Questo mese niente programmi nella rubrica del software Amiga. Solo un po' di notizie interessanti e consigli da parte del noto (e preparato) Mangrella e dal sottoscritto. Tema comune delle due parti, la comunicazione tra processi. Uniprocessor nel primo caso e addirittura multiprocessor nel secondo. Non spaventatevi più di tanto: Maurizio Mangrella ci affascinerà con messaggi e porte viste, tanto per cambiare, da Basic (ormai l'ha ridotto un colabrodo!) i miei «consigli» riguardano invece la possibilità di lanciare applicazioni MS-DOS sulla Janus (o nel Sidecar) direttamente da Workbench Amiga clickando su un'icona. A tutti gli interessati, buona lettura!

Ports, Devices e altro

di Maurizio Mangrella

Lo stato di attesa

Prima di entrare nel merito, è d'obbligo parlare di una delle caratteristiche peculiari dei sistemi multitasking: lo stato d'attesa. Quando un processo richiede un dato, non ancora disponibile, al sistema operativo o a una periferica, va direttamente in «stato di attesa», ovvero viene temporaneamente «dimenticato» dalla CPU (che può dedicarsi agli altri task) fino a quando il dato richiesto non diventa effettivamente disponibile. In tal modo tutto il sistema può girare sempre al massimo della velocità.

Nell'Amiga ad ogni task è associata una particolare struttura dati (figura 1) che contiene le caratteristiche peculiari del relativo processo. Tra le altre cose è facile rinvenire la voce sigBit: è quella che è responsabile dello stato d'attesa di un processo. Nel momento in cui un processo cerca di accedere ad una risorsa del sistema non immediatamente disponibile, deve prima associare ad essa un signal bit, il quale viene aggiunto alla maschera dei signal bit già attivati. Il sistema operativo provvede a manipolare questi bit, ponendoli a 0 se il relativo dato non è ancora disponibile (stato di attesa) o a 1 nel caso stia effettivamente arrivando. Nel momento in cui l'apposito interrupt per i quanti di tempo manda il 68000 in stato supervisore per il cambio di contesto, il sistema operativo sceglie il prossimo task ed esamina la sua schiera di signal bit: se sono tutti a 1 il processo è in stato pronto (eseguibile), altrimenti si passa al prossimo processo.

Un signal bit può essere associato a varie cose: un messaggio su un port, una richiesta di I/O o una risorsa del sistema. Noi esamineremo «ambetré» (aggettivo creato da adp) i casi.

Risorsa del sistema

Per manipolare una risorsa di un sistema, esistono vari metodi: in genere vengono associate, ad essa, particolari funzioni per l'arbitraggio. Tanto per fare un esempio ormai molto noto, prendiamo il Blitter: per questa risorsa esistono le funzioni OwnBlitter(), DisOwnBlitter() e WaitBlit(), rispettivamente per impadronirsene, lasciarlo al sistema ed aspettare che abbia finito il corrente lavoro. In particolare l'ultima manda il processo chiamante in stato d'attesa fino a quando il Blitter non sia realmente disponibile: contemporaneamente gli altri processi potranno girare più velocemente. Nota: la struct GfxBase (articolo del sottoscritto su MC 77, figura 4) contiene un puntatore al task (BlitOwner) che detiene il processo del Blitter. A buon intenditor...

I ports

È ora di parlare dei ports, mito tanto decantato di Amiga. Un port (come già dicemmo in altra occasione) è un mezzo di comunicazione che consente, a task differenti, di scambiarsi informazioni e altri dati. Un giorno (me tapino!) mi capitò sottomano una rivista, piuttosto partigiana, in cui si elogiava la facilità d'uso (!) del KickStart. Ai miei sospetti (evidentemente l'autore dell'articolo non aveva capito molto dell'Amiga...) si aggiunse una amara certezza quando decisi di mettere in pratica le seguenti parole: «Creare un port è molto semplice: basta invocare la routine CreatePort della exec.library». Seguiva una discreta descrizione del funzionamento di questa routine. Il risultato? Mi lanciai verso il mio 500, ovvero verso l'ignoto... Signore e signori, la funzione CreatePort NON ESISTE! Questo per dare un'idea della attendibilità di certe informazioni...

