coordinamento di Andrea de Prisco

# Programmare in C su Amiga

di Dario de Judicibus

*Una delle caratteristiche fondamentali di un sistema multitasking è la possibilità da parte di un task di comunicare con altri lavori, siano essi «attivi», siano essi «addormentati».*
*Tale comunicazione tuttavia, necessita di un controllore, di un componente del sistema operativo cioè, che si assume il compito di fornire i mezzi necessari e di controllarne il corretto utilizzo. Questa è appunto un'altra delle «responsabilità» di EXEC, di cui abbiamo già visto alcune funzionalità nella scorsa puntata*

Prima di passare a descrivere le tecniche di comunicazione tra task che EXEC ci mette a disposizione, vediamo brevemente la soluzione dell'esercizio proposto nella quarta puntata.

Come ricorderete, più che di un esercizio, si trattava di una piccola prova per verificare se avete acquisito uno dei concetti fondamentali del programmatore modello e, allo stesso tempo, per vedere se siete dei lettori attenti e pronti nel trovare errori nel codice (altra virtù essenziale, direi vitale, di un programmatore).

Un principio base nello sviluppo di un programma è quello di «chiudere sempre le parentesi». Il nome deriva dal classico errore che spesso si fa quando si scrive una espressione algebrica con più livelli di parentesi: basta dimenticarsene una, *et voilà*, il programma dà errore. In realtà tale principio è molto più generale, ed è in questa forma più universale, appunto, che è stato violato dal programmino presentato nella terza puntata.

«Chiudere sempre le parentesi» vuol dire infatti

- ricordarsi di chiudere tutto ciò che si è aperto, siano esse parentesi in una espressione, siano essi file;
- sbloccare tutto ciò che si è bloccato: lucchetti, File Handle, messaggi...;
- rilasciare eventuale memoria allocata;
- cancellare file temporanei e di lavoro;
- ripristinare le caratteristiche dell'ambiente *[environment configuration]* se erano state modificate;

rimettere tutto in ordine come lo si era trovato, insomma!

Non seguire questa semplice regola, vuol dire non solo degradare il sistema (ad esempio riducendo la memoria disponibile), ma anche impedire ad altri lavori di portare a compimento la loro attività.

Non rispondere ad un messaggio, ad esempio, può bloccare un lavoro fino a che non si riavvii il sistema. Analogamente, non liberare un lucchetto, impedisce l'accesso al file od alla directory associata da parte di un altro lavoro o del sistema operativo stesso.

Vediamo dunque quale di questi errori era stato introdotto nel programma in questione, riportato in figura 1.

Il codice sotto accusa è stato riportato due volte in figura: nel riquadro superiore c'è il codice «sbagliato», mentre in

*Figura 1 Soluzione all'esercizio della quarta puntata.*

```
/*************************************************************************\
*************** Questo è il programma "incriminato" ********************
\*************************************************************************/

/*                                                                       *\
 *  DATA       -    Aggiorna la data di sistema prendendola dal         *
 *                  file "Oggi".                                        *
\*                                                                       */

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

 VOID main()
 {
   struct FileHandle *infh;
   BOOL Success;

   infh = Open("S:Oggi",MODE_OLDFILE);
   if (infh == NULL) Exit(RETURN_FAIL);
/------------------------------------------ Versione originale ------\
|  Success = Execute("Date ?",infh,0);                               |
|  if (!Success) Exit(RETURN_FAIL);    <===== L'ERRORE E' QUI! =====  |
|  Close(infh);                                                      |
|---------------------------------------- Versione corretta ---------|
|  Success = Execute("Date ?",infh,0);                               |
|  Close(infh);                                                      |
|  if (!Success) Exit(RETURN_FAIL);                                  |
\--------------------------------------------------------------------/
 }
```

quello inferiore è mostrato quello corretto. L'errore consiste nel fatto che, se l'esecuzione del comando **Date** tramite la **Execute()** non fosse andata a buon termine, il programma sarebbe uscito senza prima chiudere il file **S:Oggi.** Nella versione corretta, invece, il file che contiene le informazioni relative alla data ed ora di sistema viene comunque subito chiuso, e solo dopo, se necessario, viene riportata la condizione di errore.

In realtà, in questo particolare caso, potrebbe succedere che, qualora si verificasse una condizione di errore nella **Execute()**, il file in questione verrebbe comunque chiuso. Il trucco è nella **Exit().** La **Exit()** con la «E» maiuscola è una funzione di AmigaDOS. Quando viene invocata, essa, come riporta il manuale dell'AmigaDOS, si comporta nel seguente modo:

- se il programma era stato lanciato da CLI, **Exit()** ritorna il controllo a quest'ultimo passandogli il codice di ritorno del programma stesso (nel nostro caso RETURN_FAIL);
- se viceversa questi era stato lanciato come un processo indipendente, **Exit()** cancella tale processo e rilascia la memoria allocata per lo stack, la lista dei segmenti e la struttura che contiene le informazioni sul processo stesso (non preoccupatevi se non conoscete alcuni di questi termini: per ora ignorateli, li vedremo molto più avanti).

