

Chapter 10. Device Drivers

Abstract: Chapter 10 covers the design and implementation of device drivers. These include drivers for the console display, keyboard, parallel printer, serial ports, floppy drive, IDE hard disk and ATAPI driver for CDROM. With the exception of the console display, which does not use interrupts, all device drivers are interrupt-driven. It describes the driver design principle and presents a general framework that is applicable to all interrupt-driven driver design. It describes the interaction and synchronization between process and interrupt handler and explains in detail why interrupt handlers can not sleep or become blocked. For each device, it explains the device operations before showing the actual driver implementation. In addition, it demonstrates the device drivers by sample systems, which allows the reader to test and observe the operations of I/O device drivers.

10.1. Device Drivers

A device driver is a piece code which controls a device for I/O operations. It is an interface between a physical device and processes doing I/O on the device. So far, MTX uses BIOS for I/O. In this chapter, we shall develop our own device drivers to replace BIOS for I/O operations. These include drivers for the console display, keyboard, parallel printer, serial ports, floppy drives, IDE hard disk and CD/DVD ROM. In a multitasking system, I/O operations should be done by interrupts, not by polling. Therefore, all the device drivers, except the console display, are interrupt-driven. When developing the device drivers we shall focus on the principles of driver design, rather than trying to implement complete drivers. For example, in the console display driver, we only show how to display ordinary chars, but not all the special chars. In the keyboard driver, we only show how to handle ordinary keys and some of the function and control keys, but not all the escape key sequences. In the disk and CDROM drivers, we only show how to recognize exceptions but do not attempt to handle all the error conditions, etc. Despite these simplifications, all the device drivers presented in this chapter are functional, but they are by no means complete. Interested readers may try to improve on the drivers to make them complete. As usual, after presenting the design and implementation of each device driver, we shall demonstrate the driver operations by a sample system. Since the main objective is to demonstrate the device drivers, most of the sample systems are based on MTX5.1. For ease of reference, we shall label the sample systems MTX10.xy, where the suffix letters indicate the included drivers. Among the I/O devices, the console display is the simplest because it is a memory mapped device, which does not use interrupts. So we begin with the console display driver.

10.2. Console Display Driver

The console display of IBM compatible PCs has many modes, ranging from the original VGA to what's broadly referred to as SVGA. MTX does not support graphic display. It only uses the basic VGA mode to display text in 16 colors with 640x400 resolutions. From a programming point of view, the PC's display works as follows. It is a memory mapped device. The display screen is organized as 25 rows by 80 columns, for a

total of $25 \times 80 = 2000$ characters. The display memory, known as the video RAM, is a memory area containing characters to be displayed. Each character in the video RAM is represented by 2 bytes=[attribute, character], where the attribute byte determines how to display the character, e.g. color, intensity, reverse video, blinking, etc. The second byte is the ASCII code of the character to be displayed. The video RAM begins at the segment address 0xB800. The size of the video RAM varies from 16 to 32KB, which is much larger than the 4000 bytes needed to display a screen. The interface between the video RAM and the display screen is a Video Display Controller (VDC), which has a cursor position register and a display start address register. The VDC simply displays the 4000 2-byte words, beginning from the start address, in the video RAM to the screen as 2000 characters in a 25x80 layout. The cursor shape and size can be programmed through the cursor height register. The cursor location is determined by the cursor position register. For proper visual effect, the cursor is positioned at where the next character is to be displayed. The relationship between the video RAM contents and the display screen is shown in Figure 10.1.

