ADPmttb 2.0

Multitasking Toolbox per Amiga

di Andrea de Prisco

A partire da questo numero inizieremo a «giocare duro» coi nostri Amiga. E vi assicuro che ne vedremo delle belle. Questo mese vi presento il mio ultimo giocattolino ideato espressamente per sfruttare al massimo, col minimo sforzo e con la massima «pulizia», il multitasking di Amiga

Grazie all'ADPmttb sarà infatti possibile scrivere con facilità applicazioni multitask: formate non da un solo programma, ma da una collezione di processi intercomunicanti in grado cioè di cooperare. Nel corso di questo articolo vedremo anche un paio di semplici esempi di utilizzo dell'ADPmttb, ma dietro le quinte (tanto per non fare anticipazioni) è già funzionante la release 2.0 di ADPnetwork, la rete software per Amiga scritta interamente in ADPmttb.

Ma non voglio anticiparvi nulla di più, altrimenti vi farei perdere il gusto della sorpresa. Prima di iniziare vorrei ringraziare pubblicamente (e in ordine alfabetico) Marco Ciuchini, Andrea Gozzi, Oscar Sillani, Andrea Suatoni che mi hanno aiutato a debuggare le precedenti versioni dell'mttb e coi loro consigli hanno contribuito a rendere tale progetto il più user friendly e completo possibile. Se qualche amighevole lettore, ancora fresco di 64, ricorda l'ADPbasic e il significato di tale sigla (Advanced Disk Printers Basic), aggiungerò che l'ADP di ADPmttb sta per Advanced Distribuited Programming.

Lista delle funzioni implementate dall' ADPmttb

Nome	Funzione
Send	Spedisce una stringa null terminated su una Porta
SendPointer	Spedisce un puntatore su una Porta
SendChar	Spedisce un carattere su una Porta
SendBlock	Spedisce un blocco di memoria su una Porta
Receive	Riceve una stringa null terminated su una Porta
ReceivePointer	Riceve un puntatore su una Porta
ReceiveChar	Riceve un carattere su una Porta
ReceiveBlock	Riceve un blocco di memoria su una Porta
MultiReceive	Riceve stringhe null terminated su max 5 Porte
NewPort	Crea una nuova Porta 'MTTB'
EndPort	Elimina e ripulisce una Porta 'MTTB'
PortWait	Aspetta la creazione di una Porta (con TimeOut)
MessageWait	Aspetta l'arrivo di un Messaggio (con TimeOut)
CheckPort	Restituisce lo stato di una Porta
WipePort	Ripulisce una Porta da 'n' messaggi indesiderati
MessageLen	Restituisce la lunghezza del primo msg su una Porta
StartProcess	Crea un processo e restituisce il segment utilizzato
RunProcess	Esegue fino ad 8 processi in parallelo
WaitEndProcess	Aspetta la terminazione di un processo 'MTTB'
CheckEndProcess	Restituisce lo stato (end/run) di un Processo 'MTTB'

Tabella 1

Bene, ora sono proprio nei guai. Cosi inventerò entro il prossimo mese per giustificare anche ADPnetwork?

Un po' di teoria (scusate...)

Mentre continuo a sognare il mio prossimo personal computer basato su una dozzina di 68000, vi svelerò il motivo per cui sono tanto entusiasta di Amiga. Entusiasmo nato prima ancora di vederlo funzionare, prima di assaporare modi grafici impensabili (nell'86...) o animazioni degne di Workstation ben più importanti. Ciò che mi appassiona di più di questa macchina tanto odiata da molti, è il multitasking finalmente non piu simulato sopra una macchina monotask (non per vantarmene, ma facevo esperimenti del genere anche sul mio fido 64, ben prima dei vari Multilink per PC) ma vivo, vegeto e funzionante, al livello stesso del sistema operativo. Accendete un Amiga e prima di clickare su qualsiasi icona domandatevi quanti programmi sono già in esecuzione sulla vostra macchina. Sono una decina almeno e riguardano i processi di sistema pronti ad esaudire ogni vosta richiesta. A questo punto clickate (memoria permettendo) su tutte le icone che vi pare. Vi accorgerete (anzi ve ne siete già accorti da un pezzo) che il multitask vi permette di sfruttare maggiormente ogni risorsa disponibile sulla vostra macchina, a cominciare dal processore stesso.

Attualmente, sul mio Amiga super espanso, al momento del boot lancio un Word Processor (C1-Text), un programma di comunicazione (JRCOMM), un

È disponibile, presso la redazione, il disco contenente i sorgenti dell'ADPmttb, il file di include «mttb.h», l'utility «mttb» per compilare e linkare processi mttb, più le relative istruzioni.
Per ordinare il dischetto (codice ADP/01) inviare L. 30.000 a mezzo assegno, c/c o vaglia postale alla Technimedia s.r.l. Via Carlo Perrier, 9 - 00157 Roma.

line editor (CED), lo schermo PCcolor della Janus, Gomf, TableCloth, DisplayMemory, PCdisk, Dmouse e non ricordo cos'altro. E non è affatto raro che mentre compilo un programma, da MC-Link via Jrcomm scarico un file, e col fido C1-Text stampo l'articolo da consegnare... ieri. Il tutto in un tempo totale minore della somma dei tre tempi impiegati per eseguire le operazioni sequenzialmente.

Miracolo? No, tecnologia e nemmeno tanto moderna.