Nel *Lattice C* invece, tanto per fare un esempio, esiste una funzione **exit()** (con la «e» minuscola) che, se invocata, si preoccupa di chiudere anche eventuali file lasciati aperti dopo aver scritto eventuali buffer di uscita rimasti perdenti. Se fosse stata usata questa funzione, il programma avrebbe funzionato regolarmente in ogni caso.

«Bene!» — direte voi — «Allora usiamo la **exit()** invece della **Exit().**».

Sbagliato!

Innanzi tutto perché la **exit()** chiude solo i file aperti con le funzioni del *Lattice C* **open, creat** e **creatx**, lasciando aperti quelli aperti via AmigaDOS, sia che siano state usate le funzioni **dopen, dcreat** e **dcreatx** del *Lattice C*, sia che siano state invocate direttamente quelle di AmigaDOS, come nel caso in esame. In secondo luogo perché abituarsi ad ignorare la regola delle «parentesi chiuse», anche quando ci pensa qualcun altro a risistemare le cose, è indice di cattiva programmazione e porta inevitabilmente, prima o poi, a commettere degli errori.

## Introduzione

Torniamo ora ad EXEC. Come abbiamo già detto, una delle principali responsabilità di questo componente riguarda la gestione dei task. Dato che in un sistema multitasking è fondamentale che un lavoro sia in grado di comunicare vuoi con i processi del sistema operativo, vuoi con altri lavori EXEC deve essere in grado di fornire i mezzi affinché tale comunicazione avvenga. Non solo: esso deve anche fornire delle regole di comunicazione (protocollo) in modo da evitare che si generi confusione durante lo scambio delle informazioni.

Comunicare, in realtà, vuol dire molte cose. Si parla di comunicazione quando ci si scambiano informazioni (ad esempio per telefono), ma anche il cambiare colore di un semaforo è una forma di comunicazione; una comunicazione può avvenire in una sola direzione, come nel caso di un televisore, od in entrambe le direzioni, come durante una discussione; essa può essere privata, tra due sole entità, o condivisa, come durante un dibattito. In ogni caso esistono regole ben precise, anche se differenti a seconda del tipo di comunicazione, sul modo in cui essa deve avvenire. Mai capitato che qualcuno vi telefonasse e, invece di presentarsi e verificare di aver telefonato al numero giusto, chiedere «Carlo? Hey Carlo!». Beh, quel tizio sarebbe probabilmente un pessimo programmatore, almeno se utilizzasse la stessa tecnica anche nella comunicazione tra task.

Anche nell'Amiga esistono varie tecniche di comunicazione: segnali, semafori, interruzioni, messaggi, e via dicendo. Ogni tecnica ha uno scopo diverso, vantaggi e svantaggi. Noi ne vedremo rapidamente solo alcune che avremo modo di usare in seguito. Torneremo eventualmente su EXEC più avanti, se l'interesse mostrato per questa serie di articoli giustificherà lo sforzo.

Per ora ci occuperemo di:

1. segnali
2. porte e messaggi.

## Segnali

Un segnale è, come dice la parola, un indicatore che un certo evento è occorso. Il suonare di un allarme o l'accendersi di una spia luminosa, sono esempi di segnali con cui abbiamo spesso a che fare. Nel caso dell'Amiga, un segnale è rappresentato dal mettere ad 1 un particolare bit in una doppia voce *[long word]* (32 bit). Nella struttura che definisse un task (vedi figura 2) ci sono alcuni campi che mantengono informazioni relative ai

```
/*************************************************************\
 *  Questa è la struttura relativa ad un lavoro nel sistema.  *
 *  Per ora ci limiteremo ad evidenziare i campi relativi ad  *
 *  i segnali associati ad un certo lavoro.                   *
\*************************************************************/

extern struct Task {
    struct  Node tc_Node;
    UBYTE   tc_Flags;
    UBYTE   tc_State;
    BYTE    tc_IDNestCnt;
    BYTE    tc_TDNestCnt;     /* ------------------------------ */
    ULONG   tc_SigAlloc;      /* segnali allocati               */
    ULONG   tc_SigWait;       /* segnali su cui siamo in attesa */
    ULONG   tc_SigRecvd;      /* segnali ricevuti               */
    ULONG   tc_SigExcept;     /* segnali "in eccezione"         */
    UWORD   tc_TrapAlloc;     /* ------------------------------ */
    UWORD   tc_TrapAble;
    APTR    tc_ExceptData;
    APTR    tc_ExceptCode;
    APTR    tc_TrapData;
    APTR    tc_TrapCode;
    APTR    tc_SPReg;
    APTR    tc_SPLower;
    APTR    tc_SPUpper;
    VOID    (*tc_Switch)();
    VOID    (*tc_Launch)();
    struct  List tc_MemEntry;
    APTR    tc_UserData;
};

/*----- Segnali Predefiniti ------------------------------------*/

#define SIGB_ABORT      0
#define SIGB_CHILD      1
#define SIGB_BLIT       4
#define SIGB_SINGLE     4
#define SIGB_DOS        8

#define SIGF_ABORT      (1<<0)
#define SIGF_CHILD      (1<<1)
#define SIGF_BLIT       (1<<4)
#define SIGF_SINGLE     (1<<4)
#define SIGF_DOS        (1<<8)
```

