12. Un esempio di driver di dispositivo

Nel precedente capitolo si è trattato delle operazioni di I/O ad alto livello e della tabella di scelta del dispositivo (device switch table), *devtab*, che costituisce la struttura generale collegando interrupt, dispositivi, e routine dei driver. Questo capitolo esplora le routine del driver della *CONSOLE*, che è composta dalla tastiera e dal video del PC. Vedere un esempio aiuterà a capire come operano i driver e comprendere come la struttura della scelta del dispositivo facilita il compito di configurare i dispositivi e i relativi driver nel sistema.

12.1 I dispositivi TTY

PC-Xinu usa il termine *tty* per indicare i tradizionali "terminali del computer". Ogni dispositivo tty dispone di una tastiera in grado di trasmettere caratteri al computer e un dispositivo di output in grado di visualizzare i caratteri ricevuti dal computer. Nel PC, i circuiti della tastiera e del video sono integrati nel sistema; altri dispositivi di tipo tty potrebbero essere collegati alle linee di comunicazione asincrone del computer. Il termine "tty" è preso in prestito dai vecchi sistemi che utilizzavano i dispositivi Teletype (telescriventi) come terminali.

In termini generali, il compito del driver del dispositivo tty è di far corrispondere operazioni come leggere o scrivere caratteri ad operazioni che trasmettono caratteri allo schermo e ricevono caratteri dalla tastiera. In pratica il driver comunica col BIOS, richiedendogli di inviare o ricevere caratteri. Per minimizzare le interferenze tra I/O e processi in esecuzione, il driver crea ed utilizza dei processi aventi questa particolare finalità: di trasmettere caratteri allo schermo e di ricevere caratteri dalla tastiera. In aggiunta, esso coordina le richieste per l' I/O con la velocità del dispositivo. Quest'ultimo compito è particolarmente importante per i dispositivi asincroni guidati dagli interrupt, perché i tempi di trasmissione dei caratteri sono spesso di vari ordini di grandezza più lenti della velocità di esecuzione dei processi, cosa che alcuni programmatori apprezzano finché non devono scrivere driver.

I driver tty lavorano tramite parametri, così essi possono essere usati in una varietà di configurazioni. Vari parametri controllano l'"echo" dei caratteri. Il PC opera nel cosiddetto modo *full-duplex*. In questo modo, il BIOS non mostra automaticamente il carattere così come digitato dall'utente; piuttosto il driver trasmette allo schermo ogni carattere ricevuto dalla tastiera, così l'utente può vedere ciò che ha digitato. Tuttavia, non tutti i terminali tty hanno bisogno dell'echo dei caratteri, Quelli che lavorano nel modo *half-duplex* mostrano il carattere digitato automaticamente. Se il driver effettuasse l'eco dei caratteri ricevuti da un terminale half-duplex, ne apparirebbero due copie sullo schermo. Così il driver tty mantiene altri parametri che gli dicono alcune cose come se sia o no necessario effettuare l'eco dei caratteri non stampabili come una sequenza di caratteri stampabili.

La maggior parte dei parametri dei tty risulteranno ovvi per chiunque abbia usato un terminale, ma i lettori che non hanno familiarità con l'hardware dei terminali potrebbero confondersi su quelli che si occupano dello spostamento sulle nuove linee. Il terminale riconosce due caratteri non stampabili che controllano il movimento del cursore. *RETURN*, di solito chiamato *carriage return (ritorno del carrello)* riporta il cursore all'inizio della riga corrente (senza spostarlo verticalmente). *NEWLINE*, talvolta chiamato *line feed*, sposta il cursore verticalmente, in basso di una linea (senza spostarlo orizzontalmente). Lo schermo deve ricevere sia *NEWLINE* che *RETURN* per spostare il cursore all'inizio della linea successiva. Le tastiere hanno tasti separati per generare *RETURN* e *NEWLINE*, ma la nomenclatura non è standard cosicché essi possono essere chiamati "return", "line feed", "enter", "newline", "end line", o "invio". Sui PC, il tasto *RETURN* è etichettato con "return" (o "invio"), mentre NEWLINE viene generato dalla combinazione *Ctrl-J*; il tasto "Enter" ("Invio") del tastierino numerico genera anch'esso *RETURN*.

Sebbene i terminali spediscano ed interpretino *NEWLINE* e *RETURN* separatamente, per i programmi è meglio avere a che fare con un unico carattere che indichi la fine di una riga, sia per l'input che per l'output. In PC-Xinu, il carattere "privilegiato" per indicare la fine di una riga è NEWLINE, denotato con "\n" nelle stringhe e nelle costanti di tipo carattere. Per semplificare la programmazione, il driver del tty riconosce sia *RETURN* che *NEWLINE*, secondo diversi parametri. Un parametri, indicato con *icrlf* controlla se il driver debba sostituire *RETURN* con *NEWLINE* se questo viene ricevuto dalla tastiera. Un altro parametro, *ocrlf*, controlla se il driver debba inserire *RETURN* nel flusso di caratteri diretti allo schermo ogni volta un programma scrive *NEWLINE*.

12.2 La due parti (upper and lower halves) del driver del dispositivo

Come la maggior parte dei driver, le routine del driver del tty possono essere divise in sue sottoinsiemi: la parte superiore (*upper-half*) e quella inferiore (*lower-half*). I processi utente chiamano le routine della upper-half (indirettamente tramite *devtab*) per leggere o scrivere caratteri. Queste routine non manipolano direttamente i dispositivi. Invece, essi mettono in coda le richieste di trasferimento e si appoggiano alle routine della lower-half per compiere, più tardi, i trasferimenti . Questa divisione, che è difficile da apprezzare all'inizio, è il cuore del progetto del driver - è fondamentale perché separa la normale elaborazione dalle caratteristiche dell'hardware.

La code delle richieste di trasferimento è la principale struttura dati che unisce le chiamate ad alto livello alle azioni sul dispositivo. Ogni dispositivo dispone della propria coda di richieste, e il contenuto degli elementi della coda dipende dalle caratteristiche del dispositivo. Le richieste per dischi e dispositivi simili devono specificare la direzione del trasferimento (lettura o scrittura), la posizione dei dati, e la loro lunghezza. Le richieste di trasferimento di caratteri sono più semplici: solitamente, esse sono composte solo dal carattere.

Oltre che per la coda delle richieste, il driver potrebbe avere bisogno di spazio per un *buffer*. I driver usano lo spazio del buffer per registrare i dati in uscita dal momento in cui l'utente richiede che vengano spediti fino a quando non vengono ricevuti dal driver. Essi usano lo spazio del buffer anche per registrare i dati in arrivo dal momento in cui il dispositivo li rende disponibili fino a quando il programma utente non li richiede.

I buffer sono importanti per varie ragioni. In primo luogo, il driver può accettare i dati in arrivo in un buffer prima che il processo utente li legga. Questo è importante per dispositivi come i terminali dove l'utente può iniziare a scrivere in ogni momento. In secondo luogo, dispositivi come i dischi spessa trasferiscono i dati in grossi blocchi. Il sistema deve avere un buffer grande abbastanza per contenere tutto ciò che il dispositivo trasferisce, anche se l'utente ha bisogno soltanto di un carattere. Infine, l'uso del buffer permette al driver di effettuare I/O contemporaneamente ai processi utente. Quando un processo utente scrive dati, il driver li copia nel buffer e permette al processo di continuare l'esecuzione mentre trasferisce i dati dal buffer al dispositivo.

Il driver del tty qui descritto utilizza due buffer circolari di caratteri per terminale, uno per l'input e l'altro per l'output. Le operazioni di output depositano i caratteri che devono essere scritti nel buffer di output e restituiscono il controllo al programma chiamante. Frattanto, il processo di output prende il successivo carattere dal proprio buffer e lo invia allo schermo.