Sono infatti decenni che i computer (senza personal davanti, per favore!) lavorano in multitask non per perdere tempo ma per guadagnarlo. E non mi spiego perché il giorno in cui hanno deciso di fare i personal abbiano scelto di riferirsi a schemi di sistemi di elaborazione vecchi alcuni decenni.

Che Amiga (o chi per lui) dovesse prima o poi arrivare non c'è mai stato dubbio. Chissà quanto altro tempo dovremo aspettare per vedere il primo personal multiprocessor, con schema di funzionamento vecchio solo una decina d'anni...

Ma torniamo a noi. Dicevamo che col multitasking è possibile eseguire in parallelo un certo numero di funzioni in un tempo minore della somma dei singoli tempi impiegati per eseguire la stessa serie di operazioni in modo sequenziale.

Il perché è molto semplice e per spiegarlo basta pensare ad un Word Processor che stampa un file. Qualsiasi computer utilizziate, la stampa di un file avviene inviando caratteri da stampare alla stampante. Nella stragrande maggioranza dei casi, la stampante stessa rappresenterà un vero e proprio collo di bottiglia per il sistema. Infatti è lecito pensare che il computer sia in grado di «passare» i caratteri ben più velocemente della capacità della stampante di stamparli. Buffer a parte, che, comunque, prima o poi si riempirà (il caso di buffer infinito non lo prendo in considerazione per ovvi motivi), a regime un sistema computer-periferica di questo tipo avrà un funzionameto del tipo: eccoti un carattere... hai finito? ... eccoti un altro carattere... e così via.

Durante la fase «hai finito?» il processore aspetta risposta dalla periferica e fino a quando quest'ultima non gli darà l'OK, il processore non potrà spedire nessun altro carattere. In un personal computer tradizionale la CPU non potrà fare altro che aspettare pazientemente l'OK, ma in un computer multitask si sfruttano tali tempi «morti» per elaborare altri processi pronti. In altre parole spariscono le fasi di «attesa attiva» sul verificarsi di eventi esterni e al loro posto troviamo delle commutazioni di contesto che permettono, appunto, di far avanzare altri processi. Arrivato l'OK che aspettavamo, naturalmente, occorre tornare al processo di stampa per dare un altro carattere da stampare, e il ciclo si ripete fintantoché c'è qualcosa da elaborare. Esattamente come dire che in un computer multitask la CPU non sta mai in ozio, ma lavora sempre al 100% (che poi è quello che interessa maggiormente).

Detto questo...

Amiga va ancora oltre. La sua architettura infatti non è solo multitask, ma a modo suo addirittura multiprocessor. A parte la scheda Janus che contiene oltre ad un processore aggiuntivo anche la ram e l'elettronica di controllo per fregiarsi del titolo di «computer su scheda» e alla quale possiamo demandare compiti elaborativi anche non indifferenti (tale feature, promessa da anni, non è ancora stata sfruttata da nessuno), troviamo dentro Amiga anche un processore sonoro e un processore grafico che lavorano in parallelismo reale col 68000. Ma questo esula dal nostro discorso e quindi preferirei tornare all'elaborazione «pura».

Come detto all'inizio, il multitask di Amiga non è simulato sopra una macchina monotask, ma il sistema operativo, nel suo livello più basso, Exec, mette a disposizione tutti gli strumenti per la programmazione concorrente. È infatti possibile utilizzare nei nostri programmi C alcune primitive di comunicazione interprocess per permettere a più programmi di «parlarsi». È infatti questa la caratteristica più importante dei sistemi multitask: non tanto «più processi in esecuzione parallela» quanto «più processi in esecuzione parallela in grado di comunicare». Solo in questo modo è possibile scindere applicazioni monotask in più processi concorrenti (e cooperanti). Facciamo un bell'esempio? Torniamo al nostro amato Word Processor, questa volta multiprogrammato. Mettiamo un processo per gestire l'input da tastiera, uno per il controllo ortografico, uno per la formattazione WYSIWYG, uno per la stampa in background e uno denominato «saver» che ogni 100 modifiche del testo provvede a salvarne una copia su HD. Tutti moduli semplicissimi da scrivere che ci permetteranno una volta lanciati di avere testi sempre ben formattati e corretti, salvati per sicurezza ogni cento modifiche e quando decideremo di stampare il file, una volta dato l'ordine potremo immediatamente cominciare a lavorare su un altro testo o addirittura uscire dal WP per caricare qualcos'altro (se la memoria scarseggia).

Tutte feature, s'intende, disponibili anche su programmi monotask, ma provate ad immaginare cosa vuol dire programmare la stampa in background o il controllo ortografico in real time o peggio, l'autosave senza che l'operatore venga rallentato minimamente. Se siete bravi programmatori provate a pensarci su un po' e poi fatemi sapere...

Non dilunghiamoci troppo

Avete letto finora circa novemila caratteri di questo articolo e non vi ho ancora detto, in pratica, cosa diavolo sia questo mttb. A dire il vero non ho ancora finito con l'infarinatura teorica (necessaria!) ma per premiarvi del fatto di avermi seguito fin qui (che coraggio...) vi anticiperò qualcosa. Dunque, l'mttb non è altro che una collezione di subroutine C (vi consiglio vivamente di raggrupparle, compilate, in una libreria linked) con le quali è possibile comunicare tra processi in una maniera più pulita ed intuitiva dei meccanismi offerti da Exec. Oltre a questo con l'mttb è possibile creare processi «figli» completamente indipendenti dal processo «padre» (colui che «crea»), ed effettuare numerose interrogazioni sullo stato delle porte e dei processi creati tramite mttb. La tabella 1 mostra la lista delle funzioni implementate con una breve descrizione accanto.

