coordinamento di Andrea de Prisco

Programmare in C su Amiga

di Dario de Iudicibus

quarta puntata

EXEC è forse il componente più importante del sistema operativo dell'Amiga. Proprio per questo un'analisi completa di EXEC non potrebbe essere effettuata in una singola puntata, anche perché richiederebbe da parte del lettore una conoscenza approfondita di alcuni concetti comuni a tutti i sistemi operativi multitasking. Tuttavia non si può pensare di proseguire questa serie senza aver mostrato almeno alcuni dei principali servizi offerti da questo componente; servizi che avremo modo di usare spesso in seguito. Abbiamo deciso quindi di dedicare due puntate ad EXEC riservandoci di approfondire l'argomento più avanti, quando affronteremo la gestione dell'ambiente multitasking di Amiga

Prima di affrontare EXEC, diamo un'occhiata alla soluzione dell'esercizio proposto nella terza puntata. Essendo i lettori ormai sicuramente esperti di AmigaDOS ed essendo il codice ampiamente commentato, non riteniamo necessario insistere oltre sull'argomento. Se qualcosa non vi torna non preoccupatevi: andatevi a rileggere la scorsa puntata e, se lo avete, anche il manuale dell'AmigaDOS. Non arrendetevi al primo tentativo e ricordate che esiste un solo modo per imparare veramente a programmare: provate e riprovate con pazienza a far girare i vostri programmi sul computer. Non limitatevi a copiare il codice dagli esempi. Fate delle prove, modificatelo! Non abbiate paura. Nel peggiore dei casi potreste rovinare il dischetto di prova ed essere costretti a ricaricare il Kickstart, ma quando sarete riusciti a far girare un programma tutto vostro la soddisfazione vi ripagherà dei tanti tentativi fatti.

Introduzione

Dire che questa puntata è dedicata ad EXEC è come dire che è possibile riassumere la Bibbia in dieci righe. Non per niente buona parte dei *ROM Kernel Manuals (RKMs)* descrivono questo importantissimo componente del sistema operativo dell'Amiga. Essendo un componente «vicino» all'hardware il suo utilizzo non è così intuitivo e di facile apprendimento come può esserlo quello dell'AmigaDOS o di Intuition. Data infatti la complessità intrinseca di un sistema multitasking, per affrontare EXEC è necessaria una certa conoscenza dei principi che stanno alla base di un sistema operativo di questo tipo.

D'altra parte, proprio per l'importanza che EXEC riveste nella gestione dell'Amiga, non è possibile evitare di analizzare almeno alcune delle principali caratteristiche di tale componente.

Decidere cosa riportare in queste due puntate e cosa no non è stato semplice. In effetti la complessità dell'argomento, se si volesse andare a fondo, è tale che sarebbero necessarie varie puntate per analizzarlo esaurientemente. Inoltre non si può non tener conto del fatto che chi ha scelto l'Amiga, spesso lo ha fatto per le sue straordinarie capacità grafiche e musicali,e non ci stupiremmo se molti lettori stessero aspettando con ansia le puntate che riguardano Intuition e la grafica.

Per questo motivo, invece di iniziare una serie complessa e per qualcuno forse un po' noiosa sui segreti del multitasking, descriveremo in breve alcune funzionalità di EXEC che avremo modo di utilizzare spesso più avanti proprio nei programmi che riguardano Intuition e le funzioni grafiche, rimandando eventualmente ad un altro momento l'analisi più dettagliata di EXEC.

Come abbiamo già spiegato nella prima puntata, EXEC è formato da un insieme di procedure *[routine]* che controllano l'utilizzo del M68000. La principale responsabilità di EXEC è quella di gestire l'ambiente multitasking, di coordinare cioè l'esecuzione dei vari lavori *[task]* che girano contemporaneamente nell'Amiga assegnando ad ognuno di essi una parte del tempo di utilizzo del 68000, gestire la memoria utilizzata da tali lavori, gli *interrupt* che vengono generati sia da alcuni componenti hardware del sistema, sia dal software applicativo o di base, ed infine di mantenere traccia di vari eventi quali messaggi, segnali dal mouse o dalla tastiera e via dicendo.

Per fare tutto ciò EXEC utilizza una

Note

1. Il termine nodo ha due significati: da un lato esso indica la struttura di aggancio che ci permette di costruire la catena, dall'altro esso si riferisce al contenuto informativo del singolo elemento della catena stessa. Qualora il significato non dovesse essere chiaro dal contesto, specificheremo esplicitamente quale delle due definizioni debba essere utilizzata.

2. In effetti il sistema alloca memoria allineandola sempre ad indirizzi che corrispondono a multipli pari a una doppia voce (otto byte). Inoltre gli stessi blocchi di memoria allocati hanno sempre dimensioni pari a multipli di una doppia voce. Ad esempio, se vengono richiesti 139 byte, il sistema allocherà 144 byte e li allineerà, mettiamo, all'indirizzo **0x2CF800**.

tecnica chiamata «delle liste» *[lists]*, come vedremo tra poco. Il secondo argomento di questa puntata riguarda la gestione della memoria, ovverosia come chiedere (allocare) memoria al sistema e come restituirla (rilasciare).