L'input funziona in modo simmetrico. Ogni volta che la tastiera genera un'interrupt, segnalando che ha ricevuto un carattere, il gestore degli interrupt chiama la routine dell'interrupt che gestisce l'input. Essa informa la lower-half del processo di input che legge il carattere in attesa lo deposita nel buffer circolare di input. Quando un processo chiama una routine della parte alta del driver per leggere caratteri, questa li prende dal buffer di input, aspettando ulteriore input solamente se nel buffer è presente un numero insufficiente di caratteri rispetto a quelli richiesti.

Idealmente, le due metà del driver comunicano esclusivamente tramite i buffer condivisi:

Le routine della upper-half mettono in coda le richieste per il trasferimento dei dati o per il controllo dei dispositivi; esse non interagiscono direttamente con i dispositivi. Le routine della lower-half trasferiscono i dati dai buffer o controllano i dispositivi; esse non interagiscono direttamente con i programmi utente.

In pratica, le due metà del driver potrebbero dover fare più che non semplicemente manipolare i dati condivisi. Per esempio, la upper-half del processo di output dovrebbe potere svegliare la lower-half quando inserisce dell'output nel buffer. Potrebbe anche accadere che nessun carattere sia stato digitato quando un processo cerca di leggere, o che lo spazio disponibile sia stato riempito quando un processo cerca di scrivere. In questi casi, la upper-half e la lower-half devono coordinarsi, fermando un processo che sta cercando di scrivere finché lo spazio non diventa disponibile, o facendo ripartire un processo che stava aspettando dell'input non appena arriva il prossimo carattere.

12.3 Sincronizzazione di upper-half e lower-half

Ad una prima occhiata, la sincronizzazione tra le due parti, alta e bassa, del driver, sembra essere un'istanza del problema di sincronizzazione del "produttore/consumatore" che può essere risolto molto bene con i semafori. Le routine di output della upper-half producono caratteri che le routine di output della lower-half consumano. Ma c'è un problema aggiuntivo. L'input non dà alcun problema, poiché i processi utente che chiamano routine della upper-half possono aspettare (*wait* sul semaforo) che la lower-half "produca" caratteri, e le routine della lower-half possono segnalare (*signal* sul semaforo) ogni volta che leggono ("producono") un carattere. L'output non è, tuttavia, così semplice. Supponete che il processo della lower-half rimanga ad aspettare i caratteri che la upper-half produce. Poiché è probabile che la lower-half consumi i caratteri più lentamente di quanto la upper-half possa produrli, il buffer di output si riempirà completamente.

C'è un'altra ragione per non vedere le routine di output della upper-half del driver come il produttore e quelle della lower-half come il consumatore. Supponete che le routine di output della lower-half del driver vengano ridisegnate per essere guidate dagli interrupt (interrupt-driven), che è il caso normale quando si lavora con dispositivi asincroni. Le routine della lower-half, che operano durante gli interrupt, non possono aspettare (*wait*) che una routine della upper-half produca un carattere di output. (Questa restrizione, vi ricorderete, è una conseguenza della struttura degli interrupt: chiamare wait durante un interrupt potrebbe condurre ad una situazione in cui nessun processo sia pronto per l'esecuzione).

Come possono coordinarsi le due parti, inferiore e superiore, del driver, se la lower-half non può essere vista come il consumatore, che aspetta i caratteri prodotti dalla upper-half ? Sorprendentemente, i semafori possono facilmente risolvere il problema. Lo stratagemma consiste nel "girare attorno" alla chiamate di *wait*, cambiando gli obiettivi del semaforo. Invece di avere routine della lower-half che aspettano che la upper-half produca caratteri, il nostro progetto prevede che la upper-half aspetti che ci sia spazio nel buffer. In questo modo, la lower-half non consuma nulla: le routine di input della lower-half "producono" caratteri, e le routine di output della upper-half "producono" spazio nel buffer.

12.4 Dichiarazione del blocco di controllo (control block) e del buffer

Ogni dispositivo che viene utilizzato come tty deve avere la propria coppia di semafori e i propri buffer di input e di output. Tutti questi dati sono mantenuti in una struttura solitamente conosciuta come *blocco di controllo* (*control block*); c'è un control block per ogni dispositivo tty. Insieme ai buffer ed ai semafori, il control block contiene anche i parametri menzionati precedentemente. Sebbene ciò possa disorientare, essi sono simili ai parametri forniti dagli altri sistemi.

Il codice nel file *tty.h* contiene il codice C per la definizione del control block per i tty. Nel codice, la definizione è contenuta nelle struttura denominata *tty*.

```
/* tty.h */

#include <window.h>                /* definizioni delle finestre */

#define     OBMINSP           20          /* spazio minimo nel buffer prima */
#define     EBUFLEN           32          /* dimensione della coda dell'echo */
#define     TTYOPRIO          100         /* priorità dell'output del tty */
#define     TTYIPRIO          (TTYOPRIO+1)      /* priorità dell'input del tty */

/* costanti per le dimensioni */

#ifndef     Ntty
#define     Ntty        1           /* numero delle linee tty seriali */
#endif
#ifndef     IBUFLEN
#define     IBUFLEN     128         /* num. dei caratteri nella coda di input */
#endif
#ifndef     OBUFLEN
#define     OBUFLEN     64          /* num. caratteri nella coda di output */
#endif

/* constanti per i modi di utilizzo */
```

```
#define     IMRAW       'R'   /* modo raw => non fa nulla */
#define     IMCOOKED    'C'   /* modo cooked => editing di linea */
#define     IMCBREAK    'K'   /* effettua l'echo, etc, ma senza edit di linea */
#define     OMRAW       'R'   /* modo raw => caso normale */

struct      tty   {               /* control block per ogni linea del tty /
      int   ihead;                /* testa della coda (buffer) di input */
      int   itail;                /* coda della coda (buffer) di input */
      char  ibuff[IBUFLEN];       /* buffer di input per questa linea */
      int   icnt;                 /* elementi nel buffer */
      int   isem;                 /* semaforo di input */
      int   ohead;                /* testa della coda (buffer) di output */
      int   otail;                /* coda della coda (buffer) di output */
      char  obuff[OBUFLEN];       /* buffer di output per questa linea */
      int   ocnt;                 /* elementi nel buffer */
      int   osem;                 /* semaforo di output */
      int   odsend;               /* invii ritardati per lo spazio */
      int   ehead;                /* testa della coda (buffer) di echo */
      int   etail;                /* coda della coda (buffer) di echo */
      char  ebuff[EBUFLEN];       /* coda (buffer) di echo */
      int   ecnt;                 /* elementi nel buffer */
      char  imode;                /* modo: IMRAW, IMCBREAK, IMCOOKED  */
      Bool  iecho;                /* è attivo l'echo per l'input ? */
      Bool  ieback;               /* do erasing backspace on echo? */
      Bool  evis;           /* mostra i caratteri di controllo come ^X ? */
      Bool  ecrlf;                /* invia CR-LF per una nuova linea? */
      Bool  icrlf;                /* mappa '\r' in '\n' in fase di input? */
      Bool  ierase;         /* tenere conto del carattere di cancellazione */
      char  ierasec;              /* carattere di cancellazione (backspace) */
      Bool  ikill;          /* tenere conto del car.per cancellare una riga?*/
      char  ikillc;               /* carattere per cancellare una riga */
      int   icursor;              /* posizione corrente del cursore */
      Bool  oflow;                /* tenere conto di ostop/ostart? */
      Bool  oheld;                /* l'output attualmente viene tenuto? */
      char  ostop;                /* carattere che blocca l'output */
      char  ostart;               /* carattere che fa ripartire l'output */
      Bool  ocrlf;                /* stampa CR/LF al posto di LF ? */
      char  ifullc;         /* carattere da inviare quando l'input è pieno */
      int   dnum;           /* numero di dispostivo di questa finestra */
      int   oprocnum;       /* output server process id   */
      int   wstate;               /* window state (window)     */
                            /* input server process id (tty)*/
      int   seq;            /* sequence changed at creation    */
      int   colsiz;         /* dimensione colonna di finestra logica */
      int   rowsiz;         /* dimensione riga di finestra logica */
      char  attr;           /* attributi del carattere */
      CURSOR      curcur;   /* pos. corrente del cursore nella finestra */
      CURSOR      topleft;  /* angolo in alto a sinistra della finestra */
      CURSOR      botright; /* angolo in basso a destra della finestra */
      Bool  hasborder;      /* la finestra ha un bordo ? */
};

extern      struct      tty tty[];

#define     BACKSP      '\b'
#define     BELL        '\07'
#define     ATSIGN      '@'
#define     BLANK       ' '
#define     NEWLINE     '\n'
#define     RETURN      '\r'
#define     TAB         '\t'
#define     TABSTOP     8
#define     STOPCH      '\023'          /* control-S blocca l'output       */
```

```
#define     STRTCH      '\021'          /* control-Q fa ripartire l'output  */
#define     UPARROW     '^'             /* usually for visuals like ^X      */

/* speciali tasti funzione */

#define     SPECKEY     0x100       /* offset per gli speciali tasti funzione */
#define     FKEY        0x13b       /* F1             */
#define     CFKEY       0x15e       /* control-F1     */
#define     PSNAPK      0           /* offset per l'istantanea del processo  */
#define     TSNAPK      1           /* offset per l'istantanea del tty       */
#define     DSNAPK      2           /* offset per l'istantanea del disco     */


/* ttycontrol function codes */

#define     TCSETBRK    1           /* attiva BREAK in trasmissione */
#define     TCRSTBRK    2           /* disattiva BREAK in trasmissione */
#define     TCNEXTC     3           /* guardare avanti di 1 carattere   */
#define     TCMODER     4           /* imposta il modo di input a raw   */
#define     TCMODEC     5           /* imposta il modo di input a cooked */
#define     TCMODEK     6           /* imposta il modo di input a cbreak     */
#define     TCICHARS    8           /* restituisce il num. dei car. in input */
#define     TCECHO      9           /* attiva l'echo  */
#define     TCNOECHO    10          /* disattiva l'echo          */
#define     TFULLC      BELL        /* car. da scrivere se buffer pieno */

/* messaggi passati al processo di output */
#define     TMSGOK      0           /* tutto OK */
#define     TMSGEFUL    1           /* overflow del buffer di echo */


extern      int   kprintf();        /* formatted console print    */
extern      int   printf();         /* XON/XOFF console print     */
extern      int   wputcsr();        /* put cursor routine         */

extern      int   winofcur;         /* cur window of cursor       */
```