Purtroppo, per motivi di spazio, non potremo pubblicare direttamente tutto il codice dell'ADPmttb, ma solo una piccola parte (listato 1) riguardante 4 sole funzioni che sono però sufficienti per la maggior parte dei casi. Le rimanenti funzioni le troverete a partire dal prossimo mese in un apposito riquadro che cercheremo di aprire nella rubrica del software o dove ci sarà sufficiente spazio. I più impazienti potranno richiedere il dischetto in redazione nel quale troveranno oltre ai sorgenti, anche la libreria mttb.lib bella e pronta ed una comoda utility che permetterà di scrivere in un solo file tutti i processi da compilare e linkare automaticamente con l'mttb.

Ambiente globale e ambiente locale

Essenzialmente esistono due modi per far comunicare i processi. Amiga fa, come al solito, eccezione, implementando una tecnica sua propria a metà

```
/*************************************
*                                    *
*                                    *
*           A D P m t t b            *
*                                    *
*       MultiTasking  ToolBox        *
*        (versione  ridotta)         *
*                                    *
*       ---------------------        *
*       (c) 1989  ADPsoftware        *
*       ---------------------        *
*                                    *
*************************************/


#include "proto/exec.h"
#include "exec/exec.h"
#include "exec/execbase.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "dos.h"
#include "stdio.h"
#include "workbench/startup.h"

#define MODE_ASYNC   1
#define MODE_SYNC    2
#define MODE_NOWAIT  4
#define MODE_WAIT    8
#define MODE_RVE    16
#define ACTION_ALLOC 1
#define ACTION_FREE  0
#define OP_OK        1
#define OP_FAIL     -1
#define NO_PORT     -2
#define EMPTY_PORT   0

int  SendBlock(UBYTE,char *, char *, int);
int  ReceiveBlock(UBYTE,char *, char *);
int  NewPort(char *);
int  EndPort(char *);

struct adp_message
   {
   struct Message message;
   UBYTE mode;
   int   len;
   char  testo[1];
   };


 /*************************************
 *                                    *
 *       N E W P O R T                *
 *                                    *
 *************************************/

NewPort(porta)
char *porta;
{
struct MsgPort *port;
port = (struct MsgPort *)FindPort(porta);
if (port) return(OP_FAIL);
else
   {
   port = CreatePort(porta,0);
   if (port) return(OP_OK);
   else return(NO_PORT);
   }
}


 /*************************************
 *                                    *
 *       E N D P O R T                *
 *                                    *
 *************************************/

EndPort(porta)
char *porta;
{
struct adp_message *adpmsg;
struct MsgPort *port;
Forbid();
   port = (struct MsgPort *)FindPort(porta);
   if (port)
      {
      while (adpmsg = (struct adp_message *)GetMsg(port))
            if (adpmsg->mode & MODE_SYNC) ReplyMsg(adpmsg);
            else FreeMem( adpmsg, sizeof(struct adp_message) + adpmsg->len );
      DeletePort(port);
      Permit();
      return(OP_OK);
      }
Permit();
return(NO_PORT);
}
```

Listato 1

```
 /*************************************
 *                                    *
 *    S E N D - B L O C K             *
 *                                    *
 *************************************/

SendBlock(mode,porta,msg,len)
UBYTE mode;
char  *msg,*porta;
int   len;
{
struct MsgPort *port, *replyport;
struct adp_message *adpmsg;
int    size;
size = sizeof(struct adp_message)+len;
if ((mode & MODE_SYNC || mode & MODE_ASYNC)==0) return(OP_FAIL);
adpmsg = (struct adp_message *)AllocMem(size,MEMF_PUBLIC);
if (adpmsg == 0) return(OP_FAIL);
adpmsg->mode = mode;
if (mode & MODE_SYNC)
   {
   replyport = CreatePort(0,0);
   if (replyport == 0) {
                       FreeMem(adpmsg,size);
                       return(OP_FAIL);
                       }
   adpmsg->message.mn_ReplyPort = replyport;
   }
else adpmsg->message.mn_ReplyPort = NULL;
adpmsg->message.mn_Node.ln_Type = NT_MESSAGE;
adpmsg->len = len;
CopyMem(msg, adpmsg->testo, len);
Forbid();
   port = (struct MsgPort *)FindPort(porta);
   if (port == 0)
      {
      FreeMem(adpmsg,size);
      if (mode & MODE_SYNC) DeletePort(replyport);
      Permit();
      return(NO_PORT);
      }
   PutMsg(port,adpmsg);
Permit();
if (mode & MODE_SYNC)
   {
   WaitPort(replyport);
   GetMsg(replyport);
   DeletePort(replyport);
   FreeMem(adpmsg,size);
   }
return(OP_OK);
}


 /*************************************
 *                                    *
 * R E C E I V E - B L O C K          *
 *                                    *
 *************************************/

ReceiveBlock(mode,porta,vtg)
UBYTE mode;
char  *porta,*vtg;
{
struct MsgPort *port;
struct adp_message *adpmsg;
int    size;
if ((mode & MODE_WAIT || mode & MODE_NOWAIT)==0) return(OP_FAIL);
size = sizeof(struct adp_message);
Forbid();
   port = (struct MsgPort *)FindPort(porta);
   if (port == 0)
      {
      Permit();
      return(NO_PORT);
      }
Permit();
if (mode & MODE_WAIT)
   {
   WaitPort(port);
   adpmsg = (struct adp_message *)GetMsg(port);
   CopyMem(adpmsg->testo,vtg,adpmsg->len);
   if (adpmsg->mode & MODE_SYNC) ReplyMsg(adpmsg);
   else FreeMem( adpmsg, size + adpmsg->len );
   return(adpmsg->len);
   }
if (mode & MODE_NOWAIT)
   {
   if (adpmsg = (struct adp_message *)GetMsg(port))
      {
      CopyMem(adpmsg->testo,vtg,adpmsg->len);
      if (adpmsg->mode & MODE_SYNC) ReplyMsg(adpmsg);
      else FreeMem( adpmsg, size + adpmsg->len );
      return(adpmsg->len);
      }
   else  return(EMPTY_PORT);
   }
}
```