*Figura 2*
*La struttura «task» e i segnali predefiniti.*

*Figura A*

```
int segnale, numero;

segnale = AllocSignal(numero); /* Numero è compreso fra 16 e 31  */
if (segnale == -1) Error();    /* Error() è una procedura utente */
```

*Figura B*

```
int segnale;

segnale = AllocSignal(-1);     /* Dammi il prossimo segnale libero */
if (segnale == -1) Error();    /* Error() è una procedura utente   */
```

segnali che un certo lavoro ha allocato, sui quali è in attesa, che ha già ricevuto, e via dicendo. Il meccanismo a segnali è uno dei meccanismi base di comunicazione nell'Amiga. In realtà spesso il programmatore non deve gestire direttamente i segnali, ma lavorare con meccanismi di comunicazione di livello più alto che fanno uso dei segnali, come nel caso dei messaggi *[message system]* che vedremo tra poco.

Un task può definire fino a 32 segnali indipendenti allo stesso tempo. È importante tener presente che il significato che ha un certo segnale è relativo solo al task che lo ha emesso e deve quindi essere chiaro a priori al task che lo riceve. Un segnale, infatti può essere spedito solo ad uno specifico task, e non a più task contemporaneamente. Ad esempio, supponiamo che un certo programma debba compiere ogni tanto una certa operazione. Quando il programma parte, crea un lavoro figlio *[child task]* che si addormenta rimanendo in attesa di un certo segnale (diciamo il numero 20). Appena il programma principale ha bisogno di compiere l'operazione in questione, manda al lavoro servente *[server]* il segnale concordato. Questi sa che tale segnale significa «OK. Fai quella determinata operazione», compie l'operazione e poi si riaddormenta. Prima di determinare le proprie attività, il programma principale manderà un altro segnale, differente dal precedente (diciamo 30), che ordinerà al servente di chiudere tutto e cancellarsi dalla memoria. Come abbiamo detto, il significato dei segnali è relativo, e quindi nulla ci avrebbe impedito di assegnare al segnale 30 il significato «esegui l'operazione» ed al 20 «chiudi tutto».

Dato che il sistema può fare occasionalmente uso di segnali, esiste una convenzione per l'allocazione di un segnale: i sedici inferiori sono in genere riservati per il sistema, mentre i restanti sedici sono a disposizione del lavoro (alcuni segnali di sistema sono riportati in figura 2). Quindi, se un lavoro ha la necessità di chiedere un segnale specifico come nell'esempio precedente (N. 20 & 30), allora si scriverà come in figura A, se invece non si richiede uno specifico numero, allora si scriverà come in figura B.

Una volta che un segnale non è più necessario bisogna rilasciarlo, esattamente come si fa per la memoria allocata. In questo caso si userà la funzione **FreeSignal()**. Ricordate inoltre che un segnale può essere allocato solo dal lavoro che lo usa, non da un altro. Inoltre il numero del segnale non è usato dal programma così come è, dato che rappresenta solo il numero del bit nella doppia voce a disposizione del programma. Per poterlo usare è necessario prima convertirlo in una maschera *[mask]*, nel seguente modo:

**maschera = 1 << segnale;**

A questo punto qualcuno si chiederà a cosa serve chiedere ad EXEC un segnale qualunque indipendentemente da uno specifico valore, dato che in questo caso nessun altro lavoro potrà utilizzarlo. La risposta è che ci sono casi, come vedremo per la tecnica a messaggi, in cui si associa un segnale ad una particolare struttura. Quando quella struttura viene interessata da un evento esterno, il segnale viene attivato ed il lavoro può eseguire una specifica attività predefinita. In questo caso ha poca importanza il valore del segnale in sé, ma solo come aggancio per l'identificazione della struttura interessata. È questo il caso delle porte.

È possibile specificare ad EXEC quali azioni deve compiere quando un segnale viene ricevuto.