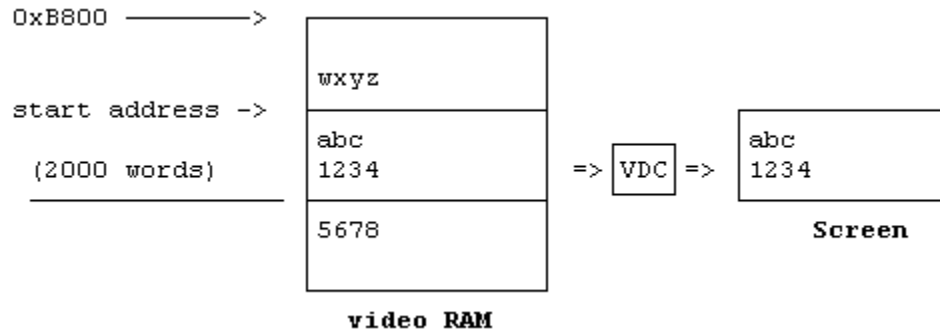


Figure 10.1 Video RAM and Display

Assume that each box of the video RAM in Figure 10.1 contains 2000 words. The VDC simply displays the words in the box pointed by the start address. If we move the start address up one line (toward low address), the displayed screen would show wxyz at the top. The visual effect is that the entire screen is scrolled downward one line. If we move the start address down one line, the screen would be scrolled up one line. When moving the start address downward (toward higher address) we must monitor its position in the video RAM area. If the remaining RAM area is less than a screen size, we must copy the next screen to the beginning of the RAM area and reset the start address to the video RAM beginning address. This is because the video RAM address does not wrap around by itself. The VDC registers can be accessed through the display adaptor's index register at the I/O port address 0x3D4. To change a VDC register, first select the intended register. Then write a (2-byte) value to the data register at 0x3D5 to change the selected register. When programming the VDC the most important VDC register pairs are start_address = 0x0C-0x0D and cursor_position = 0x0E-0x0F. The display driver algorithm is as follows.

(1). call vid_init() to initialize the cursor size, cursor position and, VID_ORG of the VDC hardware. Initialize the driver variables row and column to 0. Clear the screen and position the cursor at the top left corner.

- (2). A screen = 25 rows by 80 columns = 2000 words = [attribute byte, char byte] in the video RAM, beginning at start_address=0xB8000.
- (3). To display a screen: write 4000 bytes (2000 words) to the video RAM at the current VID_ORG location. The VDC will display the 2000 chars from start_address = VID_ORG in the video RAM to the screen.
- (4). The driver must keep track of the Cursor position (row, col). After displaying a char, the driver must advance the Cursor by one word, which may change (row, col). When col >= 80, reset col to 0 and increment row by 1. When row >= 25, scroll up one row. Handle special chars such as \n and \b to produce the right visual effect.
- (5). Scroll up or down:
- (5).1. To scroll up one row: increment VID_ORG by one row. Write a row of blanks to the last row on screen. If the last row exceeds video RAM size, copy current screen to the video RAM beginning and reset VID_ORG to 0.
- (5).2. To scroll down one row: decrement VID_ORG by one row.
- The following lists the display driver code.

```

/***** Display Driver vid.c file of MTX kernel *****/
#define VDC_INDEX      0x3D4
#define VDC_DATA        0x3D5
#define CUR_SIZE        10 // cursor size register
#define VID_ORG         12 // start address register
#define CURSOR          14 // cursor position register
#define LINE_WIDTH      80 // # characters on a line
#define SCR_LINES       25 // # lines on the screen
#define SCR_BYTES       4000 // bytes of one screen=25*80
#define CURSOR_SHAPE    15 // block cursor for EGA/VGA
// attribute byte: 0x0HRGB, H=highLight; RGB determine color
u16 base = 0xB800; // VRAM base address
u16 vid_mask = 0x3FFF; // mask=Video RAM size - 1
u16 offset; // offset from VRAM segment base
int color; // attribute byte
int org; // current display origin, r.e. VRAM base
int row, column; // logical row, col position

int vid_init() // initializes org=0 (row,column)=(0,0)
{ int i, w;
  org = row = column = 0; // initialize globals
  color = 0x0A; // high YELLOW;
  set_VDC(CUR_SIZE, CURSOR_SHAPE); // set cursor size
  set_VDC(VID_ORG, 0); // display origin to 0
  set_VDC(CURSOR, 0); // set cursor position to 0
  w = 0x0700; // white, blank char
  for (i=0; i<25*80; i++) // clear screen
    put_word(w, base, 0+2*i); // write 25*80 blanks to VRAM
}

int scroll() //scroll UP one line
{ u16 i, w, bytes;
  // test offset = org + ONE screen + ONE more line
  offset = org + SCR_BYTES + 2*LINE_WIDTH;
  if (offset <= vid_mask) // offset still within VRAM area
    org += 2*LINE_WIDTH; // just advance org by ONE line
  else{ // offset exceeds VRAM area ==> reset to VRAM beginning by
    // copy current rows 1-24 to BASE, then reset org to 0