strada tra l'ambiente locale e quello globale. Altra anticipazione: l'mttb rimette le cose a posto spostando il tutto sulla comunicazione ad ambiente locale. Che vuol dire?

Ne abbiamo già parlato in «Appunti di Informatica» (quando li scrivevo io...) ma ripeterlo in questa sede sicuramente non guasterà.

Se il processo A deve utilizzare alcuni dati del processo B, può chiedere a B di spedirglieli (Ambiente Locale) oppure chiedere in qualche modo il permesso per utilizzarli direttamente (Ambiente Globale). A questo punto dovrebbe essere chiaro il motivo della denominazione, specialmente una volta chiarito che l'ambiente è appunto l'insieme dei dati sui quali un processo opera. Per ambiente locale si intende che ogni processo ha i suoi dati su cui elaborare (il suo ambiente) e non ha modo di accedere, in nessun modo e per nessuna ragione, ai dati (all'ambiente) di un altro processo.

L'unico modo è, appunto, quello di farsi mandare una copia dei dati interessanti, da importare nel proprio ambiente, e solo dopo cominciare l'elaborazione. Nel caso in cui il processo mittente (dei dati) deve continuare la sua elaborazione sugli stessi dati dopo che il destinatario ha completato la sua elaborazione, non dovrà fare altro che aspettare che gli ritornino dopo l'elaborazione «remota».

Se invece l'ambiente è globale, i dati sono grosso modo alla mercé di tutti. Sono fisicamente dislocati in una zona di memoria raggiungibile da tutti i processi che devono utilizzarli e tramite meccanismi di mutua esclusione (tipicamente monitor, semafori o più brutali sospensioni del multitask) sono in uso ad uno o ad un altro processo.

Amiga, the best(...ia)

Cosa fa, di contro, lo sporco Amiga? Utilizza il meccanismo dello scambio messaggi, tipico della comunicazione ad ambiente locale, per implementare una sorta di comunicazione ad ambiente globale. L'unica cosa in grado di spedire è un puntatore alla zona di memoria (locale o allocata dal processo mittente) dando così implicita autorizzazione al processo destinatario ad utilizzare i dati (veri e propri, non una copia) del processo mittente. Per convenzione è poi stabilito che il processo mittente non utilizza gli stessi dati fino a quando il destinatario non restituisce l'autorizzazione, avendo finito di utilizzarli, spedendo un apposito messaggio «vuoto» di risposta. Ecco perché ad ogni PutMsg di Exec fa sempre seguito una:

WaitPort (replayport)

dal lato del mittente mentre dal lato destinatario, dopo aver utilizzato i dati il cui puntatore è arrivato a seguito della GetMsg si effettua una Replay-Msg (port). Pura sincronizzazione, nulla di più.

E per di più a carico dell'utente. Cosa succede infatti se chi programma dimentica di effettuare la ReplayMsg oppure la WaitPort (replayport)? Guru Metitation a più non posso, ma soprattutto... a ragione!

L'ADPmttb

A questo punto entra in gioco l'mttb col quale fare casino sarà veramente molto difficile. L'ambiente, abbiamo detto, diventa per incanto locale e spedire dal processo A al processo B diecimila byte di informazione significa «fisicamente» approntarne una copia per il destinatario che potrà farne quello che vuole: è sua!

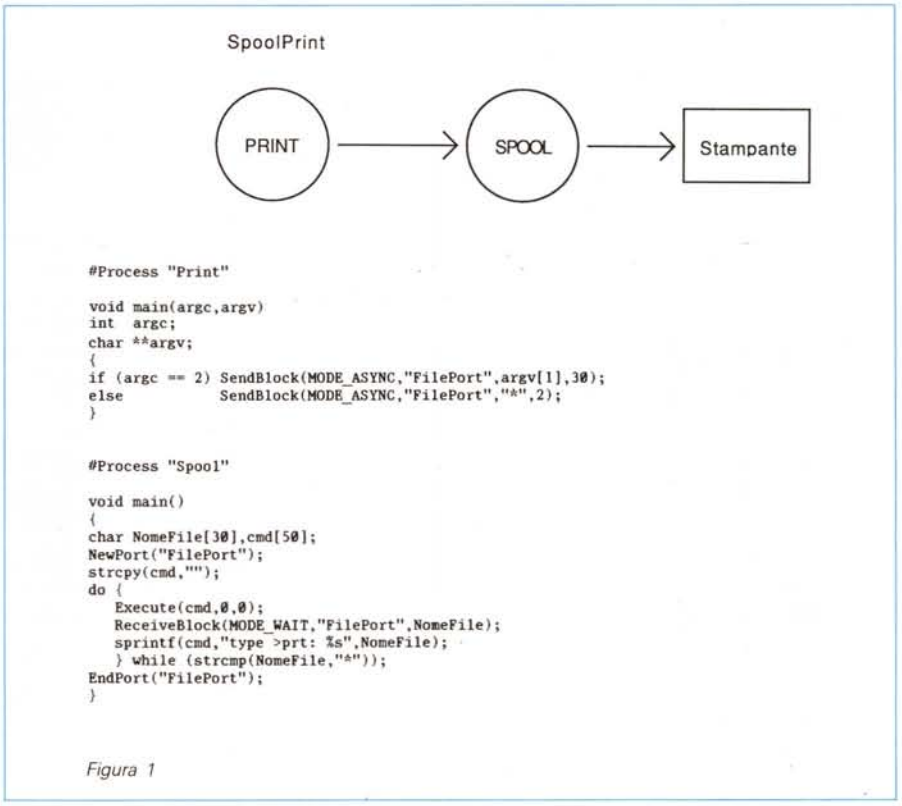
Il tutto, ovviamente, trasparente per l'utente che dovrà solo dire cosa spedire a chi o cosa ricevere da chi e il gioco (perché di gioco si tratta, a questo punto) è fatto.