I componenti chiave della struttura *tty* sono un buffer di input, *ibuff*, un buffer di output, *obuff*, e un buffer di echo, *ebuff*. Ogni buffer è un vettore (gli esercizi discutono questa scelta). I puntatori head (testa) e tail (coda) puntano rispettivamente alla prossima locazione da riempire e alla prossima da svuotare nel vettore. I caratteri sono sempre inseriti in testa e prelevati dalla coda., indipendentemente da come fluiscano, se dalla upper-half alla lower-half o viceversa.. Il driver tratta ogni buffer come una lista circolare, con la locazione zero che segue l'ultima locazione. Inizialmente, la testa e la coda puntano entrambe alla locazione zero, ma non ci può essere confusione sul fatto che il buffer sia completamente pieno o completamente vuoto perché il totale dei caratteri è controllato dai semafori, *isem* e *osem*, come descritto sopra.

C'è un control block tty per dispositivo; essi sono mantenuti nel vettore *tty*, che è indicizzato secondo il numero secondario di dispositivo. Il programma di configurazione del sistema imposta la costante *Ntty* con il numero dei dispositivi tty. Esso inoltre assegna ad ogni dispositivo tty un numero secondario di dispositivo da 0 fino a *Ntty*-1 e pone il numero secondario di dispositivo nella device switch table. In PC-Xinu, i dispositivi tty aventi il numero secondario di dispositivo maggiore di zero sono riservati per le finestre sul video; l'architettura del sistema a finestre è descritto nel Capitolo 14. Sia le routine della lower-half, che quelle della upper-half utilizzano il numero secondario di dispositivo come indice nel vettore tty. Così il numero secondario di dispositivo costituisce un collegamento cruciale tra l'identificativo (id) del dispositivo e il control block associato con quel dispositivo.

Poiché le routine della upper-half devono conoscere il processo di output della lower-half per avvertirlo quando arriva un carattere in output, l'identificativo di quest'ultimo processo è mantenuto nel campo *oprocnum* della struttura tty. Similmente, il campo wstat è usato per contenere l'identificativo del processo di input della lower-half.

12.5 Routine di input della upper-half del tty

Le routine *ttygetc*, *ttyputc*, *ttyread*, e *ttywrite* costituiscono la base della upper-half del driver del tty. Esse corrispondono alle operazioni *getc*, *putc*, *read*, e *write* descritte nel Capitolo 9. La routine più semplice è *ttygetc*.

```
/* ttygetc.c - ttygetc */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  ttygetc  --  legge un carattere da un dispositivo tty
 *------------------------------------------------------------------------
 */
ttygetc(devptr)
struct      devsw *devptr;
{
      int         ps;
      char        ch;
      struct      tty   *iptr;

      disable(ps);                  /* mutua esclusione (accesso a devptr) */
      iptr = &tty[devptr->dvminor];       /* semaforo associato al buffer di  */
                                    /* input per quel dispositivo      */
      wait(iptr->isem);             /* aspetta per un carattere nel buffer   */
      ch = iptr->ibuff[iptr->itail++];    /* estrae il carattere dal buffer   */
      --iptr->icnt;                       /* un carattere in meno nel buffer  */
      if (iptr->itail   >= IBUFLEN)       /* buffer circolare: ritorna indietro*/
            iptr->itail = 0;              /* se si è giunti alla fine   */
      if (iptr->ieof && (iptr->ieofc == ch) )  /* carattere di fine file */
            ch = EOF;                           /* con controllo attivato */
      restore(ps);
      return(ch);
}
```

Quando viene chiamata, *ttygetc* per prima cosa recupera il numero secondario di dispositivo dalla device switch table e lo utilizza come indice nel vettore *tty* per localizzare il corretto control block. Quindi essa effettua una *wait* sul semaforo di input, *isem*, finché la lower-half non deposita un carattere nel buffer. Quando termina la *wait*, *ttygetch*: estrae il successivo carattere dal buffer di input; aggiorna il puntatore alla coda per renderlo pronto per le successive estrazioni; aggiorna il totale, *icnt*, dei caratteri del buffer; e termina.

Si ricordi che l'operazione di *read* è utilizzata per ottenere più di un carattere in una sola operazione. La routine del driver tty che implementa *read* è chiamata *ttyread*; essa è mostrata di seguito. *Ttyread* non è concettualmente più difficile di *ttygetc* - solo i dettagli della programmazione la fanno apparire più complessa.

```
/* ttyread.c - ttyread */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  ttyread  --  legge uno o più caratteri da un dispositivo tty
 *------------------------------------------------------------------------
 */
ttyread(devptr, buff, count)
```

```
struct devsw *devptr;
char *buff;
int count;
{
      register struct   tty *ttyp;
      int   ps;
      int   avail, nread;

      if ( count<0 )
            return(SYSERR);
      disable(ps);
      ttyp = &tty[devptr->dvminor];       /* recupera il puntatore al tty */
      avail = scount( ttyp->isem );
      if ( (count = (count==0 ? avail : count)) == 0) {
            restore(ps);
            return(0);
      }
      nread = count;
      if (count <= avail)
            readcopy(buff, ttyp, avail, count);
      else {
            if (avail > 0) {
            readcopy(buff, ttyp, avail, avail);
            buff += avail;
            count -=avail;
            }
            for ( ; count>0 ; count--)
                  *buff++ = ttygetc(devptr);
      }
      restore(ps);
      return(nread);
}
```

La semantica di come opera *read* sui dispositivi tty illustra come le primitive di I/O possano essere adattate ad molti dispositivi. Spesso, è utile leggere tutti i caratteri che attendono nella coda di input, anche se il programma chiamante non conosce quanti stanno aspettando (se ce ne sono). Per permettere tale operazione senza introdurre ulteriori primitive di I/O, il driver tty interpreta in modo insolito quella che altrimenti potrebbe essere considerata un'operazione illegale: essa interpreta una richiesta di "leggi zero caratteri" come una richiesta di "leggi tutti i caratteri che sono in attesa".