1) Il lavoro non è in attesa: il segnale viene mantenuto e basta;

2)il lavoro è addormentato in attesa: viene svegliato in modo da poter decidere cosa fare in seguito all'arrivo del segnale;

3) il lavoro è sveglio ma in attesa attraverso un particolare modo detto «di eccezione» *[exceptions]*: uno specifico task indipendente viene attivato *qualunque cosa stesse facendo il lavoro che ha ricevuto il segnale*, svolge una certa attività e poi torna il controllo al task principale, che nel frattempo era entrato in uno stato detto «sospeso» *[suspended]*.

Un lavoro si mette in attesa su uno o più segnali per mezzo della funzione **Wait()**. Analogamente un lavoro manda un segnale ad un altro lavoro usando la funzione **Sigma()**.

Non entreremo ulteriormente nel dettaglio dato che la maggior parte di queste funzioni sono di *basso livello*. Difficilmente un programmatore le userà, a parte forse la **Wait()**. È importante comunque conoscere almeno in generale il meccanismo a segnali, dato che su di esso è basato quello a messaggi, che andiamo a spiegare.

## Porte e Messaggi

EXEC fornisce al programmatore un meccanismo di comunicazione tra lavori estremamente flessibile e potente chiamato *comunicazione tramite messaggi*.

In Amiga, un messaggio *[message]* è una sorta di autorizzazione tramite la quale il lavoro che lo riceve può accedere temporaneamente ad uno spazio di memoria che appartiene a quello che trasmette il messaggio. Da un punto di vista operativo, si tratta di una tecnica per scambiarsi informazioni sotto forma di strutture dati. Tale comunicazione deve avvenire seguendo delle regole molto precise che garantiscono l'integrità delle informazioni e la consistenza della comunicazione stessa.

Un messaggio è sempre spedito ad una porta di destinazione *[message port]*. Questa è rappresentata nel sistema da una struttura dati ed è associata in genere ad un determinato lavoro. I messaggi che arrivano ad una porta sono accodati secondo una sequenza di tipo FIFO *[first-in-first-out]*, cioè «il primo che arriva è anche il primo ad essere letto». L'unico limite al numero di porte apribili contemporaneamente od al numero di messaggi accodabili su una determinata porta è dato dalla memoria disponibile nel sistema.

## Porte

Una porta, come abbiamo detto, è una specie di cassetta delle lettere nella quale i messaggi sono accodati in ordine di arrivo. Essi possono essere stati originati da lavori differenti, senza limite teorico di numero. Ad una porta può essere associato un nome simbolico al quale fare riferimento successivamente. Tale nome può servire anche in caso di *debug* del programma (vedi figura 3).

Quando un messaggio arriva ad una porta, questa si può comportare in diversi modi a seconda del valore che ha un determinato elemento di questa struttura chiamato **mp_Flags** (vedi nota 2).

PA_IGNORE
accoda il messaggio e basta;

PA_SIGNAL
accoda il messaggio ed avverti il lavoro, tramite un segnale *[signal]*, che c'è un nuovo messaggio per lui;

PA_SOFTINT
accoda il messaggio e causa una interruzione software *[software interrupt]* del lavoro che possiede la porta in questione (vedi nota 1).

Abbiamo quindi tre possibili situazioni quando arriva un messaggio:
1. il messaggio viene accodato ed è responsabilità del lavoro che lo ha ricevuto andarselo a prendere non appena ha la possibilità di farlo, se vuole;
2. il lavoro riceve l'avviso che il messaggio è arrivato e, se stava «dormendo» *[wait state]*, viene svegliato affinché lo possa leggere;
3. il lavoro viene interrotto in ogni caso ed una particolare procedura indipendente si occupa di eseguire una serie di operazioni predefinite come conseguenza dell'arrivo del messaggio stesso.

EXEC mette a disposizione varie funzioni per la gestione delle porte. Vediamone alcune che utilizzeremo in seguito, facendo sempre riferimento alla figura 3.

### *CreatePort()*

Crea una porta e la inizializza nel modo seguente:
- alloca la memoria necessaria alla struttura che rappresenta la porta;
- alloca un segnale e lo associa alla porta;
- inizializza **mp_Flags** a **PA_SIGNAL**;
- nomina il lavoro che l'ha chiamata proprietario della nuova porta, assegnando a **mp_Task** il puntatore alla struttura che rappresenta il lavoro in questione;
- associa alla porta, se specificato, un nome definito dal programma e, in tal caso, aggiunge la porta alla lista delle porte gestite dal sistema *[system message-port list]*;
- inizializza la lista dei messaggi.