Sarà perché sono un teorico irriducibile, ma a me questo mttb «mattizza» più della stesso software di rete per Amiga che subito dopo ho realizzato sfruttando l'mttb. Finalmente programmare in multitasking su Amiga diventa possibile senza importare contemporaneamente tutta una collezione di problemi cronici che Exec si porta dietro forse perché rivolto a programmatori esperti (soprattutto in cose sporche!).

Volete un primo, agghiacciante, esempio? In figura 1 è mostrato lo schema di funzionamento di uno spool di stampa manuale. Il processo Spool gira in background e il processo Print è invece invocato da cli seguito dal nome del file da stampare. Da cli possiamo dare ordine di stampa di quanti file vogliamo, semplicemente perché Print torna subito il prompt del cli indipendentemente dal file effettivamente in fase di stampa. Se ad esempio dobbiamo stampare i file «tizio.doc», «caio.doc» e «sempronio.doc» sarà sufficiente digitare in rapida sequenza i tre comandi:

```
Print tizio.doc
Print caio.doc
Print sempronio.doc
```

e sentire la stampante che piano piano (beh, questo dipende dalla vostra stampante) stampa i tre file uno dopo l'altro. Le 19 (diconsi diciannove) linee C che




```
#Process "Print"

void main(argc,argv)
int  argc;
char **argv;
{
if (argc == 2) SendBlock(MODE_ASYNC,"FilePort",argv[1],30);
else           SendBlock(MODE_ASYNC,"FilePort","^",2);
}


#Process "Spool"

void main()
{
char NomeFile[30],cmd[50];
NewPort("FilePort");
strcpy(cmd,"");
do {
   Execute(cmd,0,0);
   ReceiveBlock(MODE_WAIT,"FilePort",NomeFile);
   sprintf(cmd,"type >prt: %s",NomeFile);
   } while (strcmp(NomeFile,"^"));
EndPort("FilePort");
}
```