Liste e nodi

Una lista è una struttura dinamica in cui ogni elemento è legato al successivo ed a quello che lo precede tramite puntatori. Possiamo immaginarle come catene in cui ogni elemento è una struttura che contiene informazioni relative ad un lavoro, un evento in ingresso, un messaggio e così via. Ogni singolo elemento è identificato da un nodo.

Ogni lista si basa su due strutture fondamentali:
1. la testa *[header]* o struttura di ancoraggio *[anchor]*,
2. ed il nodo *[node]*.

La struttura di testa contiene i puntatori al primo ed ultimo nodo della lista ed il tipo dei nodi nella catena. Di fatto essa rappresenta la lista stessa e quindi, ogni qualvolta si vorrà far riferimento ad una lista, utilizzeremo l'indirizzo di tale struttura. La figura 3 contiene la definizione di una lista, cioè, come appena spiegato, della testata. Come si può vedere, questa contiene tre puntatori a strutture nodo:
1. **lh_Head** punta al primo nodo in lista,
2. **lh_TailPred** punta all'ultimo nodo in lista, mentre
3. **lh_Tail** punta «concettualmente» alla coda della lista *[tail]*.

```
/*
 *  Creato il 30 Agosto 1987          da  Dario de Judicibus -  Roma
 *
 *  Modulo:    qsp  (Query SPace)
 *
 *  Scopo:     restituisce lo spazio disponibile su un disco
 *
 *  Sintassi:  qsp (Driver | Volume | ?)
 *
 *  Parametri: Driver            il nome di un driver (e.g. DF0: or RAM:)
 *             Volume            il nome di un volume (e.g. "Workbench 1.2:")
 *
 *  ==========================================================================
 *  @Mod |   il   | da  | descrizione
 *  -----+--------+-----+-----------------------------------------------------
 *  @001 | 870830 | DdJ | Version 1 Release 0 Modification 0
 *
 */

#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "exec/memory.h"

extern struct FileLock *Lock();

#define NOCLEAR     0
#define CLEAR       1

struct FileLock *lock;      /* Lucchetto per il volume da esaminare. */
struct InfoData *infd;      /* Puntatore alla struttura che contiene */
                            /* le informazioni necessarie.           */
main(argc,argv)
int argc;
char *argv[];
{
  int success, TotBlocks, UseBlocks, BytXBlock, FreeSpace;

 /*
  *  Controlla se il numero di parametri in ingresso è corretto
  *  o se è stato richiesto un aiuto (parametro = ?)
  */

  if (argc > 2)  /* argv[0] è il nome del programma, argv[1] il primo */
  {                                     /* parametro, e gli altri? */
    printf("Troppi parametri\n");
    ExitPgm(RETURN_ERROR,NOCLEAR);
  }
  if (argc < 2)  /* questa volta manca pure il primo parametro!      */
  {
    printf("Troppo pochi parametri\n");
    ExitPgm(RETURN_ERROR,NOCLEAR);
  }

 if (strcmp(argv[1],"?") == 0) /* OK. Esegui questo se serve aiuto.  */
 {
  printf("Scopo:    restituisce lo spazio disponibile su disco (bytes).\n");
  printf("Sintassi: %ls (Drive | Volume | ?)\n",argv[0]);
  printf("          dove... Drive    è DF0:, DF1:, RAM: e così via\n");
  printf("                  Volume   è il nome del Volume, INCLUSI i due punti\n");
  ExitPgm(RETURN_OK,NOCLEAR);
 }
 else
 {
   /*
    *  Richiedi un lucchetto in lettura sul volume specificato
    */
   lock = Lock(argv[1],ACCESS_READ);
   if (!lock)
   {
     printf("Non posso accedere %ls. Controlla la sintassi usata.\n",argv[1]);
     ExitPgm(RETURN_ERROR,NOCLEAR);
   }
 }

 /*
  *  Alloca memoria per InfoData (ricorda che devi allinearla alla voce)
  */

  infd = (struct InfoData *)AllocMem(sizeof(struct InfoData),MEMF_CLEAR);

 /*
  *  Carica le informazioni relative al volume specificato
  */

  success = Info(lock,infd);
  if (!success)                   /* Informazioni non disponibili! */
  {
     printf("Non riesco ad avere informazioni su %ls\n",argv[1]);
     ExitPgm(RETURN_ERROR,CLEAR);
  }
  if (infd->id_DiskType == -1)  /* Il disco non è inserito!      */
  {
     printf("%ls non inserito!\n",argv[1]);
     ExitPgm(RETURN_WARN,CLEAR);
  }
  else  /* OK. Trovate! Calcoliamo lo spazio libero sul volume.      */
        /* Formula usata:                                            */
        /*                                                           */
        /* <blocchi liberi> = <blocchi nel volume> - <blocchi usati> */
        /*                                                           */
        /* <byte liberi> = <blocchi liberi> * <bytes in un blocco>   */

  {
    TotBlocks = infd->id_NumBlocks;         /* Blocchi nel volume */
    UseBlocks = infd->id_NumBlocksUsed;     /* Blocchi usati      */
    BytXBlock = infd->id_BytesPerBlock;     /* Bytes in un blocco */
    FreeSpace = (TotBlocks - UseBlocks) * BytXBlock;
    printf("Lo spazio disponibile sul disco %ls è %ld Byte(s)\n",argv[1],FreeSpace);
  }

  ExitPgm(RETURN_OK,CLEAR);
}

ExitPgm(rc,flag)   /* Un altro esempio di routine per uscire in modo */
int rc, flag;      /* pulito.                                        */
{
  if (flag == CLEAR)
  {
    /*
     *  OK, sblocca il volume e libera la memoria allocata.
     */
    FreeMem(infd,sizeof(struct InfoData));
    UnLock(lock);
  }
  Exit(rc);
}
```

