funzione di chiamata di codice reale
INT   31h
ADD   SP, 6             ; toglie i parametri dallo stack
```

Figura 5 - Esempio di chiamata di una routine da eseguire in modo reale, il cui indirizzo è stato assegnato ai campi CS *e* IP *della* Real Mode Call Structure, *con passaggio di tre parametri nello stack.*

```
implementation

uses Dos;

type

  PPacket = ^TPacket;                (* per INT 2Fh, sottofunzione $01 *)
  TPacket = record
    Level: Byte;
    Path: PFileName;
  end;

  TRMCS = record                     (* Real Mode Call Structure *)
    EDI, ESI, EBP, Reserved, EBX, EDX, ECX, EAX: Longint;
    Flags, ES, DS, FS, GS, IP, CS, SP, SS: Word;
  end;

procedure TFileList.FreeItem(Item: Pointer);
begin
  StrDispose(PChar(Item));
end;

var
  RMCS: TRMCS;
  ErrInt31: Integer;
  DPMIBits: Integer;                 (* implementazione a 16 o 32 bit *)

procedure RealModeInt2F; assembler;
asm
  MOV  AX, SEG RMCS
  MOV  ES, AX
  CMP  DPMIBits, 16
  JE   @1
  DB   66h                           (* equivalente a:          *)
  MOV  DI, OFFSET RMCS               (*   MOV EDI, OFFSET RMCS  *)
  DW   0000h                         (*                         *)
  JMP  @2
  @1:
  MOV  DI, OFFSET RMCS
  @2:
  MOV  BL, 2Fh
  MOV  BH, 0
  MOV  CX, 0
  MOV  AX, 0300h
  INT  31h
  MOV  AX, 0
  JNC  @3
  MOV  AX, -31h
  @3:
  MOV  ErrInt31, AX
end;
```

Figura 7 - La prima parte dell'implementazione della unit PROSPOOL.

I nuovi metodi

L'interfaccia della unit PROSPOOL è identica a quella di PRNSPOOL; è anche identica l'implementazione della classe *TFileList* e di alcuni metodi della classe *TSpooler*: il destructor *Done*, *FirstInQueue* e *NextInQueue*.

Nella figura 8 sono riprodotti il constructor *Init* e i metodi *HoldForStatus*, *CancelAll* e *ReleaseHold*, che fanno tutti uso della procedura *RealModeInt2F*.

Nella versione vista il mese scorso, il constructor chiamava l'interrupt 2Fh con il valore 0100h in AX per verificare la presenza dello spooler del DOS. Il

```
constructor TSpooler.Init;
begin
  Err := 0;
  FillChar(RMCS, SizeOf(RMCS), 0);
  RMCS.EAX := $0100;
  RealModeInt2F;
  Err := ErrInt31;
  if (Err <> 0) or ((RMCS.Flags and fCarry) <> 0)
  or ((RMCS.EAX and $000000FF) <> $FF) then
    Fail
  else begin
    Queue := New(PFileList, Init(10, 5));
    if Queue = nil then
      Err := ERRNOLIST;
  end;
end;

procedure TSpooler.HoldForStatus;
var
  Reg: Registers;
  P: PChar;
begin
  if Err <> 0 then Exit;
  FillChar(RMCS, SizeOf(RMCS), 0);
  RMCS.EAX := $0104;
  RealModeInt2F;
  Err := ErrInt31;
  if Err = 0 then
    if (RMCS.Flags and fCarry) <> 0 then
      Err := RMCS.EAX and $0000FFFF;
  if (Err = 0) and (Queue <> nil) then begin
    Queue^.FreeAll;
    Reg.AX := $0002;
    Reg.BX := RMCS.DS;
    Intr($31, Reg);
    if (Reg.Flags and fCarry) = 0 then begin
      P := Ptr(Reg.AX, RMCS.ESI);
      while P^ <> #0 do begin
        Queue^.Insert(StrNew(P));
        P := P + 64;
      end;
    end
  end;
end;