```

```

        for (i=0; i<24*80; i++){
            w = get_word(base, org+160+2*i);
            put_word(w, base, 0+2*i);
        }
        org = 0;
    }
    // org has been set up properly
    offset = org + 2*24*80;    // offset = beginning of row 24
    // copy a line of BLANKs to row 24
    w = 0x0C00;    // HRGB=1100 ==> HighLight RED, Null char
    for (i=0; i<80; i++){
        put_word(w, base, offset + 2*i);
    }
    set_VDC(VID_ORG, org >> 1);    // set VID_ORG to org
}

int move_cursor()    // move cursor to current position
{
    int pos = 2*(row*80 + column);
    offset = (org + pos) & vid_mask;
    set_VDC(CURSOR, offset >> 1);
}

// display a char, handle special chars '\n', '\r', '\b'
int putc(char c)
{
    u16 w, pos;
    if (c=='\n'){
        row++;
        if (row>=25){
            row = 24;
            scroll();
        }
        move_cursor();
        return;
    }
    if (c=='\r'){
        column=0;
        move_cursor();
        return;
    }
    if (c=='\b'){
        if (column > 0){
            column--;
            move_cursor();
            put_word(0x0700, base, offset);
        }
        return;
    }
    // c is an ordinary char
    pos = 2*(row*80 + column);
    offset = (org + pos) & vid_mask;
    w = (color << 8) + c;
    put_word(w, base, offset);
    column++;
    if (column >= 80){
        column = 0;
        row++;
        if (row>=25){
            row = 24;
            scroll();
        }
    }
}

```

```

    }
}
move_cursor();
}
int set_VDC(u16 reg, u16 val) //set VDC register reg to val
{
    lock();
    out_byte(VDC_INDEX, reg);          // set index register
    out_byte(VDC_DATA, (val>>8)&0xFF); // output high byte
    out_byte(VDC_INDEX, reg + 1);      // next index register
    out_byte(VDC_DATA, val&0xFF);      // output low byte
    unlock();
}

```

10.2.1. MTX10.tv: Demonstration of Timer and Display Driver

MTX10.vt includes both the timer and display drivers. The following describes the changes in MTX10.vt for testing the console display driver.

- (1). When MTX10.vt starts, the video driver must be initialized first. It is observed that, in order for the video driver to work in color mode, the PC must start in mono display mode. This is done by a BIOS 0x10 call in assembly before calling main() in C. Upon entry to main(), the first action is to call vid_init(), which initializes the display driver.
- (2). Change putc() in Umode to a syscall, which calls putc() in the display driver.
- (3). Add a Umode command, color, which issues a syscall (9, color) to change the display color by changing the color bits in the attribute byte.
- (4). Modify the timer interrupt handler to display a wall clock on the lower right corner of the display screen. Figure 10.2 shows the display screen of running MTX10.tv.

```

QEMU
MTX starts in main()
init ...done
readyQueue = 1 --> NULL
loading /bin/u1 to segment 0x2000 ... done
Proc 0 forked a child 1 at segment=0x2000
timer init
P0 running
P0 switch process
main0: s=
enter main() : argc = 0
=====
I am proc 1 in U mode: segment=0x2000
***** Menu *****
* ps chname kmode switch wait exit fork exec vfork color *
*****
Command ? color
input color [r:g:b] :
=====
I am proc 1 in U mode: segment=0x2000
***** Menu *****
* ps chname kmode switch wait exit fork exec vfork color *
*****
Command ? 
00:00:52

```

Figure 10.2. Demonstration of Timer and Display Drivers

10.3. Keyboard Driver

In this section, we shall develop a simple keyboard driver for the MTX kernel. Instead of showing a complete driver all at once, we shall develop the keyboard driver in several steps. First, we focus on the design principles of interrupt-driven device drivers and present a general framework that is applicable to all interrupt-driven drivers. For the keyboard driver, we begin by considering only lower case ASCII keys first, in order not to be distracted by the details of handling special keys. After showing the basic driver design, we extend it to handle upper case and special keys. The last step is to link the keyboard driver with user mode processes, allowing them to get chars in RAW mode (get keys as they are) or COOKED mode (after processing special keys).