Comunque, anche se per altra via, è possibile creare un port (ovviamente): basta metter su la struttura dati di cui in figura 2 e passarne l'indirizzo alla routine AddPort della exec.library, così:

CALL AddPort (Port&)

Nella figura 2 compare la struct Node, «espansa» in figura 3. Un Node (letteralmente... nodo!) è un punto d'incrocio

```
extern struct Task {
     struct   Node tc_Node;
     UBYTE    tc_Flags;
     UBYTE    tc_State;
     BYTE     tc_IDNestCnt;
     BYTE     tc_TDNestCnt;
     ULONG    tc_SigAlloc;
     ULONG    tc_SigWait;
     ULONG    tc_SigRecvd;
     ULONG    tc_SigExcept;
     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;
};
```

Figura 1 - La struttura Task.

```
struct MsgPort {
     struct   Node mp_Node;
     UBYTE    mp_Flags;
     UBYTE    mp_SigBit;
     struct   Task *mp_SigTask;
     struct   List mp_MsgList;
};
```

Figura 2 - La struct MsgPort.

```
struct Node {
     struct   Node *ln_Succ;
     struct   Node *ln_Pred;
     UBYTE    ln_Type;
     BYTE     ln_Pri;
     char     *ln_Name;
};
```

Figura 3 - La struct Node.

```
struct List {
     struct  Node *lh_Head;
     struct  Node *lh_Tail;
     struct  Node *lh_TailPred;
     UBYTE   lh_Type;
     UBYTE   l_pad;
};
```

Figura 4 - La struct List.

```
struct Message {
     struct  Node mn_Node;
     struct  MsgPort *mn_ReplyPort;
     UWORD   mn_Length;
};
```

Figura 5 - La struct Message.

```
struct Unit {
     struct  MsgPort *unit_MsgPort;
     UBYTE   unit_flags;
     UBYTE   unit_pad;
     UWORD   unit_OpenCnt;
};
```

Figura 6 - La struct Unit.

```
struct IORequest {
     struct  Message io_Message;
     struct  Device  *io_Device;
     struct  Unit    *io_Unit;
     UWORD   io_Command;
     UBYTE   io_Flags;
     BYTE    io_Error;
};
```

Figura 7 - La struct ioRequest.

per una richiesta, da parte di un certo numero di task, di uno stesso dato: si compone di un puntatore al Node successivo e a quello precedente (generalmente posti a 0), di un tipo, di una priorità (tra −128 e +127) e di un puntatore al nome del port (necessario).

Il tipo può essere uno di quelli rappresentati in tabella A.

A quanto so, un interrupt software è un modo di intercettare un messaggio del S.O., come, ad esempio, una condizione di errore; non chiedetemi di più.

Per quanto riguarda il tipo, noi sceglieremo NT_MSGPORT, cioè 4.

Gli ultimi due tipi (insieme ad un certo numero di routine di gestione) sono disponibili solo a partire dalla versione 1.2 del S.O.

La priorità è codificata in un byte in complemento a 2, per cui 80 hex. vale −128 e FF hex. vale −1: dunque occhio.

Dopo il Node abbiamo un byte di flag (cfr. dopo), un byte indicante il signal bit di quel port, il puntatore al task chiamante e una List (confermazione in figura 4) dei messaggi. Una List si compone di tre puntatori a Nodes (metteteli tutti a 0 senza pietà), un byte di tipo e un «pad» per l'allineamento e indirizzo pari (anche a questi a 0).

Il flag può essere, per un port, PA_SIGNAL (stato d'attesa determinato in base a un signal bit), PA_SOFTINT (stato d'attesa su un interrupt sofwtare); o PA_IGNORE (ignora i signal bit del port); rispettivamente valgono 0, 1 e 2.

Esiste poi anche un non meglio specificato PF_ACTION, che vale 3.

Per inviare un messaggio, bisogna confermare un'apposita struttura dati (figura 5), che si compone di un Node (come sopra, ma con type NT_MESSAGE), di un puntatore al port (ReplyPort) e della lunghezza del messaggio (fino a 65535 caratteri!), che deve seguire la struttura dati.

Del ReplyPort parleremo più avanti. Per ora accontentatevi di sapere come si manda un messaggio: basta dare.

CALL PutMsg (Port&,Msg&)

Per riceverlo «dall'altra parte» basta dare

MAS&=GetMsg& (Port&)

che ritorna l'indirizzo di una struct Message (se c'è almeno un messaggio in coda ad un port) o, altrimenti, 0. GetMsg ritorna un valore, dunque va dichiarata in testa al programma. A Msg&+20 ritroveremo il nostro messaggio; vi consiglio di terminarlo comunque con un CHR$(0) (che userete anche come marker di fine stringa) perché non sempre (leggi: mai!) il S.O. lascia invariato il campo mn_Length.

