12

An Example Device Driver

The previous chapter discussed high-level I/O operations and the device switch table, *devtab*, which forms the general framework linking interrupts, devices, and device driver routines. This chapter explores device driver routines for the *CONSOLE* device consisting of the PC keyboard and video display. Seeing an example will help you understand how device drivers operate and appreciate how the device switch framework eases the task of configuring devices and device drivers into the system.

12.1 The Device Type Tty

PC-Xinu uses the name *tty* to refer to conventional ‘‘computer terminals.’’ Each tty device has a keyboard capable of transmitting characters to the computer and an output device capable of displaying characters received from the computer. In the PC, the keyboard and video display circuitry are integrated into the system; other tty-type devices may be attached to the computer’s asynchronous communication lines. The ‘‘tty’’ terminology was borrowed from older systems that used Teletype devices as terminals.

In broad terms, the task of the tty device driver is to map operations like reading or writing characters to operations that transmit characters to the display and receive characters from the keyboard. In practice, the driver communicates with the BIOS, requesting it to send characters or receive them. To minimize interference between I/O and running processes, the driver creates and uses special-purpose processes to transmit characters to the screen and to receive characters from the keyboard. In addition, it coordinates requests for I/O with the speed of the device. The latter task is especially important for asynchronous interrupt-driven devices because character transmission times are often several orders of magnitude slower than processing speed, something few programmers appreciate until they write device drivers.

The tty drivers operate from parameters so they can be used in a variety of configurations. Several of the parameters control character ''echo.'' The PC operates in what is known as *full-duplex mode*. In this mode, the BIOS does not automatically display characters as the user types them; rather, the device driver transmits to the display every character received from the keyboard, so the user can see what has been typed. However, not all tty-type terminals need character echo. Those that operate in *half-duplex mode* display keystrokes automatically. If the device driver echoes the characters received from a half-duplex terminal, two copies appear on the display. So, the tty driver keeps a parameter that tells whether characters should echo or not. Being able to control character echo is also important for full-duplex devices since it permits turning off the echo feature during entry of sensitive information such as passwords. The tty driver keeps other parameters that tell it such things as whether or not to echo unprintable characters as a printable combination.

Most of the tty parameters will be obvious to anyone who has used a terminal, but readers who are unfamiliar with terminal hardware may puzzle over those that deal with moving onto new lines. The terminal display unit recognizes two unprintable characters that control cursor movement. *RETURN*, commonly called *carriage return*, moves the cursor left to the beginning of the current line (without spacing vertically). *NEWLINE*, sometimes called *line feed*, moves the cursor vertically downward one line (without moving horizontally). A display unit must receive both *NEWLINE* and *RETURN* to move to the start of the next line. Keyboards have separate keys that generate *RETURN* and *NEWLINE*, but naming is nonstandard so they might be labeled ''return,'' ''line feed,'' ''enter,'' ''newline,'' or ''end line.'' On the PC, the *RETURN* key is labeled ''Return,'' and *NEWLINE* is generated by a *Ctrl-J* combination; the ''Enter'' key also generates the *RETURN* code.

Although terminals send and interpret *NEWLINE* and *RETURN* separately, programs like to deal with a single end-of-line character for both input and output. In PC-Xinu, *NEWLINE* is the favored end-of-line character, denoted ''*n*'' in string and character constants. To simplify programming, the tty driver recognizes both *RETURN* and *NEWLINE* according to several parameters. A parameter denoted *icrlf* controls whether the driver maps *RETURN* to *NEWLINE* when received from the keyboard. Another parameter, *ocrlf*, controls whether the driver inserts *RETURN* in the output stream whenever a program writes *NEWLINE*.

12.2 Upper And Lower Halves Of The Device Driver

Like most device drivers, the tty driver routines can be partitioned into two sets: the *upper-half* and the *lower-half*. User processes call upper-half routines (indirectly through *devtab*) to read or write characters. Upper-half routines do not manipulate devices directly. Instead, they enqueue requests for transfer and rely on routines in the lower-half to perform transfers later. This partition, difficult to appreciate at first, lies at the heart of driver design – it is fundamental because it decouples normal processing from hardware characteristics.

The queue of transfer requests is the primary data structure that connects high-level calls to actions on the device. Each device has its own queue of requests, and the contents of elements on the queue depends on the device characteristics. Requests for devices like disks must specify the direction of transfer (read or write), the location of the data, and its length. Requests for character transfer are much simpler; usually, they only consist of the character itself.

Besides the queue of requests, the driver may need space for a *buffer*. Drivers use buffer space to record outgoing data from the time the user requests it be sent until the time the device receives it. They also use buffer space to record incoming data from the time the device deposits it until a user program requests it.

Buffers are important for several reasons. First, the driver can accept incoming data and place it in a buffer before a user process reads it. This is important for devices like a terminal where the user may start to type at any time. Second, devices like disks often transfer data in large blocks. The system must have a buffer large enough to hold all that the device transfers, even if the user only needs one character. Third, buffering permits the driver to perform I/O concurrently with user processes. When a user process writes data, the driver copies it into a buffer and allows the user process to continue executing, while it transfers the data from the buffer to the device.

The tty driver described here uses two circular character buffers per terminal, one for input and the other for output. Output operations deposit characters to be written in the output buffer and return to their caller. Meanwhile, the output process picks up the next character from the output buffer and sends the character to the display screen.

Input works the other way around. Whenever the keyboard receiver interrupts, signaling that it has received a character, the interrupt dispatcher calls the input interrupt routine. The input interrupt handler notifies the lower-half input process which reads the waiting character and deposits it in the circular input buffer. When a process calls an upper-half routine to read characters, the upper-half routine takes them from the input buffer, waiting for more input only if insufficient characters remain in the buffer.