10.3.1. The Keyboard Hardware

Instead of ACSII code, the keyboard of IBM compatible PCs generates scan codes. A complete listing of scan codes is included in the keyboard driver. Translation of scan code to ASCII is by mapping tables in software. This allows the same keyboard to be used for different languages. For each key typed, the keyboard generates two interrupts; one when the key is pressed and another one when the key is released. The scan code of key release is $0x80 +$ the scan code of key press, i.e. bit-7 is 0 for key press and 1 for key release. When the keyboard interrupts, the scan code is in the data port (0x60) of the keyboard interface. The interrupt handler reads the data port to get the scan code and acknowledges the input by strobe PORT_B at 0x61. Some special keys generate escape key sequences, e.g. the UP arrow key generates 0xE048, where 0xE0 is the escape key itself. The following shows the mapping tables for translating scan codes into ASCII. The keyboard has 105 keys. Scan codes above 0x39 (57) are special keys, which cannot be mapped directly, so they are not shown in the key maps. Such special keys are recognized by the driver and handled accordingly. Figure 10.3 shows the key mapping tables.

```
#define NSCAN 58
/* Scan codes to ASCII for unshifted keys */
char unshift[NSCAN] = { // NSCAN=58
    0, 033, '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-', '=', '\b', '\t',
    'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', '[', ']', '\r', 0, 'a', 's',
    'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', ' ', 0, 0, 0, 0, 'z', 'x', 'c', 'v',
    'b', 'n', 'm', ',', '.', '/', 0, '*', 0, ' ' };
/* Scan codes to ASCII for shifted keys */
char shift[NSCAN] = {
    0, 033, '!', '@', '#', '$', '%', '^', '&', '*', '(', ')', '_', '+', '\b', '\t',
    'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P', '{', '}', '\r', 0, 'A', 'S',
    'D', 'F', 'G', 'H', 'J', 'K', 'L', ':', ' ', '~', 0, '|', 'Z', 'X', 'C', 'V',
    'B', 'N', 'M', '<', '>', '?', 0, '*', 0, ' ' };
```

Figure 10.3. Key Mapping Tables

10.3.2. Interrupt-Driven Driver Design

Every interrupt-driven device driver consists of three parts; a lower-half part, which is the interrupt handler, an upper-half part, which is called by processes and a common data

area, which are shared by the lower and upper parts. Figure 10.4 shows the organization of the keyboard driver. The top of the figure shows kbd_init(), which initialize the KBD driver when the system starts. The middle part shows the control and data flow path from the KBD device to a process. The bottom part shows the lower-half, input buffer, and the upper-half organization of the KBD driver.

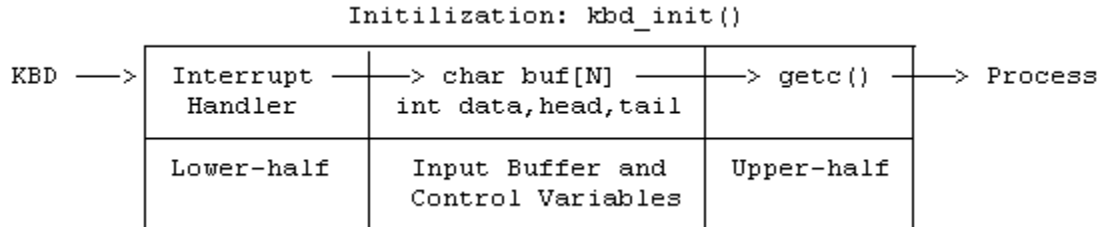


Figure 10.4. KBD Driver Organization

When MTX boots up, the keyboard hardware is already initialized by BIOS and is ready for use. `kbd_init()` only initializes the driver's control variables. When a key is typed, the KBD generates an interrupt, causing the interrupt handler to be executed. The interrupt handler fetches the scan code from KBD's data port. For normal key presses, it translates the scan code into ASCII, enters the ASCII char into an input buffer, `buf[N]`, and notifies the upper-half of the input char. When a process needs an input char, it calls `getc()` of the upper-half driver, trying to get a char from `buf`. The process waits if there is no char in `buf`. The control variable, `data`, is used to synchronize the interrupt handler and process. The choice of the control variable depends on the synchronization tool used. To begin with, we shall use sleep/wakeup for synchronization. The following shows the C code of a simple KBD driver. The driver handles only lower case keys. Upper case and special keys will be considered later.