Figura 1 - Soluzione all'esercizio della terza puntata.

Fate attenzione a non confondere **lh_Tail** con **lh_TailPred**:
il primo è un elemento «tappo» che indica la fine della lista (analogo al NIL del Pascal) ed è quindi sempre uguale a zero, il secondo punta invece all'ultimo nodo della lista, come mostrato in figura 2.

lh_Type specifica il tipo di informazioni conenute nella lista (messaggi, interrupt, lavori, etc...). In figura 5 è mostrata una lista dei possibili nodi disponibili come definiti in exec/nodes.h.

Un nodo è invece formato di due parti:
1. una struttura di aggancio (il nodo vero e proprio),
2. il contenuto del nodo stesso.

La struttura di aggancio (vedi sempre figura 3) serve al nodo per agganciarsi sia al nodo precedente che a quello successivo nella lista (o rispettivamente alla testa ed alla coda o NIL).

Il contenuto, viceversa, può essere rappresentato da una qualsiasi struttura, compresa eventualmente un'altra lista. Di fatto, anche se la struttura di aggancio si chiama Node, il nodo, come elemento di una lista, è formato da una struttura qualunque che contiene il puntatore ad una struttura di aggancio, come mostrato nella stessa figura (vedi nota 1). In molti casi tale puntatore è il primo elemento della struttura, ma non obbligatoriamente. Se tuttavia non avete specifiche ragioni per fare differentemente, vi consigliamo di situare il nodo in cima alla struttura, come nell'esempio riportato in figura **(struct Indirizzo)**.

Una lista va opportunamente inizializzata prima di essere utilizzata, a meno che essa non sia stata già inizializzata dal sistema. Per far questo si può utilizzare la funzione **NewList()** (vedi figura 4).

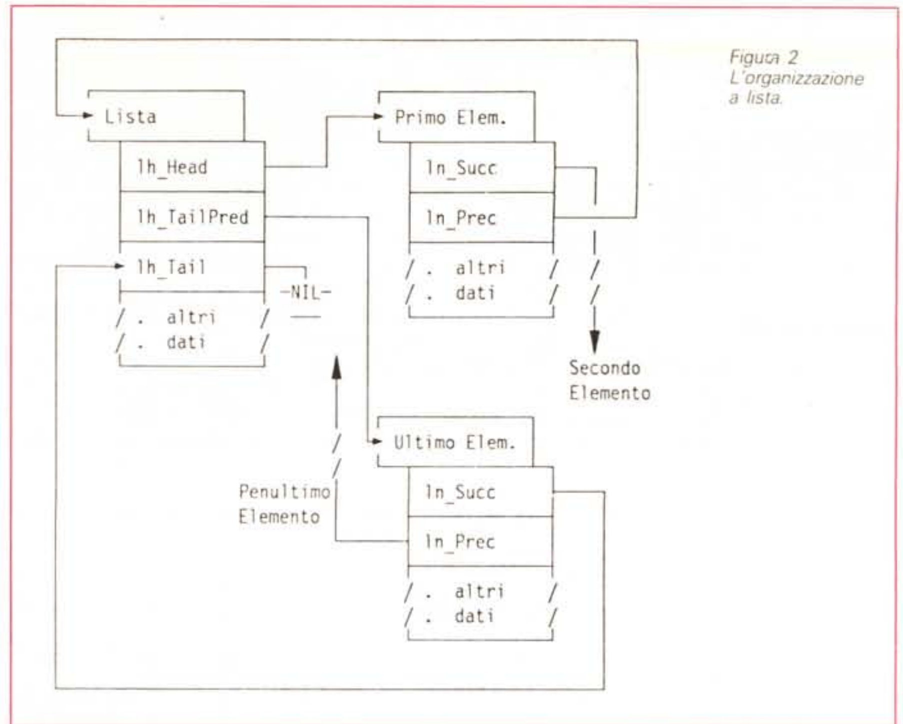


Figura 2
L'organizzazione a lista.