Ideally, the two halves of a driver communicate only through the shared buffers:

> *Upper-half routines enqueue requests for data transfer or device control; they do not interact with devices directly. Lower-half routines transfer data from buffers or control devices; they do not interact with user programs directly.*

In practice, the two halves of the driver may need to do more than manipulate the shared data. For example, the upper-half may need to awaken the lower-half output process when it deposits output in the buffer. It may also happen that nothing has been typed when a process tries to read, or the available buffer space has been filled when a process tries to write. In such cases, the upper and lower halves must coordinate, stopping a process that is trying to write until space becomes available, or starting a process that is waiting for input as soon as the next character arrives.

12.3 Synchronization Of The Upper And Lower Halves

At first glance, synchronization between the upper and lower halves of a driver appears to be an instance of ‘‘producer/consumer’’ coordination that can be solved nicely with semaphores. The upper-half output routines produce characters that the lower-half output routines consume, while the lower-half input routines produce characters that the upper-half input routines consume. But there is an added twist. Input poses no problem because user processes that call the upper-half can *wait* for the lower-half to ‘‘produce’’ an input character, and lower-half routines can *signal* each time they read (‘‘produce’’) a character. Output is not as simple, however. Suppose the lower-half process waits for characters which the upper-half produces. Since the lower-half is likely to consume characters slower than the upper-half is able to produce them, the output buffer will become overrun.

There is another reason for not viewing the upper-half output driver as the producer and the lower-half as the consumer. Suppose the lower-half output driver is redesigned to be interrupt-driven, which would be the case when working with asynchronous devices. Lower-half routines, which operate at interrupt time, cannot *wait* for an upper-half routine to ‘‘produce’’ an output character. (This restriction, you will recall, is a consequence of the interrupt structure: calling *wait* at interrupt time might lead to a situation in which no process remains ready to run.)

How can the lower and upper halves coordinate if the lower-half cannot be viewed as the ‘‘consumer,’’ *waiting* for characters produced by the upper-half? Surprisingly, semaphores can easily solve the problem. The trick is to turn around the call to *wait* by changing the purpose of the semaphore. Instead of having a lower-half routine *wait* for the upper-half to produce characters, our design has the upper-half *wait* for space in the buffer. Thus, the lower-half never ‘‘consumes’’ anything: the lower-half input routine ‘‘produces’’ characters, and the lower-half output routine ‘‘produces’’ space in the buffer.

12.4 Control Block And Buffer Declarations

Each device being used as a tty must have its own pair of input and output semaphores and its own input and output buffers. All of this data is kept in a structure commonly called a *control block*; there is a tty control block for each tty device. Along with the buffers and semaphores, the control block also contains the parameters mentioned earlier. Although these may seem confusing, they are similar to the parameters provided on other systems.

The code in file *tty.h* contains the C code for defining the tty control block. In the code, the definition is given in the structure named *tty*.

```
/* tty.h */

#include <window.h>                     /* window definitions           */

#define OBMINSP         20              /* min space in buffer before   */
#define EBUFLEN         32              /* size of echo queue           */
#define TTYOPRIO        100             /* priority of tty output       */
#define TTYIPRIO        (TTYOPRIO+1)    /* priority of tty input        */

/* size constants */

#ifndef Ntty
#define Ntty            1               /* number of serial tty lines   */
#endif
#ifndef IBUFLEN
#define IBUFLEN         128             /* num. chars in input queue    */
#endif
#ifndef OBUFLEN
#define OBUFLEN         64              /* num. chars in output queue   */
#endif

/* mode constants */

#define IMRAW           'R'             /* raw mode => nothing done     */
#define IMCOOKED        'C'             /* cooked mode => line editing  */
#define IMCBREAK        'K'             /* honor echo, etc, no line edit*/
#define OMRAW           'R'             /* raw mode => normal processing*/

struct  tty     {                       /* tty line control block       */
        int     ihead;                  /* head of input queue          */
        int     itail;                  /* tail of input queue          */
        char    ibuff[IBUFLEN];         /* input buffer for this line   */
        int     icnt;
        int     isem;                   /* input semaphore              */
        int     ohead;                  /* head of output queue         */
        int     otail;                  /* tail of output queue         */
        char    obuff[OBUFLEN];         /* output buffer for this line  */
        int     ocnt;
        int     osem;                   /* output semaphore             */
        int     odsend;                 /* sends delayed for space      */
        int     ehead;                  /* head of echo queue           */
        int     etail;                  /* tail of echo queue           */
        char    ebuff[EBUFLEN];         /* echo queue                   */
        int     ecnt;
```

```
        char    imode;              /* IMRAW, IMCBREAK, IMCOOKED    */
        Bool    iecho;              /* is input echoed?             */
        Bool    ieback;             /* do erasing backspace on echo?*/
        Bool    evis;               /* echo control chars as ^X ?   */
        Bool    ecrlf;              /* echo CR-LF for newline?      */
        Bool    icrlf;              /* map '\r' to '\n' on input?   */
        Bool    ierase;             /* honor erase character?       */
        char    ierasec;            /* erase character (backspace)  */
        Bool    ikill;              /* honor line kill character?   */
        char    ikillc;             /* line kill character          */
        int     icursor;            /* current cursor position      */
        Bool    oflow;              /* honor ostop/ostart?          */
        Bool    oheld;              /* output currently being held? */
        char    ostop;              /* character that stops output  */
        char    ostart;             /* character that starts output */
        Bool    ocrlf;              /* output CR/LF for LF ?        */
        char    ifullc;             /* char to send when input full */
        int     dnum;               /* device number of this window */
        int     oprocnum;           /* output server process id     */
        int     wstate;             /* window state (window)        */
                                    /* input server process id (tty)*/
        int     seq;                /* sequence changed at creation */
        int     colsiz;             /* logical window column size   */
        int     rowsiz;             /* logical window row size      */
        char    attr;               /* character attributes         */
        CURSOR  curcur;             /* current cursor pos. in win   */
        CURSOR  topleft;            /* top left corner of window    */
        CURSOR  botright;           /* bottom right corner of window*/
        Bool    hasborder;          /* does window have a border    */
};
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 stops output       */
#define STRTCH  '\021'              /* control-Q restarts output    */
#define UPARROW '^'
```

```
/* special function keys */

#define SPECKEY 0x100                   /* special function key offset  */
#define FKEY    0x13b                   /* F1                           */
#define CFKEY   0x15e                   /* control-F1                   */
#define PSNAPK  0                       /* offset for process snapshot  */
#define TSNAPK  1                       /* offset for tty snapshot      */
#define DSNAPK  2                       /* offset for disk snapshot     */
/* ttycontrol function codes */

#define TCSETBRK        1               /* turn on BREAK in transmitter */
#define TCRSTBRK        2               /* turn off BREAK "       "     */
#define TCNEXTC         3               /* look ahead 1 character       */
#define TCMODER         4               /* set input mode to raw        */
#define TCMODEC         5               /* set input mode to cooked     */
#define TCMODEK         6               /* set input mode to cbreak     */
#define TCICHARS        8               /* return number of input chars */
#define TCECHO          9               /* turn on echo                 */
#define TCNOECHO        10              /* turn off echo                */
#define TFULLC          BELL            /* char to echo when buffer full*/

/* messages passed to output process */
#define TMSGOK          0               /* all OK                       */
#define TMSGEFUL        1               /* echo buffer overflow         */

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         */
```