Il code in *ttyread* mostra come le richieste di lunghezza zero sono cambiate, dopo l'ingresso nella routine, in richieste dell'esatto numero di caratteri che sono in attesa, in base al valore corrente del semaforo di input, *isem*. Dopo che il caso speciale è stato trattato, *ttyread* continua procurandosi i caratteri e spostandoli nelle posizioni corrette. Se sono disponibili caratteri in numero sufficiente a soddisfare la richiesta, ttyread li copia direttamente nel buffer specificato dall'utente con *readcopy* e termina. Se l'utente ha richiesto più caratteri di quelli che sono in attesa, *ttyread* copia quelli che sono disponibili e chiama ripetutamente *ttygetc* per ottenere un carattere aggiuntivo alla volta finché la richiesta può essere soddisfatta. Il codice per *readcopy* è nel file *readcopy.c*:

```
/* readcopy.c - readcopy */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>

/*------------------------------------------------------------------------
 * readcopy  --  copia ad alta velocità dal buffer del tty al buffer dell'utente
 *------------------------------------------------------------------------
 */
readcopy(buff,ttyp,avail,count)
register char *buff;
```

```
struct tty *ttyp;
int avail,count;
{
      register char *qtail;         /* variabile per la copia */
      int   ct, i;

      i = ttyp->itail;
      qtail = &ttyp->ibuff[i];      /* indirizzo della coda */
      for ( ct=count; ct>0; ct-- ) {
            *buff++ = *qtail++;
            if ( ++i >= IBUFLEN ) {       /* coda circolare: se è alla fine  */
                  i=0;                    /* torna al 1° elemento            */
                  qtail = ttyp->ibuff;
            }
      }
      ttyp->itail = i;
      ttyp->icnt -= count;
      sreset(ttyp->isem,avail-count);
}
```

12.6 Routine di output della upper-half del tty

Le routine output della upper-half del tty sono quasi tanto semplici quanto le routine di input della upper-half. *Ttyputc*: aspetta che ci sia spazio nel buffer di output; deposita il carattere nella coda di output, *obuff*; incrementa il puntatore alla testa, *ohead*; e aggiorna il contatore degli elementi del buffer, *ocnt*.

```
/* ttyputc.c - ttyputc */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  ttyputc  --  scrive un carattere su un dispositivo tty
 *------------------------------------------------------------------------
 */
ttyputc(devptr, ch )
struct      devsw *devptr;
char  ch;
{
      struct      tty   *iptr;
      int   ps;

      iptr = &tty[devptr->dvminor];
      disable(ps);
      wait(iptr->osem);             /* aspetta che ci sia spazio nella coda */
      iptr->obuff[iptr->ohead++] = ch;
      ++iptr->ocnt;
      if (iptr->ohead   >= OBUFLEN)
            iptr->ohead = 0;
      restore(ps);
      sendn(iptr->oprocnum,TMSGOK);       /* sveglia il processo tty */
      return(OK);
}
```

Appena prima di terminare, *ttyputc* invia un messaggio al processo di output della lower-half del tty, il cui id è in *oprocnum*. Ciò garantisce che la lower-half verrà svegliato per trasferire il carattere appena inserito nel buffer. Se il processo della lower-half è pronto o corrente, esso riceverà il messaggio in qualsiasi momento finisca le operazioni di output correnti. Se è nello stato *RECEIVING* - aspettando un messaggio - riprenderà non appena avrà la possibilità di essere scelto dallo scheduler. *Ttyputc* usa *sendn*, che non

richiama lo scheduler quando il messaggio viene inviato, piuttosto che *send* che lo richiama. Si è fatto così per evitare costose operazioni di rischedulazione ad ogni carattere.

Il processo di output della lower-half è in esecuzione nel caso in cui ci siano caratteri nei buffer di output e di echo, così potrebbe non essere necessario inviargli un messaggio ogni volta che un carattere viene aggiunto al buffer. Inviare al processo un messaggio quando esso è già in esecuzione è una cosa innocua; perciò, inviare ciecamente un messaggio per ogni carattere è meno costoso di testare, se necessario. Inviare messaggi è l'unico modo con cui le routine di output della upper-half possono svegliare quelle della lower-half per iniziare il trasferimento. Le routine della upper-half non chiamano quelle della lower-half direttamente, né iniziano la trasmissione del carattere.

Il driver *tty* supporta inoltre trasferimenti di più byte (multiple-byte transfers), ovvero operazioni di write. La routine appropriata del driver è *ttywrite*. Essa copia caratteri nel buffer di output e fa partire il processo di output della lower-half. Per diminuire i tempi di gestione (overhead), *ttywrite* determina quanto spazio è disponibile nel buffer di output. Se non rimane sufficiente spazio, *ttywrite* copia i dati specificati nel buffer e termina. Altrimenti, essa riempie lo spazio disponibile e chiama *ttyputc* per aggiungere i caratteri rimanenti uno alla volta. I file *ttywrite.c* e *writcopy.c* contengono il codice.

```
/* ttywrite.c - ttywrite */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  ttywrite  --  scrive uno o più caratteri ad un dispositivo tty
 *------------------------------------------------------------------------
 */
ttywrite(devptr, buff, count)
struct      devsw *devptr;
char        *buff;
int         count;
{
      register struct tty *ttyp;
      int   avail;
      int   ps;

      if (count < 0)
            return(SYSERR);
      if (count == 0)
            return(0);
      disable(ps);
      ttyp = &tty[devptr->dvminor];
      avail = scount( ttyp->osem );
      if ( avail >= count) {
            writcopy(buff, ttyp, avail, count);
/* sveglia il processo di output della lower-half: ci sono caratteri ! */
            sendn(ttyp->oprocnum,TMSGOK);
      } else {                      /* non c'è abbastanza spazio */
            if (avail > 0) {        /* invia i caratteri finché c'è spazio */
                  writcopy(buff, ttyp, avail, avail);
                  sendn(ttyp->oprocnum,TMSGOK);
                  buff += avail;
                  count -= avail;
            }
            for (; count>0 ; count--)     /* invia i caratteri per cui non */
                  ttyputc(devptr, *buff++);     /* c'era spazio nel buffer */
      }
      restore(ps);
      return(savecount);
}
```

```
/* writcopy.c - writcopy */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  writcopy  --  copia ad alta velocità dal buffer dell'utente nel buffer tty
 *------------------------------------------------------------------------
 */
writcopy(buff, ttyp, avail, count)
register char *buff;
struct      tty *ttyp;
int         avail, count;
{
      register char     *qhead;
      int   ct, i;

      i = ttyp->ohead;
      qhead = &ttyp->obuff[i];
      for ( ct=count; ct>0; ct-- ) {
            *qhead++ = *buff++;
            if ( ++i >= OBUFLEN ) {       /* coda circolare: se in fondo     */
                  i=0;                    /* torna all'inizio                */
                  qhead = ttyp->obuff;
            }
      }
      ttyp->ocnt += count;
      ttyp->ohead = i;
/* reimposta il semaforo al valore corretto */
      sreset(ttyp->osem, avail-count);
}
```