Se non viene specificato alcun nome per la porta, questa non è aggiunta alla lista di sistema e non può essere referenziata in seguito a meno che non venga aggiunta usando **AddPort().**

Può comunque essere usata lo stesso da tutti i lavori che ne conoscono il puntatore. Viceversa due porte possono avere lo stesso nome ma priorità diversa come vedremo più avanti. Il nome di una porta è in realtà memorizzato nel campo ***ln_Name** della **struct Node** inclusa nella **struct MsgPort.**

### *DeletePort()*

Cancella una porta precedentemente creata. Attenzione: prima di cancellare una porta, assicuratevi di aver letto e risposto a tutti i messaggi in coda su quella porta, altrimenti rischiate di impedire ad un altro lavoro in attesa di una vostra risposta di continuare la propria attività.

### *FindPort()*

Restituisce al programma il puntatore ad una porta conosciuta dal sistema e referenziata tramite nome. Se si ha il dubbio che esistano altre porte con lo stesso nome, si può utilizzare la funzione **FindName()** vista nella scorsa puntata, per cercare le successive (vedi sempre figura 3).

### *Messaggi*

Come abbiamo già detto, quella dei messaggi è una tecnica usata per far sì che più lavori possano comunicare fra di loro. Ad esempio, supponiamo che un certo lavoro (mittente) voglia chiedere ad un altro lavoro (destinatario) di com-

#### *Note*

1. Quando un lavoro riceve una interruzione software, esso smette di fare quello che stava facendo, qualunque cosa fosse stata, ed esegue una serie di operazioni predefinite previste per quella specifica interruzione. Queste operazioni sono in realtà eseguite in uno spazio di memoria differente da quello del lavoro interrotto e vengono effettuate da una specifica procedura detta gestore *[handler]*.
2. In realtà solo alcuni bit di **mp_Flags** sono usati come segnalatore per l'azione da effettuare. Essi sono identificati tramite la costante predefinita **PF_ACTION**.

```
/*
 *  Strutture associate ad una porta ed ad un messaggio
 */
struct MsgPort
{
  struct Node mp_Node;      /* Utilizzata per memorizzare nome e priorità */
  UBYTE mp_Flags;           /* azioni da intraprendere all'arrivo di un   */
                            /* nuovo messaggio.                           */
  UBYTE mp_SigBit;          /* E' il numero del segnale usato se mp_Flags */
                            /* è uguale a PA_SIGNAL.                      */
  struct Task *mp_SigTask;  /* Se mp_Flags = PA_SIGNAL punta alla struct  */
                            /* Task associata al lavoro a cui segnalare,  */
                            /* se invece mp_Flags = PA_SOFTINT, punta ad  */
                            /* una struttura di tipo "interrupt".         */
  struct List mp_Msglist;   /* Testa della lista contenente i messaggi.   */
};

struct Message
{
  struct Node mn_Node;          /* Serve per accodare il messaggio alla  */
                                /* porta alla quale arrivano (MsgList).  */
  struct MsgPort *mn_ReplyPort; /* "Indirizzo de mittente" in caso sia   */
                                /* necessaria una risposta.              */
  UWORD mn_Length;              /* Lunghezza del messaggio in bytes.     */
};

/*
 *  Alcune funzioni di gestione delle porte
 */
struct MsgPort *porta;
BYTE priorita;
char *nome;

struct MsgPort *CreatePort(nome,priorita); /* Crea una nuova porta.      */
void DeletePort(porta);                    /* Cancella una porta.        */
struct MsgPort *FindPort(nome);            /* Cerca una porta "nome".    */
void AddPort(porta);                       /* Aggiungi una porta alla    */
                                           /* lista di sistema.          */
void RemPort(porta);                       /* Rimuovi una porta dalla    */
                                           /* lista di sistema.          */

/*
 *  Alcune funzioni di gestione dei messaggi
 */
struct Message *messaggio;

void PutMsg(porta,messaggio);     /* Spedisci un messaggio ad una porta */
struct Message *GetMsg(porta);    /* Ricevi un messaggio da una porta   */
void ReplyMsg(messaggio);         /* Rispondi ad un messaggio           */
struct Message *WaitPort(porta);  /* Attendi un messaggio da una porta  */

/*
 *  Esempio di ricerca di tutte le porte con il nome "PortaDiProva"
 */
#define PORTA "PortaDiProva"
struct MsgPort *porta;
USHORT numero = 0;

porta = FindPort(PORTA);

while(porta) /* Continua finchè trovi porte chiamate "PortaDiProva" */
{
  printf("Trovata la porta di prova numero %d\n",++numero);
  porta = FindName(porta, PORTA);
}
```

*Figura 3 - Porte e messaggi.*

piere una operazione di lettura da un file e di restituirgli il risultato.