Figura 3
Liste e nodi.

```
/*
 *  lista o struttura di testa
 */
 struct List
 {
   struct Node *lh_Head;     /* punta al primo nodo nella lista      -   */
   struct Node *lh_Tail;     /* puntatore "tappo" (NIL): è sempre zero   */
   struct Node *lh_TailPred; /* punta all'ultimo nodo nella lista        */
   UBYTE lh_Type;            /* definisce il tipo dei nodi nella lista   */
   UBYTE lh_pad;             /* byte di allineamento: non utilizzato     */
  };

/*
 *
 */
 struct Node
 {
   struct Node *ln_Succ;     /* punta al nodo successivo nella lista     */
   struct Node *ln_Pred;     /* punta al nodo precedente nella lista     */
   UBYTE ln_Type;            /* definisce il tipo del nodo               */
   BYTE ln_Pri;              /* specifica la priorita del nodo           */
   char *ln_Name;            /* punta una stringa di caratteri che...    */
 };                          /* ...rappresenta il nome del nodo          */

/*
 *  Esempio di nodo
 */
 struct Indirizzo
 {
   struct Node ind_nodo;
   char *nome;
   char *cognome;
   char *via_nciv;
   char *telefono;
   char *cap;
   char *citta;
 };
```

Altre funzioni che possono essere utilizzate per la gestione delle liste sono mostrate in figura 4. Queste funzioni permettono di compiere diverse operazioni sugli elementi di una lista, quali

- inserire un nodo nella lista,
- rimuovere un nodo dalla lista,
- aggiungere un nodo in testa o in coda alla lista,
- cercare uno specifico nodo in un lista.

Attenzione: non usate mai la funzione **Remove()** se non siete sicuri che il nodo da rimuovere faccia effettivamente parte di una lista. Potreste causare una caduta *[crash]* del sistema.

Per ulteriori dettagli sulle liste, rimandiamo alla prima parte del primo volume dei *RKMs*.

Gestione della memoria

Quando definite una variabile od una struttura in un programma, state praticamente dicendo al programma di riservarvi una certa area di memoria da utilizzare durante l'esecuzione. A seconda della classe di memoria *[storage class]* di tali oggetti, questi possono avere una posizione fissa in memoria e durare per tutta la durata del programma, oppure essere posizionati dove c'è posto entro un dato segmento ed essere temporanei. In entambi i casi, tuttavia, la gestione della memoria è fuori dal controllo del programmatore.

Figura 4
Funzioni per la gestione delle liste.

```
/*
 *  Strutture utilizzate qui di seguito
 */
 struct List *lista;
 struct Node *nodo, *nuovonodo;
 char *nomenodo;

/*
 *  Funzioni
 */
 void NewList(lista);                  /* Inizializza una nuova lista.      */

 void Insert(lista, nuovonodo, nodo); /* Aggiungi "nuovonodo" dopo "nodo". */

 void AddHead(lista, nuovonodo);      /* Aggiungi "nuovonodo" in testa.    */
 void AddTail(lista, nuovonodo);      /* Aggiungi "nuovonodo" in coda.     */
 void Remove(nodo);                   /* Rimuovi un nodo.                  */

 struct Node *RemHead(lista);         /* Rimuovi un nodo dalla testa di    */
                                      /* una lista e ritorna l'indirizzo   */
                                      /* di tale nodo, oppure 0 se vuota.  */
 struct Node *RemTail(lista);         /* Come sopra, ma dalla coda.        */

 void Enqueue(lista, nodo);           /* Aggiungi un nodo in una lista     */
                                      /* in ordine di priorità.            */

 struct Node *FindName(lista, nomenodo); /* Ritorna il primo nodo della    */
                                         /* lista che abbia come nome      */
         /* - OPPURE - */                /* "nomenodo". La ricerca può     */
                                         /* partire dall'inizio della      */
 struct Node *FindName(nodo, nomenodo);  /* lista o da un nodo qualunque.  */
```

Se per un qualunque motivo avete bisogno di allocare memoria durante l'esecuzione di un programma dovete richiedere al sistema di riservare un certo numero di byte utilizzando appunto EXEC. Una tale esigenza potrebbe sorgere qualora abbiate bisogno di mantenere in memoria un certo numero di dati di cui non conoscete a priori la quantità. Oppure potrebbe essere necessario posizionare una certa struttura allineandola ad una voce *[word]*, cioè ad un indirizzo multiplo di quattro byte (vedi nota 2).

Attenzione: Quando chiedete al sistema di allocare una certa quantità di memoria, ricordatevi sempre

1. che la memoria allocata non viene inizializzata in alcun modo a meno di non richiederlo esplicitamente;
2. di rilasciare sempre la memoria allocata non appena non vi serva più e comunque prima di terminare l'esecuzione del programma;
3. di utilizzare il giusto tipo di memoria a seconda dell'utilizzo che non volete fare.

Vediamo in dettaglio questi punti.