12.7 Routine della lower-half del driver del tty

La lower-half del driver del tty compie il vero lavoro: effettua le reali operazioni di input e output e intercetta gli interrupt. Essa è composta da tre procedure: il server di output, *ttyoproc*; il server di input, *ttyiproc*; e la routine di servizio dell'interrupt per la gestione dell'input (da tastiera), *ttyiin*. Come prima procedura, consideriamo la routine del server di output, che si può trovare nel file *ttyoproc.c:*

```
/* ttyoproc.c - ttyoproc */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <vidio.h>

/*------------------------------------------------------------------------
*  ttyoproc  --  processo della lower-half del driver dl tty per l'output
*                della console
 *------------------------------------------------------------------------
 */
PROCESS     ttyoproc()
{
      register struct   tty   *iptr;
      int   ct;
      int   ps;
```

```
      char  ch;
      Bool  enl,onl;
      int   rcvchr();

      iptr = &tty[0];              /* puntatore alla struttura tty */
      onl = enl = FALSE;
      disable(ps);
      for (;;) {                   /* ciclo senza fine (processo)     */
            if (enl) {             /* deve inviare un linefeed        */
                  enl = FALSE;
                  wtty(NEWLINE);
                  continue;
            }
            /* cerca nel buffer di echo */
            if ( iptr->ecnt ) { /* c'è qualche carattere?  */
                  /* se sì, lo estrae e aggiorna la coda del buffer ... */
                  ch = iptr->ebuff[iptr->etail++];
                  --iptr->ecnt;    /* ... e il numero dei caratteri presenti */
                  if (iptr->etail  >= EBUFLEN) /* coda circolare: se si è */
                        iptr->etail = 0;       /* in fondo torna all'inizio */
                  if ( (ch==NEWLINE) && iptr->ecrlf ) {
                        enl = TRUE;
                        ch = RETURN;
                  }
                  wtty(ch);   /* scrive il carattere */
                  continue;
            }
            if (iptr->oheld) {
                  rcvchr();
            continue;
            }
            if (onl) {        /* deve inviare un linefeed        */
                  onl = FALSE;
                  wtty(NEWLINE);
                  continue;
            }
            /* cerca nel buffer di output */
            if ( (ct=iptr->ocnt) > 0 ) {     /* c'è qualche carattere? */
                  /* se sì, lo estrae e aggiorna la coda del buffer ... */
                  ch = iptr->obuff[iptr->otail++];
                  --iptr->ocnt;     /* ... e il numero dei caratteri presenti */
                  if (iptr->otail  >= OBUFLEN) /* coda circolare: se si è */
                        iptr->otail = 0;       /* in fondo torna all'inizio */
                  if ( ct < (OBUFLEN-OBMINSP) && iptr->odsend == 0 )
                        signal(iptr->osem);
                  else if ( ++(iptr->odsend) == OBMINSP ) {
                        iptr->odsend = 0;
                        signaln(iptr->osem, OBMINSP);
                  }
                  if ( (ch==NEWLINE) && iptr->ocrlf ) {
                        onl = TRUE;
                        ch = RETURN;
                  }
                  wtty(ch);
                  continue;
            }
            rcvchr();
      }
}

/*------------------------------------------------------------------------
 *  rcvchr  --  aspetta che arrivi un altro carattere
 *------------------------------------------------------------------------
```

```
 */
LOCAL rcvchr()
{
      struct      tty   *iiptr;

      if ( winofcur != 0 ) {
            iiptr = &tty[winofcur];
            wputcsr(iiptr,iiptr->curcur);
      }
      if ( receive() == TMSGEFUL ) {
            wtty(BELL);       /* buffer pieno: campanella d'allarme ! */
      }
}
```

Mentre leggete il codice ricordatevi che il processo *ttyoproc* viene creato quando il driver del tty è inizializzato, e le routine di output della upper-half inviano un messaggio ogni volta che un carattere da stampare è inserito in coda.

Il driver lavora come un ciclo infinito, funzionando finché rimane output da gestire. Il trattamento dell'output è chiaro. Il driver mostra un carattere dal buffer di echo, o un carattere dal buffer di output, o non fa nulla. *Ttyoproc* dà priorità maggiore ai caratteri che aspettano nel buffer di echo, *ebuff*. Se *ebuff* non è vuoto, *ttyoproc* preleva un carattere da esso e lo scrive sul video utilizzando la routine *wput*; altrimenti procede con la normale elaborazione.

La normale elaborazione dell'output consiste nel selezionare un carattere dal buffer di output, obuff, e scriverlo sullo schermo. Prima di fare ciò, *ttyoproc* controlla il parametro del tty *oheld* per vedere se l'output non sia stato fermato. Quando *ttyoproc* trova *oheld* impostato, aspetta un messaggio dalla upper-half senza inviare altri caratteri. Poiché ttyoproc non sarà svegliato finché non verrà ricevuto un messaggio, qualche altra routine deve eventualmente resettare *oheld* e riattivare il trattamento dell'output. Come vedremo, il gestore dell'input imposta oheld quando rileva il carattere di "stop" lo resetta quando rileva un qualunque altro carattere. Per convenzione, il carattere di stop è *Ctrl-S*; l'utente lo digita per sospendere l'output (p.es. per leggere qualcosa prima che, scorrendo sullo schermo, non la si possa più vedere). Quando un qualunque tasto digitato fa riprendere l'output, il carattere convenzionale di avvio è *Ctrl-Q*; digitandolo si ripristinerà la gestione dell'output senza considerare altri effetti. Solitamente, nè il carattere di stop nè quello di avvio vengono depositati nel buffer di input.

In aggiunta a tutto questo, *ttyoproc* prende in considerazione i parametri tty *ocrlf* e *ecrlf*. Quando *ecrlf* non è nullo, ciò vuol dire che si deve far corrispondere la combinazione *RETURN+NEWLINE* ad ogni carattere *NEWLINE* da stampare. Per scrivere il carattere *NEWLINE* extra che segue *RETURN*, *ttyoproc* usa il flag *enl* per determinare se deve scrivere *NEWLINE* dopo aver scritto *RETURN*. Analogamente, il flag *onl* controlla l'eventuale espansione del carattere *NEWLINE* nel buffer di output. Si noti che i *NEWLINE* ripetuti hanno la precedenza sui normali *NEWLINE*.

La lower-half può trovare il buffer vuoto se ha inviato al dispositivo l'ultimo carattere in attesa. Questo non è un errore, soltanto l'indicazione che il processo può aspettare per un messaggio che indichi che è stato generato altro output. Così quando esso non trova nulla da inviare, *ttyoproc* chiama la routine locale *recvchr*, che infine chiama *receive*. Inoltre, se il cursore risiede in una finestra differente, *recvchr* mette il cursore al posto giusto.

Ttyoproc inizialmente disattiva gli interrupt con una chiamata a *disable* e non li riattiva mai. La maggior parte dell'elaborazione coinvolge la manipolazione di buffer, che deve essere effettuata con gli interrupt disabilitati. Quando non ci sono altri caratteri da elaborare, *ttyoproc* rimane in attesa di un messaggio. Poiché *receive* cambia lo stato del processo di output portandolo a *RECEIVING* ed chiama *resched*, il context switch che ne segue potrebbe eventualmente riabilitare gli interrupt.

12.7.1 Watermark e segnal ritardate

Ttyoproc utilizza una tecnica chiamata *watermark processing* per minimizzare gli overhead nell'interazione tra upper-half e lower-half del driver. La tecnica è degna di commento perché è sia fondamentale che diffusa.