Un possibile scenario è il seguente:

- il lavoro destinatario, dopo aver creato una porta per eventuali richieste in arrivo (chiamiamola «PostaInArrivo»), se ne va tranquillamente a dormire;
- il mittente prepara il messaggio dentro il nome del file ed un buffer sufficiente a contenere i dati letti;
- quindi crea una porta per il messaggio di ritorno e ne carica il puntatore in un campo opportuno nel messaggio (come vedremo tra breve);
- dopo chiede al sistema il puntatore alla porta «PostaInArrivo» utilizzando la **FindPort()**;
- se trovata, spedisce il messaggio e va a dormire pure lui;
- il messaggio arriva alla porta di destinazione, il lavoro destinatario si sveglia, controlla il messaggio e attiva l'operazione di lettura;
- a questo punto, se tutto è andato bene, carica i dati letti nel buffer creato appositamente nel messaggio, rimanda il tutto al mittente e torna a dormire;
- il mittente si sveglia, prende il messaggio di ritorno, carica i dati, li elabora e, se non ha più richieste per l'altro lavoro, chiude la porta che aveva creato e termina le proprie attività.

Naturalmente questo è solo uno dei tanti possibili esempi, tra l'altro neanche particolarmente ottimizzato, di scambio messaggi tra due lavori.

Un messaggio è fatto di due parti:

1. la parte di sistema, e
2. quella utente.

La parte di sistema è formata da una struttura chiamata **struct Message** (vedi figura 3). Questa contiene un nodo per agganciare il messaggio alla lista relativa alla porta di destinazione, il puntatore ad una porta da usare per la risposta (vedremo fra poco a cosa serve) e la lunghezza del messaggio vero e proprio in byte.

La parte utente può essere una qualunque struttura (o serie di strutture) complessa a piacere. Dato che la lunghezza del messaggio in **mn_Lenght** non è utilizzata dal sistema, potete dare a questo campo il significato che preferite.

Il messaggio più semplice possibile può contenere solo la parte di sistema (lunghezza 0). In tal caso chi lo riceve deve capire da solo qual è il suo significato, ad esempio guardando il contenuto del campo «nome» della struttura «nodo» in testa al messaggio stesso. Viceversa possiamo fornire una serie di informazioni altamente strutturate utilizzando strutture, unioni o vettori. Un messaggio complesso è rappresentato sempre da una struttura che contiene come primo elemento una **struct Message** (come mostrato nell'esempio in figura 4).

In ogni caso è bene aver presente i seguenti punti.

- Il tipo del nodo nella parte di sistema deve essere NT_MESSAGE per messaggi in partenza NT_REPLYMSG per quelli di risposta. In quest'ultimo caso la modifica è effettuata automaticamente dalla funzione **ReplyMsg()**.
- In Amiga, i messaggi sono passati per referenza, non copiati da uno spazio di memoria all'altro. Questo significa che «spedire un messaggio» non è altro che un modo per autorizzare temporaneamente un lavoro ad usare una parte della propria memoria.
- È importante che un lavoro risponda ad un messaggio al più presto possibile, per evitare che un altro lavoro perda tempo in attesa di una risposta e che ritardi a restituire al sistema la memoria utilizzata per il messaggio stesso. Un programma «educato» aumenta le performance del sistema a tutto vantaggio dell'utente finale.
- Quando un lavoro viene svegliato perché è arrivato un messaggio su una certa porta, deve assicurarsi che non ne siano arrivati altri prima di mettersi a dormire di nuovo. In ogni caso deve assicurarsi di aver risposto a tutti i messaggi arrivati e che la porta sia vuota prima di chiuderla e terminare le proprie attività.

Vediamo ora rapidamente le funzioni EXEC per la gestione dei messaggi (figura 3).

**PutMsg()** Spedisce un messaggio ad una determinata porta.

**GetMsg()** Riceve il primo messaggio arrivato ad una determinata porta (FIFO). Il messaggio è rimosso dalla lista associata a quella porta.

**ReplyMsg()** Rispedisce un messaggio al mittente. Utilizza il puntatore in **mn_ReplyPort**. Il contenuto del messaggio non è più disponibile una volta risposto.

**WaitPort()** Manda a dormire il lavoro in attesa di un messaggio dalla porta specificata. Restituisce il puntatore al primo messaggio in lista ma non lo rimuove dalla porta.
Bisogna usare **GetMsg()** per questo.