The key components of the *tty* structure are an input buffer, *ibuff*, an output buffer, *obuff*, and an echo buffer, *ebuff*. Each buffer is an array (the exercises discuss this choice). Head and tail pointers point to the next location in the array to fill and the next location in the array to empty, respectively. Characters are always inserted at the head and taken from the tail, independent of whether they flow from the upper-half to the lower-half or vice versa. The driver treats each buffer as a circular list, with location zero following the last location. Initially, the head and tail both point to location zero, but there is never any confusion about whether the buffer is completely empty or completely full because the count of characters is controlled by semaphores, *isem* and *osem*, as discussed above.

There is one tty control block per device; they are kept in an array *tty*, which is indexed by the minor device number. The system configuration program sets constant *Ntty* to the number of tty devices. It also assigns each tty device a minor device number from 0 through *Ntty*-1 and places the minor device number in the device switch table. In PC-

Xinu, the tty devices with minor device numbers greater than zero are reserved for windows on the video screen; the window architecture is discussed in Chapter 14. Both the lower-half processes and driver routines in the upper-half use the minor device number as an index into the array *tty*. Thus, the minor device number forms a crucial link between the device id and the control block associated with that device.

Since the upper-half routines need to know which lower-half output process to notify when an output character arrives, the lower-half output process id is kept in the *oprocnum* field in the *tty* structure. Similarly, the *wstat* field is used to hold the process id of the lower-half input process.

12.5 Upper-Half Tty Input Routines

The routines *ttygetc*, *ttyputc*, *ttyread*, and *ttywrite* form the basis of the upper-half of the tty driver. They correspond to the operations *getc*, *putc*, *read*, and *write* described in Chapter 9. The simplest driver routine is *ttygetc*.

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

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

/*------------------------------------------------------------------------
 *  ttygetc  --  read one character from a tty device
 *------------------------------------------------------------------------
 */
ttygetc(devptr)
struct  devsw   *devptr;
{
        int     ps;
        char    ch;
        struct  tty   *iptr;

        disable(ps);
        iptr = &tty[devptr->dvminor];
        wait(iptr->isem);               /* wait for a character in buff */
        ch = iptr->ibuff[iptr->itail++];
        --iptr->icnt;
        if (iptr->itail >= IBUFLEN)
                iptr->itail = 0;
        restore(ps);
        return(ch);
}
```

When called, *ttygetc* first retrieves the minor device number from the device switch table and uses it as an index into array *tty* to locate the correct control block. It then executes *wait* on the input semaphore, *isem*, until the lower-half deposits a character in the buffer. When *wait* returns, *ttygetc* extracts the next character from the input buffer; updates the tail pointer to make it ready for subsequent extractions; updates the count, *icnt*, of characters in the buffer; and returns.

Recall that the *read* operation is used to obtain more than one character in a single operation. The tty driver routine that implements *read* is called *ttyread*; it is shown below. *Ttyread* is not conceptually more difficult than *ttygetc* – only the programming details make it appear complex.

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

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

/*------------------------------------------------------------------------
 *  ttyread  --  read one or more characters from a tty device
 *------------------------------------------------------------------------
 */
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];
        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);
}
```

The semantics of how *read* operates on tty devices illustrates how the I/O primitives can be adapted to a variety of devices. Often, it is useful to read all the characters waiting in the input queue, even though the calling program does not know how many (if any) are waiting. To permit such an operation without introducing additional I/O primitives, the tty driver applies an unusual interpretation to what might otherwise be considered an illegal operation: it interprets requests to *read* zero characters as requests to ''read all characters that are waiting.''

The code in *ttyread* shows how the zero length requests are changed upon entry into requests for exactly the number of characters that are waiting, based on the current count of the input semaphore, *isem*. After the special case has been handled, *ttyread* proceeds to obtain characters and move them to the specified location. If enough characters are available to satisfy the request, *ttyread* copies them directly to the user's buffer with *readcopy* and returns. If the user requests more characters than are waiting, *ttyread* copies out those that are available and calls *ttygetc* repeatedly to get one additional character at a time until the request has been satisfied. The code for *readcopy* is in file *readcopy.c*:

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

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

/*------------------------------------------------------------------------
 *  readcopy  --  high speed copy from tty buffer into user's buffer
 *------------------------------------------------------------------------
 */
readcopy(buff,ttyp,avail,count)
register char *buff;
struct tty *ttyp;
int avail,count;
{
        register char *qtail;           /* copy variable                */
        int     ct, i;

        i = ttyp->itail;
        qtail = &ttyp->ibuff[i];        /* address of tail              */
        for ( ct=count; ct>0; ct-- ) {
                *buff++ = *qtail++;
                if ( ++i >= IBUFLEN ) { /* wrap-around                  */
                        i=0;
                        qtail = ttyp->ibuff;
                }
        }
        ttyp->itail = i;
        ttyp->icnt -= count;
        sreset(ttyp->isem,avail-count);
}
```

12.6 Upper-Half Tty Output Routines

The upper-half output routines are almost as simple as the upper-half input routines. *Ttyputc* waits for space in the output buffer; deposits the character in the output queue, *obuff*; increments the head pointer, *ohead*; and updates the buffer count, *ocnt*.

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

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

/*------------------------------------------------------------------------
 *  ttyputc  --  write one character to a tty device
 *------------------------------------------------------------------------
 */
ttyputc(devptr, ch )
struct  devsw   *devptr;
char    ch;
{
        struct  tty   *iptr;
        int     ps;

        iptr = &tty[devptr->dvminor];
        disable(ps);
        wait(iptr->osem);               /* wait for space in queue      */
        iptr->obuff[iptr->ohead++] = ch;
        ++iptr->ocnt;
        if (iptr->ohead >= OBUFLEN)
                iptr->ohead = 0;
        restore(ps);
        sendn(iptr->oprocnum,TMSGOK);   /* wake up the tty process      */
        return(OK);
}
```

Just before it returns, *ttyputc* sends a message to the tty lower-half output process, whose process id is in *oprocnum*. This guarantees that the lower-half will awaken to transfer the character just inserted in the buffer. If the lower-half process is ready or current, it will receive the message whenever it finishes its current output operations. If it is *RECEIVING* – waiting for a message – it will resume as soon as it gets the opportunity in the scheduling priority. *Ttyputc* uses *sendn*, which does not reschedule when the message is sent, rather than *send* which does reschedule. This is done to avoid costly rescheduling at each character.

The lower-half output process runs as long as there are characters in the output and echo buffers, so it may not be necessary to send it a message every time a character is added to the buffer. Sending the process a message when it is already running is innocuous; therefore, blindly sending a message at every character is less expensive than testing whether it is necessary. Sending messages is the only way upper-half output routines awaken lower-half output routines to initiate transfers. Upper-half routines do not call lower-half routines directly, nor do they initiate character transmission.

The *tty* driver also supports multiple-byte transfers (*write*s). The appropriate driver routine is *ttywrite*. *Ttywrite* copies characters into the output buffer and starts the lower-half output process. To eliminate overhead, *ttywrite* determines how much space is available in the output buffer. If enough space remains, *ttywrite* copies the specified data into the buffer and returns. Otherwise, it fills the available space and then calls *ttyputc* to add the remaining characters one-by-one. Files *ttywrite.c* and *writcopy.c* contain the code.

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

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

/*------------------------------------------------------------------------
 *  ttywrite  --  write one or more characters to a tty device
 *------------------------------------------------------------------------
 */
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);
                sendn(ttyp->oprocnum,TMSGOK);
        } else {
                if (avail > 0) {
                        writcopy(buff, ttyp, avail, avail);
                        sendn(ttyp->oprocnum,TMSGOK);
                        buff += avail;
                        count -= avail;
                }
```

```
                for (; count>0 ; count--)
                        ttyputc(devptr, *buff++);
        }
        restore(ps);
        return(count);
}


/* writcopy.c - writcopy */

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

/*------------------------------------------------------------------------
 *  writcopy  --  high-speed copy from user's buffer into tty buffer
 *------------------------------------------------------------------------
 */
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 ) { /* wrap-around                  */
                        i=0;
                        qhead = ttyp->obuff;
                }
        }
        ttyp->ocnt += count;
        ttyp->ohead = i;
        sreset(ttyp->osem, avail-count);
}
```

12.7 Lower-Half Tty Driver Routines

The lower-half of the tty driver performs the real work of carrying out the physical input and output operations and fielding interrupts. It consists of three procedures: the output server, *ttyoproc*; the input server, *ttyiproc*; and the input (keyboard) interrupt routine, *ttyiin*. First, we will consider the output server routine, found in file *ttoproc.c*:

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

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

/*------------------------------------------------------------------------
 *  ttyoproc  --  lower-half tty device driver process for console output
 *------------------------------------------------------------------------
 */
PROCESS ttyoproc()
{
        register struct tty   *iptr;
        int     ct;
        int     ps;
        char    ch;
        Bool    enl,onl;
        int     rcvchr();

        iptr = &tty[0];                 /* pointer to tty structure     */
        onl = enl = FALSE;
        disable(ps);
        for (;;) {                      /* endless loop for process     */
                if (enl) {              /* must send linefeed           */
                        enl = FALSE;
                        wtty(NEWLINE);
                        continue;
                }
                /* look at the echo buffer */
                if ( iptr->ecnt ) {     /* any chars in echo buffer?    */
                        ch = iptr->ebuff[iptr->etail++];
                        --iptr->ecnt;
                        if (iptr->etail >= EBUFLEN)
                                iptr->etail = 0;
                        if ((ch==RETURN || ch==NEWLINE) && iptr->ecrlf) {
```

```
                                enl = TRUE;
                                ch = RETURN;
                        }
                        wtty(ch);
                        continue;
                }
                if (iptr->oheld) {
                        rcvchr();
                        continue;
                }
                if (onl) {              /* must send linefeed           */
                        onl = FALSE;
                        wtty(NEWLINE);
                        continue;
                }
                if ( (ct=iptr->ocnt) > 0 ) {
                        ch = iptr->obuff[iptr->otail++];
                        --iptr->ocnt;
                        if (iptr->otail >= OBUFLEN)
                                iptr->otail = 0;
                        if ( ct < (OBUFLEN-OBMINSP) && iptr->odsend == 0 )
                                signal(iptr->osem);
                        else if ( ++(iptr->odsend) == OBMINSP ) {
                                iptr->odsend = 0;
                                signaln(iptr->osem, OBMINSP);
                        }
                        if ((ch==RETURN || ch==NEWLINE) && iptr->ocrlf) {
                                onl = TRUE;
                                ch = RETURN;
                        }
                        wtty(ch);
                        continue;
                }
                rcvchr();
        }
}

/*-------------------------------------------------------------------------
 *  rcvchr  --  wait for another character to arrive
 *-------------------------------------------------------------------------
 */
LOCAL rcvchr()
{
        struct  tty     *iiptr;
```

```
        if ( winofcur != 0 ) {
                iiptr = &tty[winofcur];
                wputcsr(iiptr,iiptr->curcur);
        }
        if ( receive() == TMSGEFUL ) {
                wtty(BELL);
        }
}
```

Remember while you read the code, the *ttyoproc* process is created when the tty driver is initialized, and the upper-half tty output routines send it a message whenever an output character is enqueued.

The driver works as an infinite loop, working as long as output must be performed. Processing output is straight-forward. The driver either displays a character from the echo buffer, a character from the output buffer, or does nothing at all. *Ttyoproc* gives priority to characters waiting in the echo buffer, *ebuff*. If *ebuff* is nonempty, *ttyoproc* takes a character from it and writes the character to the video screen using the *wput* routine; otherwise, *ttyoproc* proceeds with normal processing.

Normal output processing consists of selecting a character from the output buffer, *obuff*, and writing the character to the video screen. Before doing so, *ttyoproc* checks the tty parameter *oheld* to see whether output has been stopped. When *ttyoproc* finds *oheld* set, it waits for a message from the upper-half without sending more characters. Because *ttyoproc* will not be awakened until a message is received, some other routine must eventually clear *oheld* and resume output processing. As we will see, the input handler sets *oheld* when it detects the ‘‘stop’’ character and clears *oheld* when it detects any other character. By convention, the stop character is *Ctrl-S*; the user types it to suspend output (e.g., to read something before it moves off the screen). While any typed character will resume output, the conventional start character is *Ctrl-Q*; typing it will resume output without taking any additional action. Usually, neither the start nor stop characters are deposited in the input buffer.

In addition to the processing mentioned above, *ttyoproc* honors the tty parameters *ocrlf* and *ecrlf*. When *ecrlf* is nonzero, it indicates that echoed *NEWLINE* characters should map to the combination *RETURN* plus *NEWLINE*. To write the extra *NEWLINE* character following the *RETURN*, *ttyoproc* maintains the flag *enl* to determine if it should output a *NEWLINE* after having written a *RETURN*. Similarly, the *onl* flag controls expansion of *NEWLINE* characters in the output buffer. Note that echoed *NEWLINE*s have priority over normal output *NEWLINE*s.

The lower-half may find the buffer empty if it has sent the last waiting character to the device. This is not an error, just an indication that the process can wait for a message to indicate that more output has been generated. So when it finds nothing to send, *ttyoproc* calls the local *recvchr* routine, which ultimately calls *receive*. If the cursor belongs in a different window, *Recvchr* also positions the cursor appropriately.

Ttyoproc initially turns interrupts off with a call to *disable* and never turns them back on. Much of the processing in *ttyoproc* involves manipulating buffers, which must be done with interrupts disabled. When there are no characters to be processed, *ttyoproc* is waiting for a message. Since *receive* changes the state of the output process to *RECEIVING* and reschedules, the resulting context switch will eventually reenable interrupts.

12.7.1 Watermarks And Delayed Signals

Ttyoproc uses a technique called *watermark processing* to minimize overhead in the interaction between upper and lower halves of the driver. The technique is worthy of comment because it is both fundamental and popular.

To understand the motivation for watermark processing, suppose for a moment that the lower-half called *signal* each time it removed a character from the buffer. Because the processor may generate characters much faster than the lower-half output process can display them (for example, if the priority of the lower-half process is lower than that of the processes generating the output), the output buffer usually remains full with a process waiting for the output semaphore. When *ttyoproc* removes a character, it signals the semaphore, causing the first waiting process to deposit a character and continue processing. Because programs often write more than one character at a time, the process that was waiting has a high probability of producing another character quickly and ending up waiting for the semaphore again. The problem is that rescheduling is relatively expensive; executing it on every output character deprives other ready processes of CPU time.

To lower the rescheduling overhead, *ttyoproc* runs in two modes. It continues processing normally until it finds the buffer filled beyond the high watermark; at which time, it switches to delayed mode and stops signaling the output semaphore. While in delayed mode, it accumulates the count of times it should have called *signal*. Finally, when the buffer has drained to the low watermark, *ttyoproc* calls *signal* to make up for the signals it has skipped. Delaying when the buffer is nearly full introduces hysteresis, because it does not reschedule until some minimum number of buffer positions are available. Thus, the process that was generating output can run for a while before the buffer fills and the upper-half forces rescheduling.

In the code, constant *OBMINSP* determines the high and low watermarks. When less than *OBMINSP* space remains in the output buffer, *ttyoproc* switches to delayed mode and delays exactly *OBMINSP* times before switching back to normal mode.

12.7.2 Lower-Half Input Processing

Input processing is the most complex part of the tty device driver because it includes code for character echo and line editing. The input routine operates in one of three modes: *raw*, *cbreak*, and *cooked*, as specified by the *imode* field in the tty control block. Raw mode, the simplest of the three, accumulates characters in the input buffer *ibuff* without further processing. At the opposite extreme, cooked mode does character echo; honors suspend or restart output; and accumulates complete lines before giving them to

the upper-half routines. Cooked mode is the usual mode in which computer systems operate – it honors special characters that permit the typist to edit input by erasing the previous character or killing the entire line. Cbreak mode, something in between, honors all control characters except those related to line editing; like raw mode, it delivers characters to the upper-half routines without waiting for a complete line.

```
/* 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  --  lower-half tty device process for input characters
 *------------------------------------------------------------------------
 */
PROCESS ttyiproc()
{
        register struct tty     *iptr;  /* pointer to tty block         */
        register int    ch;
        int     ct,w;
        int     ps;

        disable(ps);
        for ( ;; ) {
                if ( (ch=kbdgetc()) == NOCH ) {
                        receive();
                        continue;
                }
                if ( ch >= SPECKEY ) {
                        if ( ch == CFKEY+PSNAPK ) { /* process snapshot */
                                send(butlerpid,MSGPSNAP);
                                continue;
                        }
                        if ( ch == CFKEY+TSNAPK ) { /* tty snapshot     */
                                send(butlerpid,MSGTSNAP);
                                continue;
                        }
                        if ( ch == CFKEY+DSNAPK ) { /* disk snapshot    */
                                send(butlerpid,MSGDSNAP);
```

```
                        continue;
                }
                if ( ch >= FKEY && ch < FKEY+10 ) { /* window?  */
                        /* F10 maps to minor device 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];  /* get pointer to tty entry     */
if (iptr->imode == IMRAW) {
        if ( iptr->icnt >= IBUFLEN )
                continue;
        iptr->ibuff[iptr->ihead++] = ch;
        ++iptr->icnt;
        if (iptr->ihead >= IBUFLEN)
                iptr->ihead = 0;
        signal(iptr->isem);
        continue;
}
/* cbreak | cooked mode */
if ( ch == RETURN && iptr->icrlf )
        ch = NEWLINE;
if (iptr->oflow) {
        if (ch == iptr->ostart) {
                iptr->oheld = FALSE;
                send(iptr->oprocnum,TMSGOK);
                continue;
        }
        if (ch == iptr->ostop) {
                iptr->oheld = TRUE;
                continue;
        }
}
iptr->oheld = FALSE;
if (iptr->imode == IMCBREAK) {          /* cbreak mode  */
```

```
                if ( iptr->icnt >= IBUFLEN ) {
                        if (iptr->iecho)
                                eputc(iptr->ifullc,iptr);
                        continue;
                }
                iptr->ibuff[iptr->ihead++] = ch;
                ++iptr->icnt;
                if (iptr->ihead >= IBUFLEN)
                        iptr->ihead = 0;
                echoch(ch,iptr);
                signal(iptr->isem);
                continue;
        }
        /* cooked mode  */
        if (ch == iptr->ikillc && iptr->ikill) {
                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) {
                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 for \n or \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;
        } /* end of forever loop */
}

/*-------------------------------------------------------------------------
 *  erase1  --  erase one character honoring erasing backspace
 *-------------------------------------------------------------------------
 */
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 ) {
                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  --  echo a character with visual option
 *-------------------------------------------------------------------------
```

```
 */
LOCAL echoch(ch, iptr)
char ch;                                /* character to echo            */
struct tty *iptr;                       /* ptr to I/O block             */
{
        if ( iptr->iecho == 0 )
                return;                 /* nothing to do                */
        if ( ch==NEWLINE || ch==RETURN || ch==TAB || ch==BELL ) {
                eputc(ch,iptr);
                return;
        }
        if ( (ch<BLANK || ch==0177) && iptr->evis ) {
                eputc(UPARROW,iptr);
                eputc(ch+0100,iptr);    /* make it printable    */
                return;
        }
        eputc(ch,iptr);
}

/*-------------------------------------------------------------------------
 *  eputc  --  put one character in the echo queue
 *-------------------------------------------------------------------------
 */
LOCAL eputc(ch,iptr)
char ch;
struct tty *iptr;
{
        if ( iptr->ecnt < EBUFLEN ) {
                iptr->ebuff[iptr->ehead++] = ch;
                ++iptr->ecnt;
                if (iptr->ehead >= EBUFLEN)
                        iptr->ehead = 0;
                send(iptr->oprocnum,TMSGOK);
                return;
        }
        sendf(iptr->oprocnum,TMSGEFUL); /* wake it up!!! */
}
```

Note that *ttyiproc* is structured as an infinite loop, with interrupts disabled, similar to *ttyoproc*. *Ttyiproc* first checks to see if there are characters available from the keyboard by calling the low-level *kbdgetc*. If no characters are available, the routine calls *receive*, waiting for a message that a character has arrived. This message comes from the keyboard interrupt handler *ttyiin*.

Certain special keys are trapped before normal input processing. A special key is identified by a key code greater than or equal to *SPECKEY*, which is defined to be 0x100 in *tty.h*. Among the special keys are the various combinations of PC function keys. Three special keys, which receive treatment here, send messages to the *butler* process, which is responsible for displaying certain information about the state of PC-Xinu. One key displays a snapshot of all active processes, another displays the current status of tty queues, and a third displays the current status of disk requests.

Ordinary function keys control the location of the cursor in one of the active screen windows. The purpose of these keys will be described in Chapter 14. In particular, input from the keyboard will appear in the window selected by these keys. The window number is stored in the global variable *winofcur*, which is used as an index into the *tty* array. When there are no windows active, the *CONSOLE* device is considered the current window, represented by a zero value for *winofcur*.

Raw mode is the simplest to implement and accounts for only a dozen lines of code as shown in file *ttyiproc.c*. In raw mode, *ttyiproc* deposits the input character in the input buffer and signals the input semaphore *isem*. If no space remains in the buffer, *ttyiproc* throws the character away.

12.7.3 Cooked Mode And Cbreak Mode Processing

Cooked and cbreak mode share code that maps *RETURN* to *NEWLINE* and handles output flow control. Field *oflow* of the tty control block determines whether the driver honors flow control at all. If it does, the driver suspends output by setting *oheld* when it receives character *ostop* and restarts output when it receives any other character. Characters *ostart* and *ostop* are considered ‘‘control’’ characters, so the driver does not place them in the buffer for the upper-half to receive.

Cbreak mode performs character echo and reports buffer overflow. It sends *ifullc* if the input buffer *ibuff* cannot hold more characters. Normally, *ifullc* is a ‘‘bell’’ that causes the terminal to sound an audible alarm; thus, a person who is typing characters before they have been read by the computer will hear the alarm and stop typing until characters have been read. Cbreak calls local routines *eputc* to place *ifullc* in the echo buffer, and *echoch* to echo the character that has been received.

Cooked mode operates much like cbreak mode except that it also performs line editing. It accumulates lines in the input buffer, using variable *icursor* to keep a count of the characters on the current line. When the erase character *ierasec* arrives, *ttyiproc* decrements *icursor* by one and backs up over the previous character. When the line kill character *ikillc* arrives, *ttyiproc* backs over all characters on the current line by decrementing *icursor* to zero. In either case, it calls procedure *erase1* to obliterate the characters from the display. Finally, when a *NEWLINE* or *RETURN* character arrives, *ttyiproc* makes the line available to the upper-half routines by signaling the input semaphore *icursor* times. Procedures *echoch* and *erase1* inspect the *iecho* flag to determine if characters are to be echoed. These routines are simple but deserve careful study.

12.8 Keyboard Interrupt Handling

Keyboard interrupt handling is exceptionally simple after having done all the remaining work in the *ttyiproc* routine. The interrupt dispatcher calls *ttyiin* upon receipt of a keyboard interrupt. It is only necessary for *ttyiin* to send a message to the *ttyiproc* process to wake it up if it is waiting for a message. Initialization puts the process id of *ttyiproc* in the *wstat* field of the *tty*[0] structure.

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

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

/*------------------------------------------------------------------------
 *  ttyiin  --  lower-half tty device driver for input interrupts
 *------------------------------------------------------------------------
 */
INTPROC ttyiin()
{
        send(tty[0].wstate,TMSGOK);
}
```

12.9 Tty Control Block Initialization

Procedure *ttyinit*, shown below, initializes the tty control block given a pointer to the *devtab* entry for the device:

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

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

/*------------------------------------------------------------------------
 *  ttyinit  --  initialize buffers and modes for a tty line
 *------------------------------------------------------------------------
 */
ttyinit(devptr)
```

```
        struct  devsw   *devptr;
{
        register struct tty *iptr;
        char    *cp;
        int     pid;
        int     ttyoproc();
        int     ttyiproc();

        iptr = &tty[devptr->dvminor];
        devptr->dvioblk = (char *) iptr;        /* fill tty control blk */

        iptr->ihead = iptr->itail = 0;          /* empty input queue    */
        iptr->isem = screate(0);                /* chars. read so far=0 */
        iptr->icnt = 0;
        iptr->osem = screate(OBUFLEN);          /* buffer available=all */
        iptr->odsend = 0;                       /* sends delayed so far */
        iptr->ohead = iptr->otail = 0;          /* output queue empty   */
        iptr->ocnt = 0;
        iptr->ehead = iptr->etail = 0;          /* echo queue empty     */
        iptr->ecnt = 0;
        iptr->imode = IMCOOKED;
        iptr->iecho = iptr->evis = TRUE;        /* echo console input   */
        iptr->ierase = iptr->ieback = TRUE;/* console honors erase      */
        iptr->ierasec = BACKSP;                 /*  using ^h            */
        iptr->ecrlf = iptr->icrlf = TRUE;       /* map RETURN on input  */
        iptr->ocrlf = iptr->oflow = TRUE;
        iptr->ikill = TRUE;             /* set line kill == @           */
        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;

        /* now set up new tty process for this device */
        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 initializes the control block for the default cooked mode. *Ttyinit* creates the input and output semaphores and resets the buffer count and head and tail pointers. Finally, it creates the *ttyoproc* and *ttyiproc* processes and puts their process ids in the *oprocnum* and *wstat* fields.

The chosen *tty* control block parameters work best for the PC which can backspace over characters on the display and erase them. In particular, the parameter *ieback* causes the driver to echo three characters, backspace-space-backspace, when it receives the erase character, *ierasec*. On the PC screen this gives the effect of erasing characters as the user backs over them. If you look again at procedure *ttyiproc*, you will see that it carefully backs up the correct number of spaces, even if the user erases a control character that is displayed as two printable characters.

12.10 Device Driver Control

So far, we have discussed the upper-half data transfer operations like *read* and *write*. Another operation, *control*, provides a way for user programs to control devices and device drivers. For example, the file *ttycntl.c* contains a sample set of control functions for the tty device driver:

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

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

/*------------------------------------------------------------------------
 *  ttycntl  --  control a tty device by setting modes
 *------------------------------------------------------------------------
 */
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 the best              */
        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);
}
```

Some tty control functions change parameters in the tty control block. For example, function codes *TCMODER* and *TCMODEC* switch between raw and cooked modes; *TCECHO* and *TCECHONO* control character echo. Other functions like *TCICHARS* allow the user to query the driver, in this case, to find how many characters are waiting in the input queue.