Ci sono vari metodi per scoprire se un messaggio ha effettivamente percorso il giusto «iter burocratico»: il migliore è quello di rispedirlo al mittente tramite il ReplyPort (in genere è lo stesso port utilizzato per inviare il messaggio). Per la cronaca, questo è il metodo utilizzato da Intuition per i suoi port: dopo aver mandato un messaggio corrispondente ad un dato IDCMP di una finestra, «chiede conferma» e non fa nulla (o quasi) fino a quando il messaggio non le viene rispedito. Per rispedire un messaggio si può usare la PutMsg o, più semplicemente, la ReplyMsg, nella forma.

CALL ReplyMsg (Msg&)

Abbiamo visto anche che, ad un port, si associa un nome: in base a questo è possibile ritrovarlo, dando

Port&=FindPort& (Name&)

dove Name& punta a una stringa (che termina con un CHR$(0)) contenente il nome del port. Se più port hanno lo stesso nome, FindPort ritorna l'indirizzo di quello con priorità più elevata.

Per attendere un messaggio...

... ci sono tre modi:

1) invocare la GetMsg fino a quando non è disponibile un messaggio;
2) aspettare che il port contenga un messaggio tramite la WaitPort ();
3) mandare il processo in stato d'attesa con la Wait ().

Con il primo metodo abbiamo la possibilità di fare qualcos'altro e di testare il port solo una volta ogni tanto: inoltre il S.O. provvede a mettere in coda eventuali messaggi multipli e a fornirli al ricevente nell'ordine in cui sono stati inviati. Con i modi 2) e 3) mandiamo il task in stato d'attesa, bloccandolo completamente ma avvantaggiando qualche altro processo che gira in concorrenza.

Per fermare un task su un port, basta dare la

CALL WaitPort (Port&)

la quale non ha effetto (ovvero ritorna direttamente al task chiamante) nel caso il port contenga già qualche messaggio. Un po' diversa è la Wait, che si

◄ *Tabella A*

Tipo	Codice	Funzione
NT_UNKNOWN	0	Per attivare un Node senza troppi problemi
NT_TASK	1	Richiesta di un processo
NT_INTERRUPT	2	Condivisione di un interrupt
NT_DEVICE	3	Condivisione di un Device
NT_MSGPORT	4	Un port attivo
NT_MESSAGE	5	Un messaggio su un port
NT_FREEMSG	6	(Boh!?) Un messaggio per tutti i task
NT_REPLYMSG	7	Un messagio ripetuto al mittente
NT_RESOURCE	8	Condivisione di una risorsa del sistema
NT_LIBRARY	9	Condivisione di una library del KickStart
NT_MEMORY	10	Condivisione della memoria
NT_SOFTINT	11	Condivisione di un interupt software
NT_FONT	12	Condivisione di un font di caratteri
NT_PROCESS	13	Per la manipolazione di un processo
NT_SEMAPHORE	14	Installazione della logica a semafori
NT_SIGNALSEM	15	Gestione dei signal bit di un semaforo

Tabella B ▼

Comando	Codice	Funzione
CMD_INVALID	0	Non dovrebbe essere invocato (credo...)
CMD_RESET	1	Resetta il device interessato
CMD_READ	2	Legge dal device
CMD_WRITE	3	Scrive al device
CMD_UPDATE	4	Aggiorna (come? Boh!?)
CMD_CLEAR	5	Pulisce i buffer
CMD_STOP	6	Ferma l'attuale attività del device
CMD_START	7	Fa partire un'attività (quale? Boh!?)
CMD_FLUSH	8	Forza la scrittura dei buffer al device
CMD_NONSTD	9	È la base di molti comandi specifici

invoca con

Result& = xWait& (Mask&)

(si noti la x preposta automaticamente dal ConvertFD per evitare conflitti con l'interprete) dove Mask& è una maschera in cui sono settati i bit corrispondenti, per posizione, ai signal bit che si vogliono testare: in pratica equivale (in C) a

while ((sigBit & Mask) = =0); /*Stato di attesa*/

Result& è il valore dell'espressione (sigBit & Mask) al momento dell'uscita dal loop: per sapere quali signal bit hanno causato il ritorno allo stato di pronto, basta testarlo.