Figura 1

```
#Process "out"

VOID main(argc,argv)
int    argc;
char **argv;
{
int i,nchr;
struct FileHandle *Open();
struct FileHandle *infh;
char   buffin[BUFSIZ];
BPTR   seg1,seg2,seg3,seg4;
if (argc<3)
   {
   printf("\nUso: Out File LineLength\n\n");
   exit(0);
   }
infh  = Open(argv[1],MODE_OLDFILE);
if (infh == 0) exit(0);
seg1 = StartProcess("MakeWord",0);
seg2 = StartProcess("MakeLine",0);
seg3 = StartProcess("FormatLine",0);
seg4 = StartProcess("Printer",0);
PortWait("word",0);
Send(MODE_SYNC,"word",argv[2]);
PortWait("line",0);
Send(MODE_SYNC,"line",argv[2]);
while ((nchr=Read(infh,buffin,BUFSIZ))!=0)
      for (i=0; i<nchr; i++) SendChar(MODE_SYNC,"char",buffin[i]);
SendChar(MODE_SYNC,"char",'\n');
SendChar(MODE_SYNC,"char",'\1');
Close(infh);
WaitEndProcess(seg1);
WaitEndProcess(seg2);
WaitEndProcess(seg3);
WaitEndProcess(seg4);
}


#Process "MakeWord"

VOID _main()
{
int   i=0;
char  parola[200];
char  carattere;
NewPort("char");
do {
   ReceiveChar(MODE_WAIT,"char",&carattere);
   if (carattere != '\1') parola[i++] = carattere;
   if (carattere == ' ' || carattere == '\n')
      {
      parola[i] = '\0';
      Send(MODE_ASYNC,"word",parola);
      i=0;
      }
   } while (carattere != '\1');
Send(MODE_ASYNC,"word","\1");
EndPort("char");
}


#Process "MakeLine"

VOID _main()
{
int  lr,lw,tot=0;
char lunghezza[5],parola[200],linea[200];
NewPort("word");
Receive(MODE_WAIT,"word",lunghezza);
lr = atoi(lunghezza);
strcpy(linea, "");
Receive(MODE_WAIT,"word",parola);
do {
   lw = strlen(parola);
   if (tot + lw < lr)
      {
      strcat(linea,parola);
      tot+=lw;
      }
   else
      {
      Send(MODE_ASYNC,"line",linea);
      strcpy(linea,parola);
      tot = lw;
      }
   if (parola[lw-1] == '\n')
      {
      Send(MODE_ASYNC,"line",linea);
      strcpy(linea,"");
      tot = 0;
      }
   Receive(MODE_WAIT,"word",parola);
   } while (strcmp(parola,"\1"));
Send(MODE_ASYNC,"line",linea);
Send(MODE_ASYNC,"line","\1");
EndPort("word");
}


#Process "FormatLine"

VOID _main()
{
int lf,lr,ins,tot,nsp,i,j,k;
char len[5],Flinea[200],linea[200];
NewPort("line");
Receive(MODE_WAIT,"line",len);
lf = atoi(len);
for (i=0;i<(80-lf)/2;i++) Flinea[i] = ' ';
Flinea[i] = '\0';
do  {
    Receive(MODE_WAIT,"line",linea);
    lr=strlen(linea);
    while (linea[--lr]==' ');
    linea[++lr]='\0';nsp=0;
    for (i=0;i<lr;i++) if (linea[i]==' ') nsp++;
    if (linea[lr-1]=='\n')
       {
       strcat(Flinea,linea);
       Send(MODE_ASYNC,"Fline",Flinea);
       Flinea[(80-lf)/2] = '\0';
       }
    else if ( lr == lf || nsp == 0 )
           {
           strcat(linea,"\n");
           strcat(Flinea,linea);
           Send(MODE_ASYNC,"Fline",Flinea);
           Flinea[(80-lf)/2] = '\0';
           }
        else if (linea[0] != '\1')
               {
               tot = lf-lr;
               ins = tot / nsp + 1;
               for (i=0,j=(80-lf)/2;i<lr;i++)
                   {
                   Flinea[j++]=linea[i];
                   if (linea[i] == ' ')
                      for (k=0;k<ins;k++)
                          if (tot)
                             {
                             Flinea[j++] = ' ';
                             tot--;
                             }
                   }
               Flinea[j] = '\0';
               strcat(Flinea,"\n");
               Send(MODE_ASYNC,"Fline",Flinea);
               Flinea[(80-lf)/2] = '\0';
               }
    } while (linea[0] != '\1');
Send(MODE_ASYNC,"Fline","\1");
EndPort("line");
}


#Process "Printer"

VOID _main()
{
int                i=0;
char               buffout[200];
struct FileHandle  *outfh;
struct FileHandle  *Open();
NewPort("Fline");
outfh = Open("prt:",MODE_NEWFILE);
if (outfh == 0) exit(0);
Write(outfh,"\n\n\n",3);
Receive(MODE_WAIT,"Fline",buffout);
while(strcmp(buffout,"\1"))
   {
   Write(outfh,buffout,strlen(buffout));
   if (++i == 60)
      {
      Write(outfh,"\n\n\n\n\n\n\n",6);
      i=0;
      }
   Receive(MODE_WAIT,"Fline",buffout);
   }
Close(outfh);
EndPort("Fline");
}
```

Listato 2

accompagnano la figura 1 sono il codice necessario, utilizzando l'mttb, ad implementare ambedue i processi Print e Spool.

Forme di comunicazione

Nella comunicazione ad ambiente locale, esistono alcune forme tipiche che ho creduto opportuno implementare nell'mttb. Infatti dire che il processo A manda qualcosa al processo B è abbastanza riduttivo. Bisogna vedere cosa succede se B non è pronto a ricevere il messaggio oppure, viceversa, se B è pronto ma A non ha ancora effettuato l'invio.

Ma prima di affrontare questo problema (con risoluzione a scelta dell'utente) è necessario stabilire in che modo avviene la comunicazione vera e propria tra due processi. Per questo mi sono completamente appoggiato allo schema di Exec che prevede l'utilizzo di porte di proprietà dei processi destinatari. Tali porte sono invece accessibili in scrittura da parte di tutti i processi mittenti. Dunque un processo che deve ricevere qualcosa su una porta, la prima cosa che deve fare è creare la porta stessa dandogli un nome. La funzione mttb da utilizzare è:

```
NewPort ("NomeDellaPorta");
```

e da quel momento in poi è possibile accedervi sia per spedire (tramite Send) che per ricevere (tramite Receive) messaggi. Quando una porta non serve più è necessario deallocarla con la funzione:

```
EndPort ("NomeDellaPorta");
```

che a differenza della corrispondente DeletePort di Exec prima di deallocarla la ripulisce di tutti i messaggi accodati ma non letti.

Tornando al discorso delle forme di comunicazione esistono essenzialmente comunicazioni sincrone, asincrone, bloccanti, non bloccanti e a rendez-vous esteso. Per comunicazione sincrona si intende che il processo mittente che effettua una Send non va avanti nell'elaborazione fino a quando il corrispondente processo destinatario non esegue la Receive che preleva il messaggio dalla porta. In altre parole, dal punto di vista logico, l'istante in cui il processo mittente spedisce e quello in cui il processo destinatario riceve coincidono temporalmente. Coincidono al punto che se... non coincidono di per sé, coincidono «a forza» dal momento che chi arriva prima aspetta l'altro.

Nella comunicazione asincrona ciò non succede. Il mittente che esegue la Send non aspetta la corrispondente Receive ma va avanti nella sua elaborazione. Il messaggio si accoda sulla porta (e più messaggi possono accodarsi) e restano disponibili al processo destinatario per quando vorrà prelevarli. Su questa forma di comunicazione si basa lo Spool prima commentato. Print non fa altro che accodare i nomi dei file da stampare sulla porta «FilePort», infischiandosene dello stato di avanzamento del processo destinatario. Questo, man mano che stampa, preleva nomi di file dalla porta; tutto qui.