Quando richiedete al sistema di allocare una certa area di memoria, questo si limiterà a riservarvi un'area che è corrispondente al multiplo superiore di MEM_BLOCKSIZE (attualmente 8 byte) più vicino al numero di byte da voi richiesti. Dato che il sistema tiene traccia solo della memoria ancora disponibile nella macchina utilizzando una lista dei blocchi di memoria liberi, esso non ha la minima idea di chi ha allocato memoria, quando e perché. Se non liberate la memoria prima di terminare il task che l'ha richiesta, quei blocchi non saranno più disponibili fino alla ripartenza del sistema *[reboot]*. Se poi l'area di memoria allocata ha dimensioni notevoli, come nel caso di quella necessaria a certe operazioni di grafica, allora è bene liberarla non appena possibile, in modo da permettere ad altri lavori di effettuare a loro volta richieste di allocazione.

Figura 5
Tipi di nodo.

```
/*----------------- Vari tipi di nodo ----------------*/

#define NT_UNKNOWN      0    /* tipo sconosciuto        */
#define NT_TASK         1    /* lavoro                  */
#define NT_INTERRUPT    2    /* interruzione            */
#define NT_DEVICE       3    /* dispositivo             */
#define NT_MSGPORT      4    /* porta                   */
#define NT_MESSAGE      5    /* messaggio               */
#define NT_FREEMSG      6    /* messaggio libero        */
#define NT_REPLYMSG     7    /* messaggio di risposta */
#define NT_RESOURCE     8    /* risorsa                 */
#define NT_LIBRARY      9    /* libreria                */
#define NT_MEMORY       10   /* memoria                 */
#define NT_SOFTINT      11   /* interruzione software */
#define NT_FONT         12   /* font                    */
#define NT_PROCESS      13   /* processo                */
#define NT_SEMAPHORE    14   /* semaforo                */
#define NT_SIGNALSEM    15   /* segnale semaforo        */
#define NT_BOOTNODE     16   /* nodo di BootStrap       */
```

Un altro aspetto importante è quello che riguarda il posizionamento dei blocchi di memoria. Come forse saprete, la memoria dell'Amiga è divisa in due parti:

- memoria CHIP (256K da **0x000000** ad **0x03FFFF** oppure 512K da **0x000000** ad **0x07FFFF**),
- memoria FAST (fino ad 8M da **0x200000** ad **0x9FFFFF**).

La memoria CHIP si chiama così perché è l'unica che può essere indirizzata da tutti e quattro i processori del sistema, sia cioè dal Motorola 68000, che dagli speciali coprocessori che si occupano di gestire i dispositivi video ed audio. Viceversa la memoria FAST (veloce) deve il suo nome al fatto che il processore centrale non deve condividerla con nessuno degli speciali coprocessori menzionati, e quindi l'accesso ai dati è più rapido.

Nella prima vanno quindi tutti quei dati che devono essere disponibili ai coprocessori video/audio, mentre nella seconda possono andare tutti gli altri dati, programmi compresi. In passato, quando non erano ancora disponibili le espansioni di memoria per l'Amiga, molti programmatori si limitavano a chiedere memoria al sistema senza specificare dove questa andasse posizionata. In questi casi l'Amiga cerca prima di allocare memoria di tipo FAST e, qualora non sia possibile, fornisce al programma memoria di tipo CHIP. Fintanto che tali programmi giravano in macchine senza espansione tutto funzionava regolarmente. Nel momento però in cui il programma veniva eseguito in una macchina dotata di espansione di memoria, poteva capitare di ricevere la visita del tristemente famoso messaggio di «Guru Meditation». Questo è dovuto al fatto che il sistema allocava come FAST anche quelle strutture che vanno assolutamente posizionate nella memoria CHIP.

La caduta del sistema era quiandi dovuta all'impossibilità da parte dei tre coprocessori speciali di indirizzare le strutture necessarie.

Dato che oggi moltissimi utenti hanno espansioni di memoria, fate molta attenzione quando allocate memoria. In questi casi è bene seguire la seguente logica.

```
/*
 *  Allocazione e deallocazione di due blocchi di memoria
 */
#include "exec/types.h"
#include "exec/memory.h"
#include "intuition/intuition.h"
#include "graphics/sprite.h"

UBYIE mask = 0x00;
#define F_MEM 0x01
#define S_MEM 0x02
#define xF_MEM 0x04

typedef struct FileInfoBlock FIB;
FIB *fib;

/*
 *  Un piccolo regalo per gli amanti degli sprites: questa è la famosa
 *  sfera a scacchi di Amiga! Sono 72 bytes. Questo è il prototipo.
 */
#define SPRITESIZE 72
UWORD sfera[] =
{
  0,0,
  0x07E0, 0x0320,
  0x1FF8, 0x0CC8,
  0x3FFC, 0x18E4,
  0x7FFE, 0x4E18,
  0x7FFE, 0x5E1C,
  0xFFFF, 0x9E0E,
  0xFFFF, 0xE1F1,
  0xFFFF, 0xC1F1,
  0xFFFF, 0xC1F1,
  0xFFFF, 0xE1F1,
  0xFFFF, 0x9E0E,
  0x7FFE, 0x5E1C,
  0x7FFE, 0x4E18,
  0x3FFC, 0x18E4,
  0x1FF8, 0x0CC8,
  0x07E0, 0x0320,
  0,0
};
UWORD *from, *to, *s;

VOID CloseAll()
{
  if (mask & F_MEM) FreeMem(fib,sizeof(FIB));
  if (mask & S_MEM) FreeMem(s,SPRITESIZE);
}

VOID main()
{
  UBYTE i;

  /*
   *  Alloca memoria per un blocco di informazioni su di un file.
   *  >>> Va bene sia FAST che CHIP, ma azzera prima.
   */
  if ( (fib = (FIB *)AllocMem(sizeof(FIB),MEMF_CLEAR)) == 0 ) CloseAll();
  mask |= F_MEM; /* OK, allocata */

  /*
   *  Alloca memoria per uno sprite a forma di sfera a scacchi.
   *  >>> Deve essere in CHIP. Non azzerare: inizializzata da programma.
   */
  if ( (s = (UWORD *)AllocMem(SPRITESIZE,MEMF_CHIP)) == 0 ) CloseAll();
  mask |= S_MEM; /* OK, allocata */
  from = sfera; to = s;
  for (i=0; i<(SPRITESIZE/2); i++) *to++ = *from++; /* copia in CHIP */

  DoWhatYouWant(); /* Adesso fa quello che vuoi... */

  CloseAll();        /* Chiudi tutto! */

}
```