Per capire le motivazioni del watermark processing, supponete per un momento che la lower-half chiami *signal* ogni volta che rimuove un carattere dal buffer. Poiché il processore potrebbe generare caratteri più velocemente di quanto il processo di output della lower-half possa mostrarli (per esempio, nel caso in cui la

priorità del processo della lower-half fosse inferiore di quello che genera l'output), il buffer di output di solito rimane pieno con un processo in attesa sul semaforo di output. Quando *ttyoproc* rimuove un carattere, esso lo segnale al semaforo, facendo sì che il primo processo in attesa depositi il carattere e prosegua l'esecuzione. Poiché i programmi spesso scrivono più di un carattere alla volta, il processo che era in attesa ha un'altra probabilità di produrre rapidamente un altro carattere e finire con l'aspettare ancora al semaforo. Il problema è che il richiamare lo scheduler è abbastanza costoso; eseguirlo ad ogni carattere priva gli altri processi del tempo di CPU.

Per diminuire l'overhead di resched, *ttyoproc* funziona in due modi. Continua la normale elaborazione finché non trova il buffer riempito oltre il livello superiore; in quel momento, si porta in modo ritardato (delayed mode) e smette di inviare segnali al semaforo. Mentre si trova in delayed mode, conta il numero delle volte in cui avrebbe dovuto chiamare *signal*. Infine, quando il buffer si è "prosciugato" fino al livello basso, *ttyoproc* chiama *signal* per recuperare i segnali che aveva tralasciato. Ritardare quando il buffer è quasi pieno introduce isteresi,, perché non chiama lo scheduler finché un numero minimo di posizioni è disponibile nel buffer. Così il processo che stava generando output più andare in esecuzione per un po' prima che il buffer si riempia e la upper-half forzi l'esecuzione dello scheduler.

Nel codice, le costanti *OBMINSP* determina i due livelli, alto e basso. Quando rimane uno spazio minore di *OBMINSP* nel buffer di output, *ttyoproc* si porta in delayed mode e ritarda esattamente un tempo *OBMINSP* prima di tornare in modo normale.

12.7.2 Elaborazione dell'input nella lower-half

L'elaborazione dell'input è la parte più complessa del driver del tty perché essa include il codice per l'echo dei caratteri è l'editing di linea. La routine di input opera in uno dei tre modi: *raw*, *cbreak*, o *cooked*, come specificato dal campo imode nel control block del tty. Il modo raw, il più semplice dei tre, accumula i caratteri nel buffer di input *ibuff* senza ulteriore elaborazione. All'estremo opposto, il modo cooked: effettua l'echo dei caratteri; sospende o fa ripartire l'output; e accumula una linea completa prima di pasasarla alla routine della upper-half. Il modo cooked è il modo normale in cui il computer opera - esso tiene conto dei caratteri speciali che permettono ai dattilografi di modificare l'input cancellando i caratteri precedemente inseriti o cancellando l'intera linea. Il modo cbreak, che sta più o meno nel mezzo, tiene conto di tutti i caratteri di controllo, tranne che di quelli correlati con l'editing di linea; come il modo raw, esso consegna i caratteri alle routine della upper-half senza aspettare di avere una linea completa.

```
/* ttyiproc.c - ttyiproc, erase1, eputc, echoch */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <bios.h>
#include <butler.h>
#include <kbdio.h>

/*------------------------------------------------------------------------
 *  ttyiproc  --  processo per l'input dei caratteri della lower-half del tty
 *------------------------------------------------------------------------
 */
PROCESS ttyiproc()
{
register struct   tty   *iptr; /* puntatore al control block tty */
      register int    ch;
      int   ct,w;
      int   ps;

      disable(ps);
      for ( ;; ) {
            if ( (ch=kbdgetc()) == NOCH ) {
                  receive();
                  continue;
            }
            if ( ch >= SPECKEY ) {
```

```
                if ( ch == CFKEY+PSNAPK ) { /* istantanea del processo */
                      send(butlerpid,MSGPSNAP);
                      continue;
                }
                if ( ch == CFKEY+TSNAPK ) { /* istantanea del tty */
                      send(butlerpid,MSGTSNAP);
                      continue;
                }
                if ( ch == CFKEY+DSNAPK ) { /* istantanea del disco */
                      send(butlerpid,MSGDSNAP);
                      continue;
                }

                if ( ch >= FKEY && ch < FKEY+10 ) { /* finestra ?  */
/* F10 corrisponde al (tty) avente 0 come numero secondario di dispositivo 0 */
                      if ( (w=ch-FKEY+1) == 10 )
                            w = 0;
                      if ( w < Ntty ) {
                            iptr = &tty[w];
                            if ( iptr->wstate > 0 ) {
                                  winofcur = w;
                                  send(iptr->oprocnum,TMSGOK);
                            }
                      }
                      continue;
                }
          }
          iptr = &tty[winofcur]; /*recupera il puntatore all'elemento di tty*/
          if (iptr->imode == IMRAW) {   /* modo raw (semplice da gestire) */
                if ( iptr->icnt >= IBUFLEN )
                      continue;
                iptr->ibuff[iptr->ihead++] = ch;  /* correct fix would be */
                ++iptr->icnt;                     /* to make ibuff int array */
                if (iptr->ihead   >= IBUFLEN)   /* coda circolare */
                      iptr->ihead = 0;
                signal(iptr->isem);
                continue;
          }
          /* modi cbreak o cooked */
          if ( ch == RETURN && iptr->icrlf )
                ch = NEWLINE;
          if (iptr->oflow) {
                if ( ch == iptr->ostart ) {    /* fa ripartire l'output */
                      iptr->oheld = FALSE;
                      send(iptr->oprocnum,TMSGOK);
                      continue;
                }
                if ( ch == iptr->ostop ) {     /* blocca l'output */
                      iptr->oheld = TRUE;
                      continue;
                }
          }
          iptr->oheld = FALSE;
          if ( iptr->imode == IMCBREAK ) {     /* modo cbreak */
                if ( iptr->icnt >= IBUFLEN ) {
                      if (iptr->iecho)
                            eputc(iptr->ifullc,iptr);
                      continue;
                }
                iptr->ibuff[iptr->ihead++] = ch; /* estrae il carattere */
                ++iptr->icnt;
                if (iptr->ihead   >= IBUFLEN)
                      iptr->ihead = 0;
```

```
                echoch(ch,iptr);
                signal(iptr->isem);
                continue;
          }
          /* modo cooked */
          if ( ch == iptr->ikillc && iptr->ikill ) { /* cancella la linea */
                iptr->ihead -= iptr->icursor;
                iptr->icnt -= iptr->icursor;
                if ( iptr->ihead < 0 )
                      iptr->ihead += IBUFLEN;
                iptr->icursor = 0;
                if (iptr->iecho)
                      eputc(NEWLINE,iptr);
                continue;
          }
          if (ch == iptr->ierasec && iptr->ierase) {
                /* cancella un carattere */
                if (iptr->icursor > 0) {
                      --iptr->icursor;
                      if ( --(iptr->ihead) < 0 )
                            iptr->ihead += IBUFLEN;
                      --iptr->icnt;
                      erase1(iptr);
                }
                continue;
          }
          if ( ch==NEWLINE || ch==RETURN || iptr->icnt < IBUFLEN) {
                echoch(ch,iptr);
                iptr->ibuff[iptr->ihead++] = ch;
                ++iptr->icnt;
                if ( iptr->ihead >= IBUFLEN )
                      iptr->ihead = 0;
                ct = iptr->icursor + 1; /* +1 per \n o \r */
                iptr->icursor = 0;
                signaln(iptr->isem,ct);
                continue;
          }
          if ( iptr->icnt >= IBUFLEN-1) {
                if (iptr->iecho)
                      eputc(iptr->ifullc,iptr);
                continue;
          }
          echoch(ch,iptr);
          iptr->icursor++;
          iptr->ibuff[iptr->ihead++] = ch;
          ++iptr->icnt;
          if (iptr->ihead   >= IBUFLEN)
                iptr->ihead = 0;
      } /* fine del ciclo infinito */
}

/*------------------------------------------------------------------------
 *  erase1  -- cancella un carattere: è stato digitato il tasto di cancellazione
 *------------------------------------------------------------------------
 */
LOCAL erase1(iptr)
struct tty *iptr;
{
      char  ch;

      if ( iptr->iecho == 0 )
            return;
      ch = iptr->ibuff[iptr->ihead];
```

```
      if ( (ch<BLANK || ch==0177) && iptr->evis ) {
/* caratteri speciali (tipo ^X): bisogna cancellare DUE caratteri, non uno ! */
            eputc(BACKSP,iptr);
            if (iptr->ieback) {
                  eputc(BLANK,iptr);
                  eputc(BACKSP,iptr);
            }
      }
      eputc(BACKSP,iptr);
      if (iptr->ieback) {
            eputc(BLANK,iptr);
            eputc(BACKSP,iptr);
      }
}

/*------------------------------------------------------------------------
 *  echoch  --  mostra un carattere con l'opzione di visualizzazione
 *------------------------------------------------------------------------
 */
LOCAL echoch(ch, iptr)
char ch;                      /* carattere da stampare */
struct tty *iptr;             /* puntatore al control block */
{
      if ( iptr->iecho == 0 )
            return;                 /* niente da fare */
      if ( ch==NEWLINE || ch==RETURN || ch==TAB || ch==BELL ) {
            eputc(ch,iptr);
            return;
      }
      if ( (ch<BLANK || ch==0177) && iptr->evis ) {
/* caratteri speciali (tipo ^X): bisogna stampare DUE caratteri, non uno ! */
            eputc(UPARROW,iptr);
            eputc(ch+0100,iptr);    /* lo rende stampabile */
            return;
      }
      eputc(ch,iptr);
}

/*------------------------------------------------------------------------
 *  eputc  --  mette un carattere nella coda di echo
 *------------------------------------------------------------------------
 */
LOCAL eputc(ch,iptr)
char ch;
struct tty *iptr;
{
      if ( iptr->ecnt < EBUFLEN ) {     /* c'è spazio nel buffer */
            iptr->ebuff[iptr->ehead++] = ch;
            ++iptr->ecnt;
            if (iptr->ehead   >= EBUFLEN) /* coda circolare */
                  iptr->ehead = 0;
            send(iptr->oprocnum,TMSGOK);  /* sveglia il processo di output */
            return;
      }
      /* sveglialo a tutti i costi: non c'è più spazio nel buffer !!! */
      sendf(iptr->oprocnum,TMSGEFUL);
}
```