Questo consente al processo di andare in stato d'attesa controllando più signal bit: se solo uno di quelli specificati diviene attivo (o lo è già) il task «si risveglia» e riprende a funzionare normalmente.

```
struct IOStdReq (
    struct  Message io_Message;
    struct  Device  *io_Device;
    struct  Unit    *io_Unit;
    UWORD   io_Command;
    UBYTE   io_Flags;
    BYTE    io_Error;
    ULONG   io_Actual;
    ULONG   io_Length;
    APTR    io_Data;
    ULONG   io_Offset;
);
```

Figura 8 - La ioRequest standard.

```
struct IODRPReq (                          /* Dump RastPort */
    struct  Message io_Message;
    struct  Device  *io_Device;
    struct  Unit    *io_Unit;
    UWORD   io_Command;
    UBYTE   io_Flags;
    BYTE    io_Error;
    struct  RastPort *io_RastPort;
    struct  ColorMap *io_ColorMap;
    ULONG   io_Modes;
    UWORD   io_SrcX;
    UWORD   io_SrcY;
    UWORD   io_SrcWidth;
    UWORD   io_SrcHeight;
    LONG    io_DestCols;               /* Porre 0 */
    LONG    io_DestRows;               /* Porre 0 */
    UWORD   io_Special;                /* Porre 84 hex. (FULLCOL | ASPECT)
);
La descrizione completa dei flags e delle modalità di funzionamento è con -
tenuta nell'include file "devices/printer.h", che non riporto per motivi di
spazio .
```

Figura 9 - La ioRequest per l'hardcopy su carta.

Bello, vero? Ovvero: sarebbe bello se funzionasse! Il problema è che nessuno dei flag del port sembra funzionare a dovere; dunque io sistemo un port, «alloco» un signal bit e poi do vita all'esperimento: niente da fare! Ho provato sia con la Wait () che con la WaitPort () (che fa uso della Wait, la maledetta), con questi risultati: con PA_SIGNAL il ricevente ignora completamente i messaggi inviati; con PA_SOFTINT si scatena la «Illegal Instruction» Exception (Guru #00000004 etc); con PA_IGNORE (l'unico che funziona a dovere!) il ricevente si blocca e con PF_ACTION c'è addirittura l'«Address Error» (Guru #00000003 etc.). Dov'è la verità? Non ci si capisce niente! Solo un'ultima nota: per trovare il puntatore alla propria struct Task, un processo deve invocare la FindTask () nella forma

Task& = FindTask (Name$)

L'argomento della FindTask () è il nome del task (al solito un puntatore a una stringa terminata da un CHR$(0)): dando 0 si ottiene l'indirizzo (niente popodimeno!) del Sistema Operativo. Per ottenere senza problemi l'indirizzo del proprio task, basta accedere al port di Intuition (vi ricordate? No? guardate su MC 77!) con

IPrt& = PEEKL (WINDOW (7)+86)

e trovare la propria struct Task con

Task& = PEEKL (IPrt&+16)

Per chi programma in C o altro, dirò che WINDOW (7) è il puntatore al Window Record ritornato dalla OpenWindow (), che provvede a creare l'apposito port di Intuition in base all'indirizzo del task chiamante.

Le I/O Requests

Al fine di favorire un buon funzionamento del sistema multitasking dell'Amiga, bisogna anche condividere le risorse con altri processi: in pratica non è sempre possibile impadronirsi di una risorsa senza chiedere il permesso al S.O. L'Amiga comprende alcuni driver per specifiche risorse, detti «Devices», il cui primo scopo è l'interfacciamento dell'hardware del sistema con il Sistema Operativo. Ogni Device ha un suo specifico nome: audio.device, printer.device, parallel.device, serial.device, trackdisk.device (per l'accesso al disco a basso livello), etc.

Per «aprire» un Device, ovvero per allocare una fetta degli accessi a un processo, basta dare

ERR& = OpenDevice& (Name&,Unit&,ioReq&,flags%)

Err& è un eventuale errore (0 se è andato tutto liscio), Name& è il nome del device (al solito modo), Unit& punta a una struct Unit (figura 6), ioReq& punta alla ioRequest che che stiamo trattando (figura 7) e flags%... non so, ponetelo a 0. Unit& dovrebbe specificare ulteriori informazioni sul Device: in genere si può porlo a 0. Una ioRequest si compone di un Message, di un puntatore a Device (ponete 0) e di un puntatore a Unit (anche qui 0, come prima). Il campo mn_Lenght della struct Message, in questo frangente, può essere benissimo ignorato. Seguono comando, flags (???) e un eventuale errore. Ogni comando ha uno specifico numero: quelli generici sono riportati nella tabella B.