Figura 6 - Allocazione e deallocazione di memoria.

La memoria da richiedere deve contenere dati accessibili ai tre coprocessori speciali?
Se SI allora bisogna esplicitamente richiedere memoria CHIP;
se NO allora è bene lasciare al sistema la scelta più opportuna.

Come si può vedere dallo schema riportato, è bene evitare di richiedere esclusivamente memoria di tipo FAST, a meno che non sia estremamente importante la velocità di accesso ai dati oppure ci si voglia assicurare la massima quantità disponibile di memoria CHIP.

In tal caso però il programma non potrà girare su macchine che non abbiano espansioni di memoria.

Alcuni prodotti software di grafica ed animazione rientrano in quest'ultimo caso.

Le seguenti quattro costanti predefinite in **exec/memory.h** possono venire utilizzate per specificare particolari esigenze quando si richiede memoria al sistema (specifiche di allocazione):
MEMF_CLEAR riempi il blocco di memoria richiesto con zeri (0x00) prima di restituire al programma il puntatore relativo;
MEMF_CHIP dammi solo memoria di tipo CHIP;
MEMF_FAST dammi solo memoria di tipo FAST;
MEMF_PUBLIC dammi memoria accessibile da qualunque task.

A queste va aggiunto il valore zero (0) che significa: «dammi memoria in qualunque area, provando prima con quella FAST e poi, eventualmente, con quella CHIP».

La cosiddetta memoria pubblica (MEMF_PUBLIC) è stata introdotta in previsione di una possibile gestione della memoria virtuale in un rilascio futuro del sistema operativo.

Lo scopo è quello di identificare aree di memoria disponibili e condivisibili da più lavori, piuttosto che legate al singolo task.

EXEC fornisce al programmatore diversi modi di allocare e deallocare memoria.

Noi vedremo solo il più semplice, per il momento. Esso fa uso delle funzioni **AllocMem()** e **FreeMem()** come mostrato in figura 6. Notare che, dato che l'arrotondamento effettuato dal sistema sul valore specificato dal programmatore relativamente alle dimensioni del blocco di memoria viene effettuato sia all'allocazione che alla deallocazione, **FreeMem()** accetta in ingresso lo stesso numero di byte specificato nella **AllocMem()**.

Come si può vedere in figura, per allocare memoria è necessario specificare il numero di byte e le specifiche di allocazione. La funzione restituisce il puntatore al blocco riservato oppure zero (0) se non c'è abbastanza memoria libera.

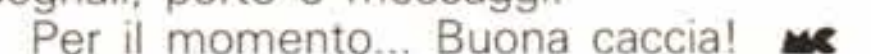
Analogamente la deallocazione si ottiene specificando in ingresso il puntatore ottenuto in precedenza e le dimensioni del blocco relativo in byte.

L'esercizio

Vediamo se siete dei lettori attenti. Nella scorsa puntata è stato volutamente introdotto un piccolo errore nel codice in figura 2 (quello che ricavava la data dal file **S:Oggi**, tanto per intenderci). Ve ne siete accorti? No? Allora andate a darci un'occhiata e la prossima volta occhi aperti.

Nella prossima puntata parleremo di segnali, porte e messaggi.

Per il momento... Buona caccia!