Si noti che *ttyiproc* è strutturata come un ciclo infinito, con gli interrupt disabilitati, in modo simile a *ttyoproc*. *Ttyiproc* inizialmente controlla se sono disponibili caratteri dalla tastiera chiamando la procedura di basso livello *kbdgetc*. Se non ci sono caratteri disponibili, la routine chiama *receive*, aspettando il messaggio che indica che un carattere è arrivato. Questo messaggio proviene dal gestore dell'interrupt della tastiera

ttyiin.
Certi tasti speciali sono gestiti prima della normale elaborazione dell'input. Un tasto speciale è identificato da un codice maggiore o uguale a *SPECKEY*, che è definita come 0x100 in *tty.h*. Fra i tasti speciale ci sono le varie combinazioni dei tasti funzione del PC. Tre tasti speciali, la cui gestione avviene qui, inviano messaggi al processo *butler* (maggiordomo), che è responsabile della visualizzazione di certe informazioni sullo stato di PC-Xinu. Un tasto visualizza un'istantanea di tutti i processi attivi, un altro lo stato corrente delle code del tty, e il terzo lo stato corrente delle richieste del disco.

I normali tasti funzione controllano la posizione del cursore in una delle finestre attive dello schermo. Lo scopo di questi tasti sarà descritto nel Capitolo 14. In particolare, l'input da tastiera apparirà nella finestra selezionata da questi tasti. Il numero della finestra è memorizzato nella variabile globale *winofcur*, che è usata come indice nel vettore *tty*. Quando non ci sono finestre attive, come dispositivo *CONSOLE* viene considerata la finestra corrente, rappresentata dal valore zero di *winofcur*.

Il modo raw è il più semplice da implementare e viene gestito solamente in una dozzina di linee di codice come si vede nel file *ttyiproc.c*. Nel modo raw, *ttyiproc* deposita il carattere in input nell'apposito buffer e segnale ciò al semaforo di input *isem*. Se non rimane spazio nel buffer, *ttyiproc* ignora il carattere.

12.7.3 Gestione dei modo cooked e cbreak

I modi cooked e cbreak condividono il codice che fa corrispondere *RETURN* a *NEWLINE* e che gestisce il controllo del flusso di caratteri in output. Il campo *oflow* del control block del tty determina se il driver debba gestire del tutto il controllo di flusso. Se è così, il driver sospende l'output impostando *oheld* quando riceve il carattere *ostop* e lo fa ripartire quando riceve un qualunque altro carattere. I caratteri *ostart* e *ostop* sono considerati caratteri di "controllo", così il driver non li mette nel buffer perché siano ricevuti dalla upper-half.

Il modo cbreak effettua l'echo dei caratteri e riporta gli overflow del buffer. Esso invia *ifullc* se il buffer di input non può contenere altri caratteri. Normalmente, *ifullc* è la "campanella" (bell) che fa' sì che il terminale suoni un allarme udibile; così una persona che sta digitando caratteri prima che siano letti dal computer sentirà l'allarme e si fermerà finché i caratteri non saranno stati letti. Cbreak chiama la routine locale *eputc* per inserire *ifullc* nel buffer di echo, e *echoch* per effettuare l'echo i caratteri che sono stati ricevuti.

Il modo cooked opera in modo molto simile al modo cbreak; rispetto a quest'ultimo, effettua anche l'editing di linea. Esso accumula le linee nel buffer di input, usando la variabile *icursor* per tenere il conto del caratteri della linea corrente. Quando arriva il carattere di cancellazione *ierasec*, *ttyiproc* decrementa *icursor* di uno e torna indietro, sul carattere precedente. Quando arriva il carattere di cancellazione della linea (line kill) *ikillc*, *ttyiproc* torna indietro su tutti i caratteri della linea corrente decrementando *icursor* fino a zero. In entrambi i casi, esso chiama la procedura *erase1* per cancellare i caratteri dallo schermo. Finalmente., quando arriva un carattere *NEWLINE* o *RETURN*, *ttyiproc* rende la linea disponibile per le routine della upper-half segnalando *icursor* volte al semaforo di input. Le procedure *echoch* e *erase1* controllano il flag *iecho* per determinare se il carattere debba essere stampato. Queste routine sono semplici ma meritano uno studio accurato.