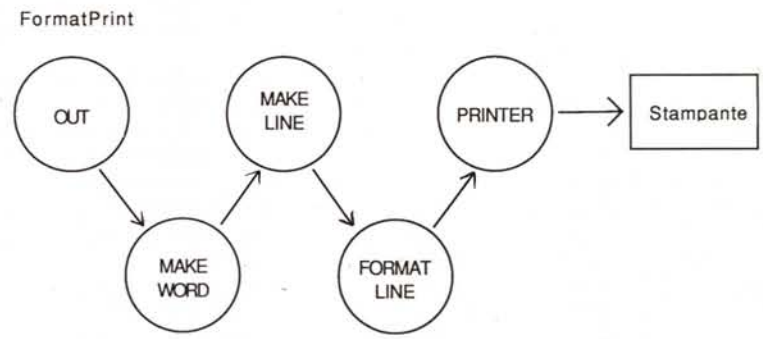


Figura 2

Per quanto riguarda la Receive, bisogna stabilire cosa fare nel caso in cui il messaggio non sia ancora presente sulla porta. Possiamo infatti rimanere in attesa del messaggio (quindi il destinatario aspetta) oppure non aspettare e ottenere un messaggio nullo di ritorno. Nel primo caso si dice che la Receive è bloccante, nel secondo caso non bloccante.

E veniamo ora alle due funzioni di scambio messaggio mostrate nel listato 1: SendBlock e ReceiveBlock. Esse permettono di trasferire da un processo ad un altro un qualsiasi insieme contiguo di byte. Il primo parametro da passare alla SendBlock è la forma di comunicazione adottata, sincrona o asincrona, il secondo la porta interessata (ovvero il nome della porta sulla quale spedire il messaggio), il terzo parametro è il puntatore alla zona di memoria da trasferire e il quarto la lunghezza, in byte, del blocco di memoria in questione. Quindi, se dobbiamo spedire 200 byte di memoria a partire dalla locazione 1000 sulla porta «Pippo» in modo sincrono scriveremo:

```
SendBlock(MODE_SYNC,"Pippo",1000,
200);
```

in modo asincrono:

```
SendBlock(MODE_ASYNC,"Pippo",1000,
200);
```

Ovviamente il caso di trasferimenti di porzioni di memoria nude e crude ci interessa poco, ma fortunatamente in C si gioca tutto coi puntatori che altro non sono che indirizzi di memoria.

Ad esempio se buffer è un array di char lungo 200, e dobbiamo spedirlo sulla porta Pippo, nel caso sincrono scriveremo:

```
SendBlock(MODE_SYNC,"Pippo",buffer,
200);
```

discorso analogo anche per strutture più complesse: se «record» è una struct possiamo scrivere:

```
SendBlock(MODE_SYNC,"Pippo",&record-
,sizeof(record));
```

dove, come noto, &record denota appunto l'indirizzo di memoria dove è memorizzato record. Per finire, il valore restituito dalla SendBlock è una delle tre costanti predefinite OP_OK, OP_FAIL, NO_PORT che indicano rispettivamente che l'operazione è andata a buon fine, che è fallita, che la porta sulla quale intendiamo spedire non esiste (più).

Per quanto riguarda la ReceiveBlock, i parametri da passare sono rispettivamente il modo di ricezione (bloccante o non bloccante), la porta dalla quale prelevare il messaggio e un indirizzo di memoria (un puntatore) dove riporre il messaggio ricevuto.

La ReceiveBlock restituisce la lunghezza del messaggio letto (praticamente il quarto parametro passato alla SendBlock corrispondente).

Se ad esempio aspettiamo sulla porta Pippo (che abbiamo preventivamente creato noi) un buffer di caratteri e siamo interessati al modo bloccante, scriveremo (dopo aver naturalmente dichiarato buffer come array di char sufficientemente grande):

len = ReceiveBlock(MODE_WAIT,"Pippo", buffer)

Per il modo non bloccante:

len = ReceiveBlock(MODE_NOWAIT,"Pippo",buffer)

Resterebbe da descrivere il modo di comunicazione a rendez-vous esteso, ma visto che ho già superato i ventimila caratteri di questo articolo e devo necessariamente dirvi dell'altro prima di chiudere, rimando tale descrizione ai prossimi... riquadri, tanto più che tale modo di comunicazione non è implementato per la SendBlock e ReceiveBlock, ma solo per la Send e Receive che trasferiscono stringhe null terminated da un processo ad un altro.

Note tecniche

Tralascio di commentare anche le due funzioni NewPort e EndPort (listato 1) che sono abbastanza autoesplicative data la loro estrema semplicità. Dedichiamoci alla SendBlock e alla ReceiveBlock.

Innanzitutto il messaggio effettivamente trasferito (ma questo l'utente è autorizzato, anzi, invitato a ignorare) è di tipo struct adp_message, dichiarato in testa al listato 1. Esso è composto da un effettivo campo struct message utilizzato da Exec (e dal sottoscritto) per la trasmissione vera e propria, e da tre campi mode, len e testo che contengono rispettivamente il modo di comunicazione (MODE_SYNC oppure MODE_ASYNC), la lunghezza del messaggio da trasmettere e il testo del messaggio (i byte da trasmettere). Notare come «testo» sia dichiarato come un array di un solo carattere ma non per questo non è possibile trasmettere cose ben più corpose. Vediamo come agisce la SendBlock (seguitemi per favore sul listato 1). Per prima cosa dichiara due puntatori ad una struttura di tipo struct MsgPort. Segue la dichiarazione di un puntatore ad una struttura di tipo struct adp_message. Dopo il controllo sul corretto modo di trasmissione (MODE_SYNC o MODE_ASYNC) si alloca una quantità di memoria atta a contenere il messaggio da trasmettere.