SOFTWARE

software originale, sigillato, con garanzia ufficiale e possibilità di aggiornamento

SPREADSHEET/INTEGRATI	
Microsoft Excel 2.0 MS-DOS (It.)	**790.000**
Microsoft Multiplan 3.0 (It.)	390.000
Microsoft Multiplan 2.0 XENIX	429.000
Microsoft Works (It.)	**350.000**
Lotus 1-2-3 Rel 2.01 (It.)	690.000
Lotus Symphony 2.0 (It.)	950.000
Computer Associates Supercalc 4.0 (It.)	680.000
Software Group Enable (It.)	1.090.000
Borland Quattro	**350.000**

IL SOFTWARE ORIGINALE NON HA IL VIRUS

Microsoft Excel Macintosh (It.)	690.000
Microsoft Works Macintosh (It.)	450.000
Lotus Jazz Macintosh (It.)	560.000

WORD PROCESSING	
Microsoft Word 4.0 (It.)	**750.000**
Microsoft Word 3.0 LAN - 5 users (It.)	1.990.000
Microsoft Word 3.0 XENIX	890.000
Lotus Manuscript (It.)	**690.000**
MicroPro WordStar 4.0 (It.)	595.000
MicroPro WordStar 2000 Rel. 3.0 (It.)	890.000
Ashton-Tate Multimate Advantage II	790.000
Samna Word IV Plus (It.)	**1.250.000**
Satellite Word Perfect	1.190.000
Microsoft Word 3.0 Macintosh (It.)	690.000
Microsoft Word 3.0 LAN Macintosh 5 users (It.)	1.490.000

HARDCARD PLUS
Hard Disk su scheda
35 ms. tempo medio di accesso
Versione 20 MB L.1.190.000
Versione 40 MB L.1.590.000

DATABASE MANAGEMENT	
Microsoft RBase System (It.)	**1.090.000**
Ashton-Tate Rapid File (It.)	590.000
Borland Paradox	**1.190.000**
Borland Reflex (It.)	290.000
Nantucket Clipper	1.190.000
Wordtech Quicksilver	1.090.000
Microsoft File Macintosh	295.000
Ashton-Tate dBASE MAC Macintosh	630.000

GRAFICI	
Microsoft Chart 2.0 (It.)	395.000
Microsoft Chart 3.0	590.000
Lotus Freelance Plus (It.)	**690.000**
Autodesk AutoCAD Base (It.)	690.000
Autodesk AutoCAD ADE 2 (It.)	4.360.000
Autodesk AutoCAD ADE 3 (It.)	5.950.000

SPEDIZIONI GRATUITE IN TUTTA ITALIA

Ashton-Tate ChartMaster	730.000
Microsoft Chart Macintosh (It.)	280.000

DESKTOP PUBLISHING	
Aldus PageMaker (It.)	**1.150.000**
RANK XEROX Ventura Publisher (It.)	**1.390.000**
Microsoft Pageview	90.000
Letraset Ready, Set, Go! 4.0 Macintosh (It.)	890.000

COMUNICAZIONI	
Microsoft Access	390.000
DCA Crosstalk XVI	**420.000**
DCA Crosstalk Mark 4	430.000
Meridian Carbon Copy Plus	490.000

SPECIALE SOFTWARE

dBASE III Plus (It.)
(include un buono per aggiornamento a dBASE IV al prezzo di L. 200.000)
L. 1.090.000

Framework II (It.)
(include un buono per aggiornamento a Framework III al prezzo di L. 100.000)
L. 1.090.000

Computer Discount Italia®
è diventata

Quotha32

rivenditori indipendenti di hardware e software

NOVITA'!
Stampanti NEC 24 aghi in STOCK
P6 Plus, P7 Plus e P2200
con garanzia ITALIANA
TELEFONARE

LINGUAGGI	
Microsoft QuickBASIC 4.0	**175.000**
Microsoft QuickC	195.000
Microsoft BASIC Interpreter	590.000
Microsoft BASIC Compiler	460.000
Microsoft C Compiler	**595.000**
Microsoft FORTRAN Compiler	595.000

CONDIZIONI AGEVOLATE PER ENTI PUBBLICI E SCUOLE
richiedete i nostri preventivi!

Microsoft COBOL Compiler	990.000
Microsoft Pascal Compiler	390.000
Ryan MacFarland COBOL Full System	1.390.000
Lattice C Compiler	690.000
Borland Turbo Pascal 4.0 (It.)	250.000
Microsoft BASIC Interpreter XENIX 286	495.000
Microsoft BASIC Compiler XENIX 286	**990.000**
Microsoft COBOL Compiler XENIX 286	1.390.000
Microsoft FORTRAN Compiler XENIX 286	990.000
Microsoft Pascal Compiler XENIX 286	990.000
Microsoft XENIX 386 Toolkit	550.000
Microsoft BASIC Interpreter Macintosh	195.000
Microsoft BASIC Compiler Macintosh	290.000
Microsoft FORTRAN Compiler Macintosh	450.000

UTILITIES	
Microsoft Windows 2 (It.)	**216.000**
Microsoft Windows 386 (It.)	**370.000**
Microsoft Windows 2 Toolkit	690.000
Norton Utilities 4.0	**280.000**
Norton Commander	170.000
Fastback	495.000
Software specifico/gestionale	Telefonare