Questo è quanto. Ai fini della comprensione di alcune tecniche che useremo nelle prossime puntate, quando parleremo di Intuition, non è necessario aggiungere altro. In realtà il discorso è molto più complesso e quanto detto non è certo sufficiente per scrivere lavori che siano in grado di comunicare fra loro. Abbiate pazienza: ci arriveremo...

```
/*
 *  Esempio di messaggio per la ricezione di dati da un file
 */
struct BufferMsg
{
  struct Message msg;
  char filename[161];
  BYTE buffer[BUFSIZE];
};

/*
 *  Esempio di messaggio per la trasmissione di coordinate
 */
struct CoordMsg
{
  struct Message messaggio; /* nota: questa DEVE essere una struttura, */
                            /*       NON un puntatore!                 */
  UWORD x,y;
};
```

*Figura 4 - Messaggi.*

## *Conclusione*

Con questa puntata si conclude la prima parte del ciclo di articoli dedicati alla programmazione in C su Amiga. Lo scopo di questa prima parte era quello di fornire al lettore le informazioni di base necessarie a sviluppare programmi in un ambiente multitasking. Questo non vuol dire essere in grado di sfruttare al massimo la potenza di un sistema operativo come quello dell'Amiga, ma conoscere un certo numero di regole fondamentali per interagire con il sistema operativo per quello che riguarda la gestione delle operazioni di scrittura e lettura su file, quelle di lettura da CLI (prima e seconda puntata) e la gestione della organizzazione a directory (terza puntata), e per chiedere alcuni servizi essenziali ad EXEC, come quelli per la gestione delle liste e della memoria (quarta puntata) e quelli per la comunicazione fra lavori (quinta puntata). Dato che queste conoscenze sono essenziali per il prosieguo della nostra analisi della possi-

bilità che l'Amiga offre ad un programmatore C, questa volta non proporremo alcun esercizio, ma vi consigliamo di andare a rileggere le precedenti puntate (le avete tutte, vero?) ed a provare a scrivere qualche programmino per vedere se vi è tutto chiaro. A costo di essere pedante lo ripeto ancora una volta: non sperate di imparare a programmare in C su Amiga solo leggendo queste poche pagine! Questi articoli hanno come scopo quello di invogliarvi a scrivere programmi in C sull'Amiga. Se ci tenete veramente ad imparare dovete provare e riprovare, cercando sui manuali il motivo dei vostri insuccessi, migliorando programmi che siete riusciti a far girare, analizzando il codice sorgente fornito insieme a quello eseguibile in molti dischetti di *Public Domain Software*. Negli Stati Uniti ed in Germania ci sono programmi scritti da persone come noi e voi, che possono magari dedicare solo poche ore alla settimana al proprio Amiga, che non hanno nulla da invidiare a molti programmi di note Software House. Se non sapete niente di PD Software, ne parleremo un po' in una delle prossime puntate.

A proposito... la seconda parte di questa serie è dedicata ad uno dei pezzi forti di Amiga: grafica ed animazione. Inizieremo dalla prossima puntata con *Intuition*: schermi e finestre.
Buona rilettura...

## *Errata corrige*

Nella seconda puntata (MCmicrocomputer n. 75 giugno '88) c'è un errore in figura 3 e nel testo corrispondente.

Al contrario di quanto affermato nell'articolo, la costante **EOF**, predefinita in **stdio.h** con il valore (–1), indica la fine di un file in lettura solo nel caso che vengano utilizzate le funzioni interne del C **fgetc, getc, getchar** e simili. Nel caso della funzione AmigaDOS **Read()**, la fine di un file viene notificata con un valore nullo (0) di ritorno, mentre il valore (–1) indica un possibile errore in lettura.

L'errore è dovuto al fatto che, dato che difficilmente si usano nello stesso programma contemporaneamente funzioni AmigaDOS ed interne per la lettura di un file, molti programmatori, tra cui anch'io, son soliti modificare la **stdio.h** o comunque ridefinire **EOF** come zero. Il programma in figura 3, quindi, funziona regolarmente se, dopo le istruzioni di inclusione (**# include**), viene inserita la seguente dichiarativa:

**# define EOF (Ø)**

Ad i puristi, che, a ragione, preferiscono evitare questo genere di ridefinizioni, dato che portano appunto a generare un po' di confusione, suggerisco la definizione di una nuova costante da usare solo per le funzioni di I/O dell'AmigaDOS, come qui di seguito:

**#define ADOS_EOF (Ø)**

In effetti, la segnalazione di «fine file» non è del tutto costante anche per le funzioni interne del C. Per questo, prendo spunto dall'errore in questione, di cui mi scuso comunque con i lettori, per fornire la seguente tabellina. Le informazioni si riferiscono a funzioni AmigaDOS e del *Lattice C 4.0*.