Il puntatore restituito dalla AllocMem è associato ad adpmsg: se ci pensate un attimo, questo equivale a dire che il nostro array «testo» dichiarato nella struttura lungo un solo byte in realtà può essere lungo quanto ci pare, semplicemente allocando una quantità di memoria maggiore di quella strettamente necessaria (pari, per la cronaca, a sizeof (struct adp_message)). Ovviamente ci facciamo anche complici del fatto che il C non esegue alcun controllo sugli indici degli array. Andiamo avanti.

Nel campo adpmsg–>mode mettiamo il modo di trasmissione così come ci è passato dal chiamante. Se tale modo è quello sincrono, occorre creare una replayport e inserirla nel campo «message.mn_ReplayPort», altrimenti poniamo tale campo a NULL. Seguono altri «settaggi» comuni a tutt'e due i modi e infine si copia l'oggetto da spedire nel campo adpmsg–>testo tramite una bella CopyMem di Exec. A questo punto, racchiusa in una coppia di Forbid — Permit effettuiamo la spedizione vera e propria, naturalmente dopo aver rintracciato il puntatore alla porta (noi, infatti, passiamo il nome di questa e non il puntatore alla struct MsgPort).

Infine, se tutto è andato bene e il modo era sincrono, il processo esegue una WaitPort (replayport) per attendere il completamento della corrispondente ReceiveBlock. Segue ovviamente (e solo nel caso MODE_SYNC), la rimozione della replayport e la deallocazione della memoria utlizzata, per adpmsg. Se qualcuno si sta chiedendo chi deallocherà la memoria nel caso di modo asincrono la risposta è molto semplice: la ReceiveBlock corrispondente che ora andrò a commentare.

Le linee iniziali della ReceiveBlock sono molto simili a quelle della SendBlock. Qui non troviamo allocazione di memoria in quanto già fatta dalla SendBlock. Dopo aver cercato l'effettivo puntatore alla porta, occorre distinguere il caso in cui la ReceiveBlock sia bloccante o non bloccante.

Nel primo caso infatti effettueremo una WaitPort(port) nel secondo semplicemente una lettura al volo tramite GetMsg di Exec che restituisce NULL se la port è vuota. Se avevamo richiesto una ReceiveBlock non bloccante e il messaggio non è ancora arrivato restituiamo un bel EMPTY_PORT. Se il messaggio c'è oppure non c'era ma noi abbiamo aspettato il suo arrivo con la WaitPort, copiamo il testo nell'area di memoria passataci come terzo parametro (vtg) e se il modo di trasmissione effettuato dalla SendBlock era sincrono effettuiamo una ReplayMsg(adpmsg) altrimenti il modo era asincrono e non dobbiamo fare altro che deallocare la memoria allocata dalla SendBlock.

Utilizzazione pratica

Per usare l'ADPmttb è sufficiente includere ai vostri processi mttb il listato 1 prima di compilarli normalmente. In questo caso però, saranno presenti nel vostro codice oggetto anche routine da voi non invocate con conseguente spreco di memoria. Una maniera più corretta di utilizzo è, come già detto prima, quella di preparare una libreria di funzioni mttb da linkare ai vostri processi dopo la normale compilazione. In questo caso, la parte da includere in ogni processo sarà la sola parte iniziale del listato 1 fino alla prima funzione (NewPort) esclusa. Col Lattice C 4.0 che uso normalmente, è sufficiente separare le funzioni in tanti file (avendo l'accortezza di ripetere in ognuno le definizioni e gli include iniziali) e compilarli col comando:

LC -Rmttb.lib [lista di file]

Quando poi compilerete i vostri processi, darete il comando:

LC -L+mttb.lib [NomeFile.c]

Per chi invece acquisterà il dischetto in redazione, la procedura di compilazione sarà completamente automatizzata utilizzando una apposita utility che troverete sul dischetto.

Conclusioni

Per quei pochi coraggiosi che sono giunti fin qui, il listato 2, e la figura 2, mostrano un esempio di utilizzo di ADPmttb completo: non la versione ridotta mostrata questo mese. Si tratta di uno spool di stampa multitask che provvede anche alla giustificazione e alla paginazione del testo da stampare. Il primo processo, out, digitato da cli e seguito dal nome del file da stampare e dalla larghezza di stampa desiderata, non fa altro che aprire il file e carattere dopo carattere lo spedisce al processo MakeWord (creato, come gli altri, dallo stesso Out).

Questo riceve caratteri e forma parole che provvede a spedire a MakeLine il quale si occupa di formare appunto le linee lunghe meno della larghezza di stampa desiderata. A questo punto FormatLine, che riceve da MakeLine la linea da formattare, esegue la giustificazione e spedisce la linea formattata al processo printer che stampa. È chiaro che si tratta solo di un esempio di programmazione multitasking e la stessa funzione si sarebbe potuta ottenere con minore dispendio di energie con un solo programma monotask.

Ma come programma monotask, cosa l'avrei messo a fare in questo articolo? Beh, l'unica cosa certa è che la stanchezza non perdona. La mia e la vostra. Buona notte...

MC