RICHIEDETE PREVENTIVI PER PC/XT, AT COMPATIBILI e 386

HARDWARE

hardware originale con garanzia ITALIANA di 1 anno

PERSONAL COMPUTER	
Olivetti M15, 2 FDU 720 KB, 512 KB RAM, alimentatore, cavi, borsa, manuali, MS-DOS	1.295.000
Olivetti M24, 1 HDU 20 MB, 1 FDU 360 KB, 640 KB RAM, completo di monitor e tastiera	2.650.000
Olivetti M240, 1 HDU 20 MB, 1 FDU 360 KB, 640 KB RAM, completo di monitor e tastiera	3.150.000
Olivetti M28, 1 HDU 20 MB, 1 FDU 1.2 MB, 512 KB RAM, completo di monitor e tastiera	3.850.000
Olivetti M290, 1 HDU 20 MB, 1 FDU 1.2, 2 MB RAM, completo di monitor, tastiera e MS-DOS	**5.790.000**
Olivetti Altri Modelli	Telefonare
PC Philips	**Telefonare**
PC Bit Computers	Telefonare
Portatili ZENITH	Telefonare

MONITOR	
Monitor NEC MultiSync II 14" colore 800x600	**1.290.000**
Monitor NEC MultiSync Plus 15" colore 960x720	1.890.000

Monitor NEC MultiSync II
+ scheda VEGA VGA
L. 1.890.000

Monitor NEC MultiSync XL 20" colore 1.024x768	4.490.000
Monitor Hantarex Boxer 14" CGA/Herc.	220.000
Monitor Colore Hantarex CT 9000 SHR EGA/CGA 14"	690.000
Sistema grafico WYSE 700 15" CRT TTL, monocr. fosfori bianchi scheda grafica ris. max. 1280x800 compatibile CGA e MGA base basculante	1.490.000

STAMPANTI/PLOTTER	
Stampante PANASONIC KX-P1081, 80 col., 120 c.p.s.	495.000
Stampante PANASONIC KX-P1592, 136 col., 180/270 c.p.s.	950.000
Stampante NEC P7, 136 col., 24 aghi, 216 c.p.s. con trascinatore monodirezionale	**1.495.000**
Stampante NEC P9XL a colori, 136 col., 24 aghi, 400 c.p.s.	**2.990.000**
Altri Modelli PANASONIC in STOCK	Telefonare
Plotter Roland DXY 880/A	1.750.000
Stampanti OKI	Telefonare

Hard Disk Seagate ST225, 20MB
completo di controller WD e cavi
l'hard disk più venduto del mondo
L. 495.000

DISK DRIVE - STREAMER	
Floppy Disk Drive da 3.5" 720 KB completo di kit per alloggiamento e cavi	270.000
Floppy Disk Drive da 3.5" 1.44 MB completo di kit per alloggiamento e cavi	**320.000**
Hard Disk NEC D5126H 20 MB "veloce" (37 ms.)	**850.000**
Hard Disk NEC D5146H 40 MB "veloce" (37 ms.)	1.090.000
Streamer e sistemi di backup	Telefonare

SCHEDE - CHIP - MOUSE	
Coprocessore matematico Intel 8087-1 10 MHz	470.000
Coprocessore matematico Intel 80287-10 10 MHz	650.000
Coprocessore matematico Intel 80387 25 MHz	**Telefonare**
Scheda Microsoft MACH 20	Telefonare
Scheda Grafica VEGA VGA	**750.000**
Microsoft Mouse 7 (NUOVO!)	**280.000**
Scheda Telefax LEXICON LEXIFAX	1.150.000
Schede INBOARD 386 Intel per PC/XT e AT	Telefonare
Schede Varie	Telefonare

VARIE	
Modem Discovery 1200H su scheda, V21/V22, 300-1200 b.p.s., set esteso HAYES	330.000
Modem Discovery 2400C esterno, V22/V22B, 1200-2400 b.p.s., set esteso HAYES	750.000
Scanner ottico LEXICON LEXISCAN	650.000

DISCHETTI

Formato 3 1/2"	
Micro Mito DS/DD	2.600
Sony DS/DD	**3.500**
Verbatim DS/DD	**4.600**
3M DS/DD	6.900

Formato 5 1/4"	
Duratech DS/DD	1.050
Verbatim DS/DD	**2.700**
3M DD/DS	3.500
Mito-Mega 96TPIDS/HD 1.2 MB	3.100
Verbatim DS/HD 1.2 MB	5.200
3M DS/HD	**5.900**

ordine minimo 100 pezzi

Tutti i prezzi sono al netto di I.V.A.

TERMNI E CONDIZIONI DI VENDITA - Tutti i prezzi sono al netto di I.V.A. - Spese di spedizione a carico della Quotha 32 s.r.l. - Pagamento in contrassegno con assegno circolare intestato a Quotha 32 s.r.l o contante - Sconto del 2% per pagamento anticipato. - Ci riserviamo di accettare ordini di importo inferiore a 300.000 lire. - La merce si intende salvo il venduto. - Ulteriori sconti per quantità. - La presente offerta è valida sino al 15 ottobre 1988 e sostituisce ogni nostra precedente offerta

per ordini, informazioni o richieste di listini completi telefonare allo

055 - 22.99.851

oppure scrivere, precisando il recapito telefonico, a

Quotha32 s.r.l.
Via Accursio, 2 — 50125 FIRENZE — Telefax 055-2280674