A questi (che necessitano della specificazione del Device nel campo io_Device) si aggiungono quelli specifici, che si auto-indirizzano al relativo Device: ad esempio, l'hardcopy ha il codice 11. La forma standard di una ioRequest è in figura 8; a questa si accompagnano le forme particolari per i comandi specifici. Per lanciare una ioRequest basta dare

Err& = DoIO& (ioReq&)

dove Err& è diverso da 0 se qualcosa è andata male, altrimenti vale 0. Nel caso il processo non sia andato in stato d'attesa (perché la ioRequest può essere soddisfatta contemporaneamente al task chiamante) si può attenderne la fine con

CALL WaitIO (ioReq&)

o fermarla con

CALL AbortIO (ioReq&)

Conclusioni

Elencare tutte le ioRequests speciali sarebbe stato troppo lungo, perciò mi sono limitato a quella standard e a quella per l'hardcopy (figura 9).

Comunque le trovate tutte negli include file del C per quanto riguarda i Devices (devices/gameport.h, devices/trackdisk.h, etc.)

Con questo vi lascio: buona sperimentazione!

Icone per la Janus

di Andrea de Prisco

Secondo appuntamento con i trucchetti Janus, la scheda che permette la compatibilità IBM pressoché totale. Questo mese vedremo come «appiccicare» una bella icona di Intuition anche ai nostri bravi programmi MS-DOS, in modo da richiamarli con un semplice (anzi doppio) colpo di mouse. Sì avete proprio capito bene, seguendo i consigli di questo mese per lanciare una applicazione MS-DOS non dovremo più digitare il suo nome dalla PCwindow, ma potremo farlo più semplicemente manovrando solo ed esclusivamente il «topo».

Ma il bello di tutto ciò è che il trucco è realizzabile senza scrivere una sola riga di codice ma semplicemente sfruttando le risorse del sistema e un po' di batch file dei due sistemi.

Innanzitutto è necessario disporre della release 1.3 del sistema operativo o, quantomeno, del device logico «pipe» e dell'utility Xicon che permette di «lanciare» batch file da workbench.

Pipe è, secondo me, il device logico più interessante di Amiga. Premetto però che non dispongo ancora di nessuna documentazione tecnica a riguardo, e tutto quello che so è solo frutto delle mie elucubrazioni mentali notturne e diurne sull'argomento.

Nel device pipe possiamo scaricare file per poi rileggerli (una sola volta). La lettura può avvenire da parte dello stesso programma che ha scritto o da programmi diversi. Anche la Janus può accedere al pipe essendo questo un device vero e proprio e quindi indirizzabile come tutti gli altri prefissando il nome del file dal pipe: la cosa più importante (dal punto di vista strettamente informatico) del pipe è che permette una cooperazione tra processi asincrona.

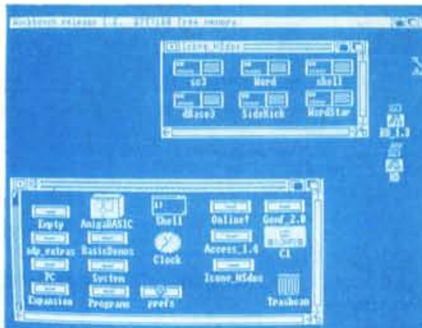
Un processo scarica un file nel pipe e procede per la sua strada, un altro processo che aspetta qualcosa dal pipe non rileva errore per la sua mancanza del file ma semplicemente aspetta. Così, per fare un esempio, se da un cli digito:

type pipe:NomeFile

se il file è presente nel pipe sarà stampato, altrimenti il cli rimane in attesa. A questo punto, per sbloccare la situazione occorre che qualcun altro inserisca il file nel pipe, ad esempio un altro cli:

dir > pipe:NomeFile

Xicon permette invece di attaccare una icona ad un batch file in modo da permettergli di partire anche da workbench e non esclusivamente da cli. Il suo uso è assai semplice: dopo aver preparato il batch file prepariamo un'icona di tipo project appropriata. Il modo



più semplice per farlo è prendere una icona di tipo project già esistente (un qualsiasi documento, anche del notepad) copiarla da cli e rinominandola come il file batch da lanciare. Fatto questo (disponiamo del «file» e del «file.info») l'ultimo passo è quello di accedere da wb all'info dell'icona (primo menu pull down a sinistra) e cambiare il default tool in Xicon, meglio se preceduto da tutto il percorso atto a raggiungerlo.