12.8 Gestione dell'interrupt della tastiera

La gestione dell'interrupt della tastiera è eccezionalmente semplice dopo aver fatto tutto il lavoro rimanente nella routine *ttyiproc*. L'interrupt disptcher chiama *ttyiin* in seguito alla ricezione di un interrupt della tastiera. E' sufficiente per *ttyiin* semplicemente inviare un messaggio al processo *ttyiproc* per svegliarlo se era in attesa di un messaggio. L'inizializzazione pone l'identificativo del processo *ttyiproc* nel campo *wstat* della struttura *tty[0]*.

```
/* ttyiin.c - ttyiin */

#include <conf.h>
#include <kernel.h>
#include <tty.h>

/*------------------------------------------------------------------------
 *  ttyiin  -- routine dell'interrupt dell'input (lower-half del driver del tty)
 *------------------------------------------------------------------------
 */
```

```
INTPROC ttyiin()
{
    sendf(tty[0].wstate,TMSGOK);
}
```

12.9 Inizializzazione del control block del tty

La procedura *ttyinit*, mostrata di seguito, inizializza il control block del tty restituendo un puntatore all'elemento di *devtab* del dispositivo:

```
/* ttyinit.c - ttyinit */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <bios.h>
#include <kbdio.h>

/*------------------------------------------------------------------------
 *  ttyinit  --  inizializza i buffer e i modi per una linea tty
 *------------------------------------------------------------------------
 */
ttyinit(devptr)
struct      devsw *devptr;
{
      register struct   tty *iptr;
      char  *cp;
      int   pid;
      int   ttyoproc();
      int   ttyiproc();

      iptr = &tty[devptr->dvminor];     /* dispositivo */
      devptr->dvioblk = (char *)iptr;   /* riempie il control block tty */

      iptr->ihead = iptr->itail = 0;    /* svuota la coda di input */
      iptr->isem = screate(0);          /* car. letti fino ad ora = 0 */
      iptr->icnt = 0;
      iptr->osem = screate(OBUFLEN);     /* buffer completamente disponibile */
      iptr->odsend = 0;                /* invia un carattere finora ritardato */
      iptr->ohead = iptr->otail = 0;        /* coda di output vuota */
      iptr->ocnt = 0;
      iptr->ehead = iptr->etail = 0;        /* coda di echo vuota */
      iptr->ecnt = 0;
      iptr->imode = IMCOOKED;
      iptr->iecho = iptr->evis = TRUE;      /* mostra l'input della console  */
                        /* la console elabora i caratteri di cancellazione */
      iptr->ierase = iptr->ieback = TRUE;
      iptr->ierasec = BACKSP;             /*  usa ^h */
      iptr->ecrlf = iptr->icrlf = TRUE;   /* converte CR in CR+LF */
      iptr->ocrlf = iptr->oflow = TRUE;
      iptr->ikill = TRUE;         /* carattere cancellazione linea == @   */
      iptr->ikillc = ATSIGN;
      iptr->oheld = FALSE;
      iptr->ostart = STRTCH;
      iptr->ostop = STOPCH;
      iptr->icursor = 0;
      iptr->ifullc = TFULLC;
      iptr->curcur.row = 0;
      iptr->curcur.col = 0;

      /* ora avvia il nuovo processo tty per questo dispositivo */
```

```
      pid = create(ttyoproc,INITSTK,TTYOPRIO,"*TTYO*",0);
      if ( pid == SYSERR )
            kprintf("Can't create console output process\n");
      else
            ready(pid);
      iptr->oprocnum = pid;
      pid = create(ttyiproc,INITSTK,TTYIPRIO,"*TTYI*",0);
      if ( pid == SYSERR )
            kprintf("Can't create console input process\n");
      else
            ready(pid);
      iptr->wstate = pid;
}
```

Ttyinit inizializza il control block per il modo di default, cooked. *Ttyinit* crea i semafori di input e di output e reimposta il contatore degli elementi del buffer e i puntatori alla testa ed alla coda. Infine, crea i processi *ttyoproc* e *ttyiproc* e mette i loro identificatori nei campi *oprocnum* e *wstat*.

I parametri scelti per il control block del tty lavorano meglio con i PC che possono spostarsi indietro tra i caratteri sullo schermo e cancellarli. In particolare, il parametro *ieback* fa sì che il driver scriva tre caratteri, backspace-spazio-backspace, quando riceve il carattere di cancellazione, *ierasec*. Sullo schermo del PC, questo effettua la cancellazione dei caratteri come se l'utente fosse tornato indietro su di essi. Se riguardate la procedura *ttyiproc*, vedrete che essa accuratamente torna indietro del giusto numero di spazi, anche se l'utente cancella un carattere di controllo che viene mostrato come due caratteri stampabili.

12.10 Controllo del driver

Fino a questo punto abbiamo discusso delle operazioni di trasferimento dei dati della upper-half, come *read* e *write*. Un'altra operazione, *control* fornisce un sistema ai programmi utente per controllare i dispositivi e i relativi driver. Per esempio, il file *ttycntl.c* contiene un insieme di esempio di funzioni di controllo per il driver del dispositivo tty.

```
/* ttycntl.c - ttycntl */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  ttycntl  --  controlla un dispositivo tty impostandone gli attributi
 *------------------------------------------------------------------------
 */
SYSCALL ttycntl(devptr, func)
struct devsw *devptr;
int func;
{
      register struct   tty *ttyp;
      int   ps;
      int   c;

      disable(ps);
      ttyp = &tty[devptr->dvminor];
      c = OK;                               /* assume il caso migliore */
      switch ( func )   {
      case TCNEXTC:
            wait(ttyp->isem);
            c = ttyp->ibuff[ttyp->itail];
            signal(ttyp->isem);
            break;
      case TCMODER:
```

```
            ttyp->imode = IMRAW;
            break;
      case TCMODEC:
            ttyp->imode = IMCOOKED;
            break;
      case TCMODEK:
            ttyp->imode = IMCBREAK;
            break;
      case TCECHO:
            ttyp->iecho = TRUE;
            break;
      case TCNOECHO:
            ttyp->iecho = FALSE;
            break;
      case TCICHARS:
            c = scount(ttyp->isem);
            break;
      default:
            c = SYSERR;
      }
      restore(ps);
      return(c);
}
```

Alcune funzioni di controllo del tty cambiano i parametri nel control block del tty. Per esempio, i codici di funzione *TCMODER* e *TCMODEC* cambia il modo in, rispettivamente, quello raw e quello cooked; *TCECHO* e *TCECHONO* controllano l'echo dei caratteri. Altre funzioni come *TCICHARS* permettono agli utenti di richiedere al driver, in questo caso, di trovare quanti caratteri stanno aspettando nella coda di input.

I lettori attenti potrebbero aver notato che il parametro *addr* non è utilizzato dalla procedura *ttycntl*. Esso è stato dichiarato, tuttavia, perché la routine di I/O indipendente dal dispositivo *control* fornisce sempre tre argomenti quando chiama *ttycntl*. Omettere la dichiarazione degli argomenti potrebbe rendere il codice meno portabile e più difficile da capire.

12.11 Sommario

Un driver di dispositivo è composto da un insieme di procedure che controllano le periferiche. Le routine del driver sono divise in due parti: la upper-half che contiene le routine chiamate dai programmi utente e la lower-half che contiene le routine che gestiscono le attività specifiche del driver. Le due parti comunicano attraverso una struttura dati condivisa chiamata control block del dispositivo.

Il driver di esempio esaminato in questo capitolo riguarda il tty. Esso gestisce l'output verso lo schermo del PC e l'input dalla tastiera del PC. La upper-half del driver del tty contiene le routine che implementano le operazioni *read*, *write*, *getc*, *putc* e *control*; l'utente le chiama indirettamente tramite le procedure di I/O indipendenti dal dispositivo. Il processi di output della lower-half esegue un ciclo infinito, mostrando sullo schermo i caratteri prelevati dalla coda di output. Quando arriva un carattere in input, il gestore dell'interrupt della tastiera (che appartiene alla lower-half) sveglia il processo di input che deposita il carattere nella coda dei caratteri in arrivo, dove può essere ritrovata dalla upper-half. Il driver contiene inoltre una procedura di inizializzazione che riempie il control block del dispositivo e crea i processi di input e di output quando il sistema si avvia.

Per approfondire

I driver sono raramente descritti in dettaglio, perché essi dipendono dall'hardware e dai più alti livelli del sistema operativo. Una trattazione generale della gestione dei dispositivi può essere trovata in Freeman [1975], Calingaert [1982], e Habermann [1976]. Watson [1970] ha posto l'attenzione sui driver per i terminali.

Lo stile di base dell'interfaccia dei terminali usati qui, così come il nome "tty", sono stati presi dal sistema UNIX (Ritchie e Thompson [1974]).