procedure TSpooler.CancelAll;
begin
  if Err <> 0 then Exit;
  FillChar(RMCS, SizeOf(RMCS), 0);
  RMCS.EAX := $0103;
  RealModeInt2F;
  Err := ErrInt31;
  if Err = 0 then
    if (RMCS.Flags and fCarry) <> 0 then
      Err := RMCS.EAX and $0000FFFF;
end;
procedure TSpooler.ReleaseHold;
begin
  if Err <> 0 then Exit;
  FillChar(RMCS, SizeOf(RMCS), 0);
  RMCS.EAX := $0105;
  RealModeInt2F;
  Err := ErrInt31;
  if Err = 0 then
    if (RMCS.Flags and fCarry) <> 0 then
      Err := RMCS.EAX and $0000FFFF;
end;
```

Figura 8 - Il constructor della classe TSpooler *e i metodi* HoldForStatus, CancelIAll *e* ReleaseHold.

flag carry settato, indice di un errore nell'esecuzione dell'interrupt, o un valore diverso da FFh nel registro AL provocavano la chiamata della procedura *Fail*. Nella versione per il modo protetto, si procede in primo luogo all'azzeramento di tutti i campi della variabile *RMCS*, per evitare che valori indesiderati provochino errori (si pensi in particolare ai campi *SS* e *SP*). Si assegna quindi il valore 0100h al campo *EAX* di *RMCS* e si chiama la procedura *RealModeInt2F*, che chiama l'interrupt 2Fh mediante la funzione 0300h dell'INT 31h.

Si devono poi verificare tre possibili situazioni di errore: in primo luogo può fallire la chiamata dell'INT 31h; in questo caso la variabile *ErrInt31* avrebbe un valore diverso da zero. Se così non è, si possono esaminare i valori che l'esecuzione dell'interrupt 2Fh ha attribuito ai flag ed al registro AL, mediante i corrispondenti campi della variabile *RMCS*. Il constructor prosegue poi analogamente a quanto accadeva nella versione per il modo reale, chiamando *Fail* in caso di errore o inizializzando la variabile *Queue* se non vi sono errori e lo spooler del DOS risulta installato.

L'implementazione dei metodi *CancelAll* e *ReleaseHold* adotta la stessa impostazione per chiamare le sotto-funzioni 03h e 05h della funzione 01h dell'interrupt 2Fh; in caso di errore, viene assegnato alla variabile *Err* il valore –31h se si tratta di un errore dell'INT 31h, altrimenti il codice d'errore reso dall'interrupt 2Fh nel registro AX.

Il metodo *HoldForStatus* richiede invece qualche accorgimento in più. La sotto-funzione 04h dell'interrupt 2Fh, infatti, ritorna nei registri DS:SI l'indirizzo del primo nome di file presente in una lista di stringhe ASCIIZ di 64 byte ognuna, contenente i nomi dei file accodati per la stampa. Quell'indirizzo verrà ora reso nei campi *DS* e *ESI* della variabile *RMCS*, ma, mentre nel campo *ESI* troveremo un normale offset, nel campo *DS* avremo un valore interpretabile come segmento solo nel modo reale.

Per poter accedere alla lista dei file in coda di stampa, quindi, avremo bisogno del selettore di un descrittore che abbia il valore reso nel campo *DS* come base. A ciò provvede la funzione 0002h dell'INT 31h, che, posto un segmento del modo reale in BX, ritorna in AX il selettore utilizzabile per accedere a questo in modo protetto. La variabile *P*, utilizzata per accedere al primo nome di file della lista, può venire così inizializzata con tale selettore e con l'offset già contenuto nel campo *ESI* della variabile *RMCS*.

È questo un primo esempio di condivisione di una stessa area di memoria tra modo reale e modo protetto: dato il suo indirizzo nel modo reale, la funzione 0002h dell'INT 31h consente di ottenere il corrispondente indirizzo nel modo protetto.

Il mese prossimo dovremo affrontare il problema inverso: convertire un indirizzo del modo protetto in un indirizzo del modo reale, per comunicare allo spooler il nome del file che vogliamo stampare o rimuovere dalla coda di stampa. La soluzione non sarà così immediata, in quanto, mentre ogni indirizzo «reale» può essere convertito in indirizzo «protetto», l'inverso non è vero; nel modo protetto, infatti, soprattutto se l'implementazione DPMI in cui si opera supporta la memoria virtuale, si ha accesso a locazioni di memoria che potrebbero ben essere sopra quel primo megabyte accessibile in modo reale. Il programma che usa la unit PROSPOOL potrebbe operare in una zona di memoria tale che la stringa contenente il nome del file da stampare o da rimuovere potrebbe avere un indirizzo fisico irraggiungibile nel modo reale. Vedremo che sarà necessario copiare quella stringa in una zona di memoria allocata nel primo megabyte. Già conoscete, dal numero di gennaio, le funzioni dell'INT 31h che dovremo usare. Se qualcuno vorrà provare, tra trenta giorni potremo confrontare le soluzioni. MC

Sergio Polini è raggiungibile tramite MC-link alla casella MC1166 e tramite Internet all'indirizzo MC1166@mclink.it.