Detto questo passiamo ai «trucchetti» veri e propri. Il metodo per ottenere icone funzionanti anche per i programmi msdos si basa, naturalmente, sulla cooperazione tra scheda Janus e Amiga. Dobbiamo cioè mettere in un certo senso la scheda Janus in attesa di ordini da Amiga e quest'ultimo, a seconda dell'icona clickata, impartirà questo o quell'ordine. La prima cosa da fare per realizzare questa comunicazione, come noto, è di lanciare in background dal lato Amiga l'utility pcdisk e da lato msdos impiantare nel config.sys un bel DEVICE=JDISK.SYS. Quest'ultimo, si sa, deve essere presente al momento del boot msdos, quindi se lo inserite in un secondo momento occorre agire di ctrl-alt-del prima di rendere attivo. Poi bisogna preparare il seguente batch file msdos che chiameremo LOOP.BAT:

AREAD PIPE:COMANDO COMANDO.BAT
COMANDO

Cosa fa questo file? Semplice: aspetta che Amiga ponga il file COMANDO nel pipe, lo copia nel suo file COMANDO
.BAT e lo esegue. Per lanciare il file LOOP potremo, naturalmente, inserire tale parola nell'AUTOEXEC.BAT oppure digitarlo da shell una volta in msdos. E dal lato IBM siamo a posto.
Per quanto riguarda Amiga la cosa non si complica gran che e tutto dipende da come avrete installato le varie applicazioni msdos sull'hd. Di solito l'utente crea una directory per ogni applicazione quindi per farla partire occorrerà prima entrare in quella directory e poi digitare il nome del programma da richiamare. E noi faremo proprio così. Amiga non farà altro che spedire via pipe dei piccoli batch file che una volta lanciati in msdos faranno partire l'applicazione nella directory giusta. Un esempio? Eccolo:

CD WORD
WORD
CD ..
LOOP

Indovinate cosa fa...

Semplice, ma per capirlo bene dobbiamo tornare un attimo sulla Janus e ricordarci che lì avevamo lanciato il file LOOP.BAT. Allora, Amiga spedisce nel pipe il batch file appena mostrato (dandogli come nome «comando») e la Janus, ferma sull'AREAD non ancora soddisfatto, può caricare il file nel suo disco, dandogli come nome COMANDO.BAT. Appena effettuato il trasferimento parte il batch file ricevuto (la seconda linea di LOOP fa proprio questo) e così in msdos si passerà alla directory word e sarà caricato il programma di videoscrittura. Attenzione: il batch file COMANDO resta di videoscrittura. Attenzione: il batch file COMANDO resta in attesa che si esca da word (questa volta coi metodi tradizionali) in modo da ritornare alla root ("CD ..") e ricaricare LOOP per un nuovo ordine da parte di Amiga. Discorso analogo per un'altra applicazione, ad esempio db3, dove il batch file da spedire sarà:

CD DB3
DBASE
CD ..
LOOP

e così via per le altre applicazioni presenti sul vostro HD msdos. Bene, a questo punto la domanda è obbligata: chi spedisce i file nel pipe? Risposta: le nostre icone. Ed è qui che salta fuori l'utility Xicon. Infatti, per ogni applicazione da lanciare dovremo preparare oltre al batch file msdos da inviare (e quindi da tenere nella partizione Amiga) anche il minuscolo file «speditore» che appiccicheremo alla icona settata come visto prima per lanciare automaticamente Xicon. Ad esempio se il file da spedire si chiama «word.bat» attaccheremo all'icona il file:

COPY WORD.BAT PIPE:COMANDO

per il db3 lo speditore sarà:

COPY DB3.BAT PIPE:COMANDO

e così via. Un'ultima cosa: se non desideriamo lanciare altre applicazioni, ma tornare alla shell msdos (che rimane inchiodata ad aspettare il pipe) come dobbiamo fare? Semplicissimo: con lo stesso sistema creeremo una icona settata Xicon, ad esempio di nome shell, contenente la linea:

COPY SHELL.BAT PIPE:COMANDO

e SHELL.BAT sarà un file contenente semplicemente un return. Tutto qui...

MC