| Funzione | Tipo | End Of File | Errore di I/O |
|---|---|---|---|
| Read | AmigaDOS | 0 | -1 |
| Write | AmigaDOS | non applicabile | -1 |
| read | C 4.0 | 0 (*) | -1 (**) |
| write | C 4.0 | non applicabile | -1 (**) |
| dread | C 4.0 | 0 (*) | -1 (**) |
| dwrite | C 4.0 | non applicabile | -1 (**) |
| fread | C 4.0 | n (***) | n (***) |
| fwrite | C 4.0 | n (***) | n (***) |
| fflush | C 4.0 | non applicabile | -1 |
| flushall | C 4.0 | non applicabile | -1 |
| fgetc | C 4.0 | -1 | -1 |
| fgetchar | C 4.0 | -1 | -1 |
| getc | C 4.0 | -1 | -1 |
| getchar | C 4.0 | -1 | -1 |
| fgets | C 4.0 | 0 | 0 |
| gets | C 4.0 | 0 | 0 |
| fputc | C 4.0 | non applicabile | -1 |
| fputchar | C 4.0 | non applicabile | -1 |
| putc | C 4.0 | non applicabile | -1 |
| putchar | C 4.0 | non applicabile | -1 |
| fputs | C 4.0 | non applicabile | 0 |
| puts | C 4.0 | non applicabile | 0 |

| Funz/Var | Tipo | Descrizione |
|---|---|---|
| IoErr | AmigaDOS | Ritorna un codice di ritorno relativo all'errore occorso. |
| feof | C 4.0 | Non-zero se raggiunta la fine del file specificato. |
| ferror | C 4.0 | Non-zero se si è verificata una condizione di errore per il file specificato. |
| errno | C 4.0 | Variabile di controllo. Azzerata all'inizio, assume il valore del codice di errore UNIX se si verifica una condizione di errore di I/O. Va riazzerata dal programma dopo la verifica. |
| _OSERR | C 4.0 per AmigaDOS | Variabile di controllo. Azzerata all'inizio, assume il valore del codice di errore Amiga se si verifica una condizione di errore di I/O. Va riazzerata dal programma dopo la verifica. |
| perror | C 4.0 | Manda a "stderr" il messaggio di errore UNIX relativo al valore assunto da errno. |
| poserr | C 4.0 | Manda a "stderr" il messaggio di errore Amiga relativo al valore assunto da _OSERR. |

### *Note:*

* Questa funzione specifica la lunghezza del buffer in lettura. Si ha quindi EndOfFile anche se il numero di byte effettivamente letti (valore di ritorno della funzione) è inferiore alla lunghezza specificata.

** Questa funzione specifica la lunghezza del buffer in lettura o scrittura. Si ha quindi una condizione di errore anche se il numero di byte effettivamente letti o scritti (valore di ritorno della funzione) è superiore alla lunghezza specificata.

*** Questa funzione specifica il numero di blocchi da leggere o scrivere. Si ha quindi EndOfFile od una condizione di errore anche se il numero di blocchi effettivamente letti o scritti (valore di ritorno della funzione) è inferiore a quello specificato.

Come si può vedere, sia la condizione di fine file, sia quella di errore di I/O, vengono notificate nei vari casi in modi molto differenti. Inoltre ci sono alcuni casi in cui non è possibile dire a priori quando si è verificata l'una e quando l'altra. In questo caso è bene sempre fare riferimento alle seguenti funzioni e variabili di controllo. MC

# RICORDI ▷ Archimedes

## Buon lavoro, con la potenza del RISC!

▷ **RISC:** è il principio di **Archimedes**, lo straordinario e velocissimo personal computer a 32 bit ▷ Mettetelo alla prova con un foglio elettronico come **SigmaSheet**, 200 volte più rapido dei suoi simili (ricalcola un cash-flow di 32 anni *in meno di 25 secondi*), o con un integrato come **Pipe-dream** (predisposto per comunicare con i portatili della nuova generazione), o con un project-manager versatile come **Logistix**, o con un database come **System Delta Plus** (che può gestire oltre *due miliardi di records*) ▷ Confrontate la potenza dei pacchetti di *grafica*, del software per applicazioni *musicali, didattiche, scientifiche, mediche* ▷ Valutate la facilità con cui sono state sviluppate soluzioni originali e sofisticatissime nei vari linguaggi disponibili (**BBC Basic, Assembly, C, Pascal, Fortran 77, Lisp, Prolog**) ▷ Apprezzate la possibilità di continuare a utilizzare tranquillamente i vostri pacchetti **MS-DOS** preferiti ▷ Mai un computer così nuovo e rivoluzionario ha avuto tanto software così presto ▷ Ed è solo il principio.


DOPPIOUNI

Acorn
The choice of experience.
Un'azienda del gruppo Olivetti

G. RICORDI & C.
Settore Informatico
Via Salomone, 77
20138 MILANO
tel. 02/5082-315

Distributore esclusivo:

Per maggiori informazioni, inviate questo coupon a G. RICORDI & C. Settore Informatico, Via Salomone, 77, 20138 MILANO

Desidero avere maggiori informazioni su Archimedes

Nome:

Cognome:

Qualifica professionale:

Ditta, Ente o Scuola:

Indirizzo: