14

Window Management

Now that we have all the elements necessary to create and run concurrent processes, we have the problem of displaying the output of these processes in a sensible way on the PC screen. In this chapter we show how to divide the screen into nonoverlapping, rectangular ‘‘windows,’’ each behaving as a single tty device. Since a process may write characters to any tty device, windows make it possible to show the results of different processes in physically separate regions of the screen.

14.1 Windows As Pseudo-Devices

From a low-level point of view, writing a character to a window involves two steps: positioning the cursor in the window, and displaying the character at the cursor position. Both of these functions can be performed by BIOS calls. But requiring a process to manage the details of cursor positioning conflicts with our goals of flexibility, simplicity, and generality related to device independent I/O. Writing a character to a window should appear no different to the process as writing a character to the *CONSOLE* device or even to a disk file.

The idea is to treat a window as a *device*, even though it is physically controlled by the same video display hardware and BIOS software as the *CONSOLE*. Once a window is given status as a device, an operation such as *putc* will be mapped into appropriate low-level operations of cursor positioning and character display. Since these low-level operations occur within the device driver, their details will be hidden from the caller. Because a window device does not correspond directly to hardware, it is called a *pseudo-device*.

If a window is a device, where does it appear in the device switch table? One possibility is to have four window devices, each corresponding to a quadrant of the video screen. This has the advantage of simplicity, but prevents a user process from choosing the size and location of the window. Our design establishes a number of window

pseudo-devices, each of which can be reconfigured by a process at run-time to correspond to any rectangular screen area. *Open*ing a window connects an executing process to a free pseudo-device slot, initializes the device data structures, creates an output server process, and returns the device number of the slot for use with the high-level I/O operations. Operations *putc* and *write* display characters in the window, advancing the cursor position and scrolling the window when it becomes full. The *getc* and *read* operations obtain characters from the keyboard in the same way as the *CONSOLE* device, but only those characters which are typed when the cursor is in the window. Finally, *close* clears the window and detaches the running process from the pseudo-device, killing the server process and marking the pseudo-device as available.

14.2 Window-Specific Fields In The tty Structure

The drivers for window devices are very similar to those for tty devices, except that they contain code for such window-specific operations as positioning the cursor, drawing a border around the window, and scrolling the window area. In fact, the window device software uses the same *tty* structure as the *CONSOLE* device. There is a small number of fields in this structure that pertain solely to window operations and that are ignored in the tty device driver discussed in Chapter 12. The reader should review the contents of file *tty.h* in the following discussion.

14.2.1 The CURSOR Type

A window's location and size are determined by specifying the coordinates of the upper-left and lower-right corners of the window on the screen. Screen coordinates are given as a pair

$$(\text{column}, \text{row})$$

of integers, with $(0, 0)$ corresponding to the upper-left corner of the screen. It is convenient to package this pair in a structure definition, so that a screen location can be passed as a unit rather than as two integers. This structure type is called *CURSOR* and is defined in file *window.h*.

```
/* window.h - window definitions */

#define LWFREE          -1              /* window is free               */
#define LWLIMBO         0               /* window is in limbo           */
#define LWALLOC         1               /* window is allocated          */

#define BW              0x07            /* white on black attributes    */

#define TLBORDER        0xc9
```

```
#define TRBORDER        0xbb
#define BLBORDER        0xc8
#define BRBORDER        0xbc
#define HBORDER         0xcd
#define VBORDER         0xba

#define BORDER          0x100           /* attribute bit for border     */

/* character display screen - 25 rows x 80 columns */
#define G_ROWS  25
#define G_COLS  80

/* cursor position */
typedef struct  {
        unsigned  char  col;
        unsigned  char  row;
} CURSOR;
```

Since the column and row coordinates are limited in practice by the screen size of 80 columns and 25 rows on a typical PC, the coordinates can be stored in fields of type *unsigned char*.

The window size and location are determined by the *tty* structure variables *topleft* and *botright*, which are the coordinates of the top-left and bottom-right corners of the window.

14.2.2 Relative Cursor Positions

The *curcur* field of the *tty* structure corresponds to the current cursor position in the window, *i.e.*, where the next character will be displayed. This field does not contain absolute screen coordinates. Rather, the column and row coordinates in *curcur* are relative to the *topleft* coordinates of the window. For example, if *curcur* contains the coordinates (0,0), then the window cursor position is at the upper-left corner of the window.

14.2.3 Window Borders And Character Attributes

A window may be opened with or without a border. The *window.h* file contains PC-specific character codes for drawing a border around a window. Since output characters cannot appear on the window border, having a border effectively reduces the number of columns and rows available by two. The *topleft* and *botright* fields are adjusted accordingly if a border is present. Since closing the window erases the window display, the *hasborder* Boolean variable is used to determine if the window border needs to be erased too.

A user process may also choose to have the characters in the window displayed in colors (intensities for a monochrome display) different from the default white characters on a black background. The display colors/intensities are called *attributes*. For an open window, the attributes are stored in the *attr* component of the *tty* structure. These attributes are obtained from a parameter in the *open* call.

14.3 Opening A Window

Procedure *ttyopen* makes a connection between a running program and a window device. After checking the window coordinates, *ttyopen* looks for an unused window slot in the device switch table. An output server process *lwoproc* is created whose function is the same as *ttyoproc* – namely, to serve as the lower-half device driver for the window. The rest of the code draws a border around the window if required and sets up the remaining *tty* structure components. File *ttyopen.c* contains the code.

```
/* ttyopen.c - ttyopen */

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

/*------------------------------------------------------------------------
 *  ttyopen  --  open a window on a tty
 *------------------------------------------------------------------------
 */
ttyopen(devptr,bp,ap)
struct  devsw   *devptr;
char *bp;                               /* border string                */
char *ap;                               /* attribute string             */
{
        register struct tty *iptr;
        int     pid;
        int     ps;
        char    cp[7];          /* output server process name           */
        int     i;
        CURSOR  boxp[2];        /* window corners from passed params    */
        CURSOR  topl,botr;      /* topleft, bottom right of window      */
        int     attr;           /* window attributes (color, etc.)      */
        int     lwoproc();      /* tty server process                   */
        int     brdr;           /* true if the window has a border      */
```

```
if ( bp == NULL || *bp == 0 )   /* reopen the console           */
        return(tty[0].dnum);
if ((brdr=lwbord(bp,boxp))==SYSERR||(attr=lwattr(ap,BW))==SYSERR)
        return(SYSERR);
topl = boxp[0];
botr = boxp[1];
if ( topl.col >= G_COLS || topl.row >= G_ROWS
       || botr.col >= G_COLS || botr.row >= G_ROWS )
        return(SYSERR);
i = brdr ? 2 : 0;               /* offset for border            */
if ( topl.row+i > botr.row || topl.col+i > botr.col )
        return(SYSERR);         /* not enough room for window   */
if ( brdr ) {                   /* make room for the border     */
        botr.col--;
        botr.row--;
        topl.col++;
        topl.row++;
}
disable(ps);
if ( (i=wfree()) == SYSERR ) {
        restore(ps);
        return(SYSERR);
}
iptr = &tty[i];
strcpy(cp,"*LWO *");
cp[4] = '0'+i;
if ( (pid=create(lwoproc,INITSTK,TTYOPRIO,cp,1,i)) == SYSERR ) {
        iptr->oprocnum = -1;
        iptr->wstate = LWFREE;  /* mark it as free              */
        restore(ps);
        return(SYSERR);
}
iptr->oprocnum = pid;
ready(pid);
iptr->hasborder = brdr;
iptr->topleft = topl;
iptr->botright = botr;
iptr->attr = attr;
iptr->rowsiz = botr.row-topl.row+1;
iptr->colsiz = botr.col-topl.col+1;
iptr->curcur.row = 0;
iptr->curcur.col = 0;
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;
        scrollup(boxp[0],boxp[1],0,attr); /* erase the window           */
        if ( brdr )
                border(boxp[0],boxp[1]); /* draw the border             */
        restore(ps);
        return(iptr->dnum);
}

/*-------------------------------------------------------------------------
 *  border  --  draw a border around a screen window
 *-------------------------------------------------------------------------
 */
LOCAL border(tl,br)
CURSOR tl,br;
{
        CURSOR  csr;            /* used for absolute cursor positioning */
        int     pcx;

        xdisable(pcx);
        for ( csr.row=tl.row; csr.row<=br.row; csr.row++ ) {
                if (csr.row==tl.row) {
                        csr.col = tl.col;
                        putcsrpos(csr,0);
                        putchr(TLBORDER,1,0);
```

```
                        for ( csr.col++ ; csr.col<br.col ; csr.col++ ) {
                                putcsrpos(csr,0);
                                putchr(HBORDER,1,0);
                        }
                        putcsrpos(csr,0);
                        putchr(TRBORDER,1,0);
                } else if (csr.row==br.row) {
                        csr.col = tl.col;
                        putcsrpos(csr,0);
                        putchr(BLBORDER,1,0);
                        for ( csr.col++ ; csr.col<br.col ; csr.col++ ) {
                                putcsrpos(csr,0);
                                putchr(HBORDER,1,0);
                        }
                        putcsrpos(csr,0);
                        putchr(BRBORDER,1,0);
                } else {
                        csr.col = tl.col;
                        putcsrpos(csr,0);
                        putchr(VBORDER,1,0);
                        csr.col = br.col;
                        putcsrpos(csr,0);
                        putchr(VBORDER,1,0);
                }
        }
        xrestore(pcx);
}

/*---------------------------------------------------------------------------
 *  wfree  --  get a free window slot
 *---------------------------------------------------------------------------
 */
LOCAL
wfree()
{
        int     i;
        struct tty      *iptr;

        for ( i=1 ; i<Ntty ; i++ ) {
                iptr = &tty[i];
                if ( iptr->wstate == LWFREE ) {
                        iptr->wstate = LWALLOC;
                        iptr->seq++;
                        return(i);
```

```
                }
        }
        return(SYSERR);
}
```

Note that much of the code for *tty* structure initialization in *ttyopen* is the same as in *ttyinit*. The reason these fields are initialized in *ttyopen* rather than at system initialization is that when a window is closed and then re-opened, the window characteristics should be returned to their default values. Since the *CONSOLE* is never closed, its characteristics need to be initialized only once.

The *wfree* routine in *ttyopen* searches the *tty* structure for a free device. The *wstat* field contains *LWFREE* if the window is unused. When a free slot is found, *wfree* fills the *wstat* field with *LWALLOC*, marking the window as unavailable for further *open* operations. (Recall that for the *CONSOLE* device, the *wstat* field is used differently – to hold the id of the keyboard server process.)

As the name *ttyopen* implies, this procedure is associated with the *CONSOLE* driver. This may seem confusing, but it will become clear when you understand the following:

> *Because windows are dynamically allocated, the open operation is associated with the* CONSOLE *driver, not with the individual window devices.*

To connect a process to a window, the user calls *open*, passing *CONSOLE* as the first argument, a character string describing the coordinates of the upper-left and bottom-right window corners as the second argument, and a character string describing the window attributes as the third. *Open* uses the device switch table to pass the call to *ttyopen*, which returns the device number of the window pseudo-device allocated by the *open* operation. If *open* is called with an empty string or *NULL* as its second argument, it simply returns the descriptor of the *CONSOLE* device without opening a new window.

Translating border and attribute strings into numeric window coordinates and attribute bytes is carried out with the *lwbord* and *lwattr* routines, respectively.

```
/* lwbord.c - lwbord */

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

/*------------------------------------------------------------------------
 *  lwbord  --  parse window border attribute string
 *------------------------------------------------------------------------
 */
```

```
int lwbord(bp,w)
char *bp;
CURSOR w[2];                            /* pointer to cursor array      */
{
        int     i,j;
        int     coord;                  /* row/col coordinate generated */
        int     brdr;                   /* true if window has a border  */

        if ( bp == NULL )
                return(SYSERR);
        if ( brdr = (*bp=='#') )
                bp++;
        for(i=0 ; i<2 ; i++) {          /* loop for two sets of ints    */
                for(j=0 ; j<2 ; j++) {  /* loop for two ints per set    */
                        if ( isdigit(*bp) == 0 )
                                return(SYSERR);
                        coord = 0;
                        while ( isdigit(*bp) ) {
                                coord *= 10;
                                coord += (*bp++) - '0';
                        }
                        if ( coord < 0 || coord >= 256 )
                                return(SYSERR);
                        switch(j) {
                                case 0:
                                        w[i].col  = coord;
                                        break;
                                case 1:
                                        w[i].row  = coord;
                                        break;
                        }
                        if ( *bp == ',' && j == 0 )
                                bp++;
                }
                if ( *bp == ':' && i == 0 )
                        bp++;
        }
        if ( *bp != 0 )
                return(SYSERR);
        return(brdr);
}
```

```
/* lwattr.c - lwattr */

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

#define ATTR_NOBLINK            0x3f
#define ATTR_BLINK              0x80
#define ATTR_NOINTENSE          0x87
#define ATTR_INTENSE            0x08
#define ATTR_DEFAULT            0x07

#define newcol(c,n,j)   ( ((c) & ~( 7<<(j) )) | ((n) << (j)) )

static char *clrs[] = { "blk","blu","grn","cyn","red","mag","yel","wht" };

/*------------------------------------------------------------------------
 *  lwattr  --  parse window color attribute string
 *------------------------------------------------------------------------
 */
int  lwattr(ap,attr)
char    *ap;
int     attr;                           /* default value                */
{
        char    tmp[4];                 /* used to compare with clrs    */
        int     i,j;
        int     cnum;                   /* value of selected attr       */
        int     shift = 0;              /* shift count                  */

        if ( ap == NULL ||  *ap == 0 )
                return(attr);
        while (  isalnum(*ap) == 0 && *ap != '/' ) {
                if ( *ap == 0 )
                        return(attr);
                switch ( *ap++ ) {
                        case '*':
                                attr &= ATTR_NOBLINK;
                                break;
                        case '?':
                                attr |= ATTR_BLINK;
                                break;
                        case '-':
                                attr &= ATTR_NOINTENSE;
```

```
                        break;
                case '+':
                        attr |= ATTR_INTENSE;
                        break;
                default:
                        return(SYSERR);
        }
    }
    for( i=0 ; i<3 ; i++ ) {
        if ( *ap == 0 )
                return(attr);
        if ( i == 1 ) {
                if ( *ap != '/' )
                        return(SYSERR);
                ap++;
                shift = 4;
                continue;
        }
        if ( isdigit(*ap) )
                cnum = (*ap++) - '0';
        else if ( isalpha(*ap) ) {
                for ( j=0 ; j<3 ; j++ ) {
                        if ( isalpha(*ap) == 0 )
                                return(SYSERR);
                        tmp[j] = tolower(*ap);
                        ap++;
                }
                tmp[j] = 0;
                for ( cnum=0 ; cnum<8 ; cnum++ )
                        if ( strcmp(clrs[cnum],tmp) == 0 )
                                break;
        } else
                continue;
        if ( cnum >= 8 )
                return(SYSERR);
        attr = newcol(attr,cnum,shift);
    }
    if ( *ap == 0 )
            return(attr);
    return(SYSERR);
}
```

The border string passed as the first parameter to *lwbord* has the form ''#c1,r1:c2,r2'' where ''c1,r1'' is a string representing the decimal coordinates (in column,row order) of the upper-left corner of the window, and ''c2,r2'' is a string representing the coordinates of the bottom-right corner. If the '#' character is omitted at the beginning of the border string, the window will be created without a border. The second parameter to *lwbord* is a pointer to an array of two cursor variables; upon return, this array contains the coordinates of the upper-left and bottom-right coordinates of the window to be created. The value returned by *lwbord* is one if a border should be present, and zero otherwise.

The attribute string passed as the first parameter to *lwattr* has the form ''fff/bbb.'' The 'fff' and 'bbb' strings are three-character color codes representing the foreground and background colors to be used with a color display. The color codes and corresponding colors are given in Figure 14.1.

Code	Color
blk	black
blu	blue
grn	green
cyn	cyan
red	red
mag	magenta
yel	yellow
wht	white

The 'fff' or 'bbb' fields may be replaced by a single decimal digit in the range 0 to 7, which specifies the numeric code for the foreground or background color, respectively. For monochrome displays, the numeric code represents levels of gray with 0 for black and 7 for white. If either or both of the 'fff' or 'bbb' fields are missing, the default values are taken. The window color codes may be preceded by optional blink or intensity specifiers in the attribute string. A '?' blink specifier indicates that the foreground blinks, while a '*' specifies that it does not. A '+' specifies that the foreground is intensified, while a '-' specifies that it is not.

Attributes that are not explicitly given in the attribute string are taken from the second parameter to *lwattr*. *Lwattr* returns the attribute value.

An example of a call to *open* is

```
open( CONSOLE, "#10,4:30,20", "red/wht" );
```

This will create a bordered window with upper-left coordinates (10,4) and lower-right coordinates (30,20), and with red characters on a white background.

14.4 Upper-Level Window Driver Routines

The high-level window input/output operations are similar to those for the *CONSOLE* tty device. The differences arise from the dynamic nature of window creation and deletion. A process may be blocked in the midst of performing input/output operations to a window when another process closes and deallocates it. Care must be taken to ensure that a blocked operation completes only if the state of the window has not changed.

14.4.1 Window Output Operations

Circular buffer management for the window *putc* driver is identical to that for the tty driver. File *lwputc.c* contains code for the driver routine *lwputc*:

```
/* lwputc.c - lwputc */

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

/*------------------------------------------------------------------------
 *  lwputc  --  put a character into a logical window
 *------------------------------------------------------------------------
 */

lwputc(devptr, ch)
struct  devsw   *devptr;
char    ch;
{
        struct  tty     *iptr;
        int     ps;
        int     seq;

        iptr = &tty[devptr->dvminor];
        disable(ps);
        if ( iptr->wstate != LWALLOC ) {/* is window open?             */
                restore(ps);
                return(SYSERR);
        }
        seq = iptr->seq;
        wait(iptr->osem);               /* wait for space in queue      */
        if ( iptr->wstate != LWALLOC    /* is window still open?        */
             || iptr->seq != seq ) {
                restore(ps);
                return(SYSERR);
        }
        iptr->obuff[iptr->ohead++] = ch;
        ++iptr->ocnt;
        if (iptr->ohead >= OBUFLEN)
                iptr->ohead = 0;
        sendn(iptr->oprocnum,TMSGOK);   /* wake up the tty process      */
        restore(ps);
        return(OK);
}
```

The initial code in *lwputc* verifies that the window is open. What happens next is more interesting. *Lwputc* waits on the output buffer semaphore *osem*. The call returns im-

mediately if the buffer is not full, but it delays otherwise. What may seem odd is that *lwputc* records the value of *seq* before the call to *wait* and then verifies that it remained the same after the call. This code has been introduced because windows may be closed (and even reopened) while processes remain enqueued waiting to write to them. When this happens, the window sequence number *seq* changes, and the waiting processes are resumed. The idea is to have the waiting processes verify that the *wait* did not terminate because the window was closed. If it did, and the sequence number changed, *lwputc* reports an error to its caller.

Lwwrite is almost identical to *ttywrite*. The only differences are that *lwwrite* checks to see if the window is open before writing to the buffer and also checks the status of the repeated calls to *lwputc*.

```
/* lwwrite.c - lwwrite */

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

/*------------------------------------------------------------------------
 *  lwwrite  --  write one or more characters to a console window
 *------------------------------------------------------------------------
 */
lwwrite(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(OK);
        disable(ps);
        ttyp = &tty[devptr->dvminor];
        if ( ttyp->wstate != LWALLOC ) {/* is window open?              */
                restore(ps);
                return(SYSERR);
        }
        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--)
                        if ( lwputc(devptr, *buff++) == SYSERR ) {
                                restore(ps);
```

```
                                return(SYSERR);
                        }
        }
        restore(ps);
        return(OK);
}
```

14.4.2 Window Input And Control Operations

The upper-level window input and control driver routines *lwgetc*, *lwread*, and *lwcntl* are similar to their tty counterparts. The code is in files *lwgetc.c*, *lwread.c*, and *lwcntl.c*:

```
/* lwgetc.c - lwgetc */

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

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

        iptr = &tty[devptr->dvminor];
        disable(ps);
        if ( iptr->wstate != LWALLOC ) {/* is window open?              */
                restore(ps);
                return(SYSERR);
        }
        seq = iptr->seq;
        wait(iptr->isem);               /* wait for a character in buff */
        if ( iptr->wstate != LWALLOC    /* is window still open?        */
             || iptr->seq != seq ) {
                restore(ps);
                return(SYSERR);
        }
        ch = iptr->ibuff[iptr->itail++];
        --iptr->icnt;
        if (iptr->itail >= IBUFLEN)
                iptr->itail = 0;
        restore(ps);
        return(ch);
}
```

```
/* lwread.c - lwread */

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

/*------------------------------------------------------------------------
 *  lwread  --  read one or more characters from a logical window
 *------------------------------------------------------------------------
 */
lwread(devptr, buff, count)
struct devsw *devptr;
char *buff;
int count;
{
        register struct tty *ttyp;
        int     avail, nread;
        int     ps;

        if ( count<0 )
                return(SYSERR);
        disable(ps);
        ttyp = &tty[devptr->dvminor];
        if ( ttyp->wstate != LWALLOC ) {/* is window open?              */
                restore(ps);
                return(SYSERR);
        }
        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--)
                        if ( (*buff++ = lwgetc(devptr)) == SYSERR ) {
```

```
                                restore(ps);
                                return(nread-count);
                        }
        }
        restore(ps);
        return(nread);
}


/* lwcntl.c - lwcntl */

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

/*------------------------------------------------------------------------
 *  lwcntl  --  do misc. tty control functions
 *------------------------------------------------------------------------
 */

SYSCALL lwcntl(devptr, func)
struct devsw *devptr;
int func;
{
        register struct tty     *ttyp;
        int     ps;
        int     c;

        disable(ps);
        ttyp = &tty[devptr->dvminor];
        if ( ttyp->wstate != LWALLOC ) {/* can't access unopened window */
                restore(ps);
                return(SYSERR);
        }
        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);
}
```

14.4.3 Closing A Window

Closing a window is straightforward. The *lwclose* routine kills the output server process, deletes the buffer semaphores (releasing any waiting processes), and erases the window.

```
/* lwclose.c - lwclose */

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

/*------------------------------------------------------------------------
 *  lwclose  --  routine to close a window device
 *------------------------------------------------------------------------
 */

lwclose(devptr)
struct  devsw   *devptr;
{
        struct  tty     *iptr;
        int     ps;
        int     sct;

        disable(ps);
        iptr = &tty[devptr->dvminor];
        if ( iptr->wstate != LWALLOC ) {/* can't close unopened window  */
                restore(ps);
                return(SYSERR);
        }
        iptr->wstate = LWLIMBO;
        iptr->seq++;
        kill(iptr->oprocnum);           /* kill the output process      */
        if ( winofcur == devptr->dvminor )
                winofcur = 0;   /* current cursor can't be here anymore */
        sdelete(iptr->isem);
        sdelete(iptr->osem);
        if (iptr->hasborder) {
                --iptr->topleft.col;
                --iptr->topleft.row;
                ++iptr->botright.col;
                ++iptr->botright.row;
        }
        scrollup(iptr->topleft,iptr->botright,0,BW);    /* erase window */
        iptr->wstate = LWFREE;
        restore(ps);
        return(OK);
}
```

Lwclose marks the state *wstate* of the window as *LWLIMBO* while it carries out the remainder of the code. When the window is in this state, processes attempting to perform input/output operations will be denied access (even those that were previously blocked, as in *lwputc*). But equally important, requests to open a new window will find that the device is not free and will consequently not attempt to open it. Only after the entire deallocation process has completed is the state changed to *LWFREE*, making it possible for the window pseudo-device slot to be allocated to another screen window. This two-step deallocation technique is necessary because *lwclose* calls routine *sdelete* that may result in a reschedule.

The *seq* component of the window *tty* structure is incremented in *lwclose* as it was in *ttyopen*. This safeguards against the scenario where a window is closed and then immediately re-opened, while a process is blocked on the window's input or output buffer semaphore. In such a situation, the blocked process will find that the sequence number has changed, so the input or output operation will fail.

14.5 The Lower-Half Window Output Server Process

Each open window has a server process which is essentially the same as the tty output server process discussed in Chapter 12. The window output server is created when the window is opened and killed when the window is closed. When a process calls an upper-half window output operation, the upper-half routine queues the output characters and sends a message to the lower-half server process, which dequeues the output characters and displays them in the proper window on the video display. The lower-half process must manage the translation of logical cursor position in the window into the corresponding physical position on the screen. The logical cursor position, contained in the *curcur* field of the *tty* structure, is relative to the window size and location on the physical screen, determined by fields *topleft* and *botright*. File *lwoproc* contains the code for the lower-half server process.

```
/* lwoproc.c - lwoproc */

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

/*------------------------------------------------------------------------
 *  lwoproc  --  lower-half tty device driver process for window output
 *------------------------------------------------------------------------
 */

PROCESS lwoproc(i)
int i;                                  /* tty minor device number      */
{
        register struct tty   *iptr;
        int     ps;
        int     ct;
        char    ch;
        Bool    enl, onl;
        CURSOR  *c;
        int     wput();
        int     rcvchr();

        iptr = &tty[i];                 /* pointer to tty structure     */
        onl = enl = FALSE;
        c = &(iptr->curcur);
        disable(ps);
        for ( ; ; ) {                   /* endless loop                 */
                if (enl) {
                        enl = FALSE;
                        wput(iptr,c,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;
```

```
                        }
                        wput(iptr,c,ch);
                        continue;
                }
                if (iptr->oheld) {
                        rcvchr();
                        continue;
                }
                if (onl) {
                        onl = FALSE;
                        wput(iptr,c,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;
                        }
                        wput(iptr,c,ch);
                        continue;
                }
                rcvchr();
        }
}

/*------------------------------------------------------------------------
 *  wput  --  put a single character to the window; honor NEWLINE, etc.
 *------------------------------------------------------------------------
 */
LOCAL wput(iptr, c, ch)
register struct tty   *iptr;
CURSOR  *c;
char    ch;
{
```

```
        int     wrap();
        int     pcx;

        if ( ch < BLANK ) {
                switch (ch) {
                case RETURN:
                        c->col = 0;
                        break;
                case NEWLINE:
                        if ( c->row == iptr->rowsiz-1 )
                                scrollup(iptr->topleft,iptr->botright,
                                    ((iptr->rowsiz>1)?1:0),iptr->attr);
                        else
                                c->row++;
                        break;
                case BELL:
                        wtty(ch);
                        return;
                case BACKSP:
                        if ( c->col > 0 )
                                c->col--;
                        break;
                case TAB:
                        c->col += TABSTOP;
                        c->col -= (c->col % TABSTOP);
                        wrap(iptr,c);
                        break;
                default:
                        return;         /* do nothing */
                }
                wputcsr(iptr,*c);
        } else {
                xdisable(pcx);
                wputcsr(iptr,*c);
                putchr(ch, 1, 0);
                xrestore(pcx);
                c->col++;
                wrap(iptr,c);
        }
}

/*---------------------------------------------------------------------
 *  wrap  --  wrap around to a new line in the window; scroll if necessary
 *---------------------------------------------------------------------
```

```
 */
LOCAL wrap(iptr,c)
register struct tty   *iptr;
CURSOR  *c;
{
        if (c->col >= iptr->colsiz ) {
                if (c->row >= iptr->rowsiz-1)
                   scrollup(iptr->topleft,iptr->botright,
                        ((iptr->rowsiz>1)?1:0),iptr->attr);
                else
                   c->row++;
                c->col = 0;
        }
}

/*-------------------------------------------------------------------------
 *  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);
        }
}
```

Writing a character to the screen involves moving the cursor into the window and displaying the character at the cursor position. *Lwoproc* is more complicated than *ttyoproc* because it must contend with the details of cursor position and window management that the tty device leaves to the *wtty* BIOS routine. The *wput* routine in *lwoproc.c* carries out both cursor positioning and character writing operations. Observe that *wput* must deal with special characters, such as *RETURN*, *NEWLINE*, or *TAB*, and when to wrap around to a new line. *Wput* brackets its code to display a character with calls to *xdisable* and *xrestore* to avoid reschedules between positioning the cursor and writing a character to the screen; such reschedules can result displaying the character at the wrong location on the screen.

14.6 Low-Level PC Screen Operations

Like *wtty*, low-level screen operations including positioning the cursor, writing a character to the screen, and scrolling a screen region are carried out by calls to the BIOS video interrupt *VID*, described in Chapter 2. File *pcscreen.c* contains the code:

```
/* pcscreen.c - putcsrpos, scrollup, putchr */

#include <dos.h>
#include <conf.h>
#include <kernel.h>
#include <window.h>
#include <vidio.h>

#define INT10H(r)       vidint(&(r))

#define AH(r)           ((r).h.ah)
#define AL(r)           ((r).h.al)
#define BH(r)           ((r).h.bh)
#define BL(r)           ((r).h.bl)
#define CH(r)           ((r).h.ch)
#define CL(r)           ((r).h.cl)
#define DH(r)           ((r).h.dh)
#define DL(r)           ((r).h.dl)
#define AX(r)           ((r).x.ax)
#define BX(r)           ((r).x.bx)
#define CX(r)           ((r).x.cx)
#define DX(r)           ((r).x.dx)
#define CF(r)           ((r).x.cflag)

#define UNSIGNED(x)     ( *( (unsigned int *) &(x) ) )  /* alias!       */

#define SCSRPOS 2
#define SCRLUP  6
#define WCHR    10

/*------------------------------------------------------------------------
 *  putcsrpos  --  put the cursor at a given position
 *------------------------------------------------------------------------
 */
putcsrpos(csr,page)
CURSOR csr;
unsigned page;
{
```

```
        union REGS r;

        AH(r)=SCSRPOS;
        DX(r)=UNSIGNED(csr);
        BH(r)=page;
        INT10H(r);
}

/*---------------------------------------------------------------------------
 *  scrollup  --  scroll a window up
 *---------------------------------------------------------------------------
 */
scrollup(tl,br,lines,attr)
CURSOR tl,br;
unsigned lines;
unsigned char attr;
{
        union REGS r;

        AH(r)=SCRLUP;
        AL(r)=lines;
        CX(r)=UNSIGNED(tl);
        DX(r)=UNSIGNED(br);
        BH(r)=attr;
        INT10H(r);
}

/*---------------------------------------------------------------------------
 *  putchr  --  put character at current cursor position
 *---------------------------------------------------------------------------
 */
putchr(ch,count,page)
char ch;
unsigned count,page;
{
        union REGS r;

        AH(r)=WCHR;
        AL(r)=ch;
        CX(r)=count;
        BH(r)=page;
        INT10H(r);
}
```

Note the use of macros in this code. Macros such as *AH* serve two purposes: they simplify coding, and they are easier to read, as one can see by comparing *AH*(*r*) to *r.h.ah*. The *UNSIGNED* macro looks complicated, but it is easily understood once you recognize that its purpose is to translate a two-byte *CURSOR* structure into an unsigned (16-bit) integer. This subterfuge is necessary only to trick the compiler into accepting the code without complaint.

A related screen management routine is *wputcsr*. The parameters to *wputcsr* are a pointer *iptr* to the *tty* structure of a window and a cursor position *csr*. The purpose of *wputcsr* is to translate a relative cursor position in a logical window into a physical cursor position on the screen. The *iptr* parameter provides access to the window corner parameters *topleft* and *botright* in the *tty* structure. The code for *wputcsr* is extremely simple.

```
/* wputcsr.c - wputcsr */

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

/*------------------------------------------------------------------------
 *  wputcsr  --  cursor position routine
 *------------------------------------------------------------------------
 */

wputcsr(iptr,csr)
struct  tty     *iptr;
CURSOR csr;
{
        csr.row += iptr->topleft.row;
        csr.col += iptr->topleft.col;
        putcsrpos(csr,0);
}
```

14.7 Window Keyboard Input

It is obvious that there is only one keyboard, even though there may be many open windows. If a process is waiting to *getc* a character from a window device, how can the system know which window device a keyboard character should go to? Another issue concerns output flow control. If processes are displaying characters to several windows simultaneously, should a suspend request (usually *Ctrl-S*) suspend *all* window output?

Careful design will permit the keyboard to be assigned only to one window at a time. A global variable *winofcur* (standing for ‘‘window of cursor’’) contains the current tty device to which keyboard input will be directed. The corresponding window will be called the *input window*. *Winofcur* is initialized to zero, representing the *CONSOLE* device. The reader should consult the *ttyiproc* code described in Chapter 12.

Keyboard function keys are used to switch the keyboard input between windows and the *CONSOLE*. We identify windows with their minor device numbers; the *CONSOLE* is treated as window zero. The tty keyboard server process traps special function keys *Fn* to switch to window *n*; as a special case, function key *F10* switches to the *CONSOLE*. If window *n* is not open, the switch is not made.

Switching to an input window amounts to assigning the window number to the *winofcur* variable. Displaying the cursor in the input window is more complicated, since the cursor is continually moving from window to window during screen display. To ensure that the cursor will appear in the input window when screen output is idle, each window output server process *lwoproc* positions the cursor in the input window after it has emptied its buffer, just before calling *receive*. To ensure that the cursor is positioned in the input window as soon as a switch is made, *ttyiproc* sends a message to the window output server process, guaranteeing that it will awaken to position the cursor if it is idle. Observe that this design results in output flow control on a window-by-window basis. Flow control characters only affect the current input window.

Each window keeps track of its current cursor position in *tty* variable *curcur*, and input characters are echoed in the window in the same way output characters are displayed. The *CONSOLE* is treated differently, because no *curcur* position is maintained for the *CONSOLE*. (The exercises explore why this is the case.) For this reason, *CONSOLE* output is displayed and input is echoed wherever the cursor happens to be on the screen.

When the current input window is closed, it clearly cannot continue to be the input window. *Lwclose* therefore makes the *CONSOLE* the input window in this case.

14.8 Window Driver Initialization

Driver initialization, which is done once when PC-Xinu is started, is straightforward for window devices. Since much of the initialization of the *tty* structure for windows is done in *lwopen*, the *lwinit* routine is concerned only with those *tty* structure fields which remain constant, or which relate to window allocation.

```
/* lwinit.c - lwinit */

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

/*------------------------------------------------------------------------
 *  lwinit  --  initialize console window
 *------------------------------------------------------------------------
 */
lwinit(devptr)
        struct  devsw   *devptr;
{
        register struct tty *iptr;

        iptr = &tty[devptr->dvminor];
        iptr->dnum = devptr->dvnum;
        iptr->oprocnum = -1;                    /* no output process    */
        iptr->wstate = LWFREE;                  /* window is free       */
        iptr->seq = 0;                          /* init sequence no.    */
        devptr->dvioblk = (char *) iptr;        /* fill tty control blk */
}
```

14.9 Summary

Windows provide a way to display the output from logically different activities in physically separate screen display areas, making it easier to observe concurrency. This chapter has shown how windows can be implemented as logical tty devices. Upper-level window driver routines are similar to their respective tty driver routines and share the same *tty* structure.

Window driver design is complicated by hardware-specific screen management requirements such as cursor positioning, scrolling, and character display. These are isolated in the lower-level output driver where their details are hidden from higher levels.

Since windows may be opened and closed dynamically, the design has taken into account the disposition of processes blocked on window input/output when the window status changes.

FOR FURTHER STUDY

Myers [1986] gives details for implementing overlapping windows, including changing a window's position and size. A comparison of overlapping and non-overlapping windows is given in Bly *et. al.* [1986]. Human factors relating to windows are described in Holcomb *et. al.* [1986].

EXERCISES

14.1 Both *lwputc* and *lwgetc* check the sequence number *seq* to determine if the window status has changed, but neither *lwwrite* nor *lwread* examine *seq*. Why?

14.2 Does the sequence number technique absolutely guarantee that a process blocked on the *osem* semaphore in *lwputc* will find a different value for *seq* if the window status has changed when the process resumes? Explain.

14.3 Correct the design flaw suggested by the previous exercise.

14.4 *Lwoproc* handles *TAB* output characters (expanding them to an appropriate number of spaces), but *ttyoproc* does not. Modify *ttyoproc* to handle *TAB*s.

14.5 Should *TAB* expansion be handled by the *ttyiproc* input server instead of the output server? Discuss the advantages and liabilities of each.

14.6 In ''cooked'' mode, the keyboard delete character does not correctly delete expanded *TAB*s from the window. Modify the drivers so that expanded *TAB*s are deleted properly.

14.7 Design a *lwcntl* function that positions the cursor in the window at a location determined by a parameter of type *CURSOR*. The *CURSOR* parameter should give relative, not absolute screen coordinates.

14.8 Design a *lwcntl* function that erases a window (except for the border if there is one) and positions the cursor at the upper-left corner.

14.9 What other *lwcntl* functions are appropriate for windows? Design and implement them.

14.10 Design a mechanism that allows for overlapping windows. When window *A* overlaps window *B*, output to the region of window *B* that lies under window *A* will be saved, but not displayed on the screen. When window *A* is closed, the characters in window *B*, which were ''under'' window *A*, will be displayed. Your design should include the ability to bring a window ''to the top.'' In the example above, bringing window *B* to the top will mean that the full window *B* will be displayed, and characters in window *A* now ''under'' window *B* will disappear from the screen.

14.11 Making BIOS calls to position the cursor and to display characters on the screen is extremely inefficient. Redesign the screen output routines to write characters directly to PC screen display memory. What are the disadvantages of this approach?

14.12 Arrange the system configuration so that the *CONSOLE* is just another window. What can happen in this case if a processes *close*s the *CONSOLE*?

14.13 Why doesn't the *CONSOLE* keep track of its current cursor position? (Consider what happens with *kprintf*.)

14.14 Read about the *X* window system, designed at MIT. How does it differ from the windows provided by PC-Xinu?

14.15 SUN Microsystems has developed a window system for bit-mapped displays called NeWS that interprets postscript commands. Read about NeWS and identify the features that distinguish it from other window systems.