Observant readers may have noticed that parameter *addr* is not used by procedure *ttycntl*. It has been declared, however, because the device-independent I/O routine *control* always provides three arguments when calling *ttycntl*. Omitting the argument declaration would make the code less portable and more difficult to understand.

12.11 Summary

A device driver consists of a set of procedures that control peripheral hardware devices. The driver routines are partitioned into two halves: an upper-half that contains the routines called from user programs and a lower-half that contains routines that handle device-specific activities. The two halves communicate through a shared data structure called the device control block.

The example device driver examined in this chapter is referred to as a tty driver. It manages output to the PC display screen and input from the PC keyboard. The tty driver upper-half contains routines that implement *read*, *write*, *getc*, *putc*, and *control* operations; a user calls them indirectly with the device-independent I/O procedures. The lower-half output process executes an infinite loop, displaying characters from the output queue. When an input character arrives, the lower-half keyboard interrupt handler awakens the input process which deposits the character in the queue of incoming characters, where it can be retrieved by the upper-half. The driver also contains an initialization procedure that fills in the device control block and creates the input and output processes when the system starts.

FOR FURTHER STUDY

Device drivers are seldom described in detail because they depend on the hardware and higher-levels of the operating system. A general discussion of device management can be found in Freeman [1975], Calingaert [1982], and Habermann [1976]. Watson [1970] focuses on drivers for terminal devices.

The basic style of the terminal interface used here, as well as the name ‘‘tty,’’ have been taken from the UNIX system (Ritchie and Thompson [1974]).

EXERCISES

12.1 Predict what would happen if two processes executed *ttyread* concurrently when both requested a large number of characters. Experiment and see what happens.

12.2 Making the input and output buffers arrays simplifies programming but may introduce additional overhead at run-time. Rewrite the tty driver routines to use pointers instead of array subscripts. Can you measure a change in performance?

12.3 The lower-half tty output process stops signaling the output semaphore when less than *OBMINSP* remains. Can the buffer ever be filled completely?

12.4 Explain how more than *OBMINSP* positions can be free in the output buffer immediately after *ttyoproc* switches back to normal mode from delayed mode. What is the maximum number of free positions? The minimum?

12.5 Suppose *ttyoproc* switched to delayed mode when the buffer was full and switched back to normal mode when it contained less than *OBMINSP*. Would the change result in more or fewer reschedules?

12.6 Multi-level drivers are often needed on systems that have more than one type of output devices, including asynchronous serial line devices, all of which connect to terminals that should act identically. Explore the code for a system that uses heterogeneous serial line hardware. Identify the low-level and middle-level drivers.

12.7 A user observes that his process, which uses *ttywrite*, writes a different sequence of characters when run on a system with a low output baud rate than when run on a system with a high output baud rate. The problem occurs at high baud rates when other processes call *ttywrite* concurrently. Can you explain the problem?

12.8 Rewrite *ttywrite* to correct the error referred to in the previous question.

12.9 *Ttycntl* handles changes of mode poorly because it does not reset the cursor or buffer pointers. Rewrite the code to improve it. What happens to partially entered lines when changing from cooked to raw mode?

12.10 Consider the following deadlock: processes are waiting for space in the output buffer; *ttyoproc* is in delayed mode so, although space remains, it has not signaled the output semaphore; and device output interrupts are disabled so the driver will not awaken to signal the output semaphore. If the processes waiting for buffer space could execute, they would restart interrupts. If the device interrupts were enabled, they would start output and signal the semaphore. Can this occur?

12.11 When connecting two computers, it is useful to have flow control in both directions. Modify the tty driver to include ‘‘tandem’’ flow control.

12.12 One design alternative to speed up the *ttyread* routine would be to assign the integer value *iptr->icnt* to *avail* instead of calling *scount* to assign it the count of semaphore *iptr->isem* to *avail*. Show that with this change it is possible for the input buffer to be corrupted. What is the exact relationship between *iptr->icnt* and the count of *iptr->isem*?

12.13 *Ttyinit* sets the priority of the keyboard input process higher than that of the screen output process. Explain this choice. Conduct an experiment to determine the effect of changing the keyboard input process priority to a value lower than the screen output process.

12.14 The priority (*TTYOPRIO*) of the process *ttyoproc* is set by *ttyinit* to be higher than that of an ordinary process (*INITPRIO*). What policy does this implement? Experiment with changing the priority of *ttyoproc* to a value less than typical user processes.

12.15 Input characters from terminals often come in spurts: when one terminal input is idle, another may be very busy. Some systems employ a pool of small input buffers shared among all the tty devices. Input from any one of the terminals will be deposited in a buffer allocated from the pool; several of these buffers may be linked together to contain a stream of input characters for the particular terminal, and only those tty devices experiencing activity will use the buffers. Buffers no longer in use are returned to the pool. Implement this input buffer scheme in PC-Xinu, and comment on its effectiveness.

12.16 Conduct experiments to see whether watermark processing has an effect on system performance.

12.17 Replace the process-driven keyboard input with direct interrupt-driven input from the PC keyboard port.

12.18 Add support for the PC's asynchronous communication lines, treating them as tty devices.