10.3.3. A Simple Keyboard Driver using sleep/wakeup

```

/***** A simple KBD Driver *****/
#define KEYBD    0x60    // I/O port for keyboard data
#define PORT_B   0x61    // port_B of 8255
#define KBIT     0x80    // bit used to ack chars to keyboard
#define KBLEN    64     // size of input buffer in bytes
#define NSCAN    58     // number of scan codes to map
#define SPACE    0x39    // space char, above which are special keys
#include "keymap.c"      // keymap tables shown above
struct kbd{
    char buf[KBLEN];     // CIRCULAR buffer for input chars
    int head, tail;
    int data;            // number of chars in buf[ ]
} kbd;                  // keyboard driver structure
int kbd_init()
{
    kbd.head = kbd.tail = kbd.data = 0;
    enable_irq(1);       // enable IRQ1
    out_byte(0x20, 0x20);
}

int kbhandler()
{ int scode, value, c;
  
```

```

// get scan code from keyboard's data port and ack it.
scode = in_byte(KEYBD);           // get scan code
value = in_byte(PORT_B);          // strobe PORT_B to ack the key
out_byte(PORT_B, value | KBIT);   // first, set the bit high
out_byte(PORT_B, value);          // then set it low
//printf("kbd interrupt %x\n", scode);
if (scode & 0x80 || scode > SPACE) // ignore release OR special keys
    goto out;
c = unshift[scode];               // map scan code to ASCII char
if (kbd.data == KBLEN){           // sound warning if buf is full
    printf("%c kbd buf FULL!\n", 007);
    goto out;
}
kbd.buf[kbd.head++] = c;          // enter ASCII char into buf[ ]
kbd.head %= KBLEN;               // buf[ ] is circular
kbd.data++;
wakeup(&kbd.data);               // wakeup process in upper half
out:
    out_byte(0x20, 0x20);         // send EOI to 8259 PIC controller
}

/***** upper-half driver routine *****/
int getc()
{
    u8 c;
    lock();                       // kbd.buf[ ] and kbd.data form a CR
    while (kbd.data==0){          // between process and interrupt handler
        unlock();
        sleep(&kbd.data);        // wait for data
        lock();
    }
    c = kbd.buf[kbd.tail++];
    kbd.tail %= KBLEN;
    kbd.data--;
    unlock();
    return c;
}

```

The simple KBD driver is intended mainly to illustrate the design principle of an interrupt-driver input device driver. The driver's buffer and control variables form a critical region since they are accessed by both process and interrupt handler. When the interrupt handler executes, the process is logically not executing. So process can not interfere with interrupt handler. However, while a process executes, interrupts may occur, which diverts the process to execute the interrupt handler, which may interfere with the process. For this reason, when a process calls `getc()`, it must mask out interrupts to prevent keyboard interrupts from occurring. Then it checks whether there are any keys in the input buffer. If there is no key, it sleeps with interrupt enabled. Upon waking up, it disables interrupts before checking for input keys again. If there are input keys, it gets a key, modifies the input buffer and the number of input keys while still inside the critical region. Finally, it enables interrupts before return a key. On the x86 CPU it is not possible to mask out only keyboard interrupts. The `lock()` operation masks out all interrupts, which is a little overkill but it gets the job done. Alternatively, we may write to the PIC mask register to disable/enable the keyboard interrupt.

10.3.4. A Simple Keyboard Driver using Semaphore

Instead of sleep/wakeup, we may also use semaphore to synchronize the process and interrupt handler. In that case, we only need to define data as a semaphore with a 0 initial value and replace sleep/wakeup with P/V on the semaphore, as shown below.

```
struct kbd{
    char buf[KBLLEN];
    int head, tail;
    SEMAPHORE data;          // use semaphore for synchronization
}kbd;

int kbd_init()
{
    kbd.head = kbd.tail = 0;
    kbd.data.value = kbd.data.queue = 0; // initialize semaphore to 0
}
int kbhandler()
{
    /***** same as before ****/
    V(&kbd.data);
out:
    out_byte(0x20, 0x20);
}
int getc()
{
    u8 c;
    P(&kbd.data); // wait for input key if necessary
    lock();
    c = kbd.buf[kbd.tail++];
    kbd.tail %= KBLLEN;
    unlock();
    return c;
}
```

The advantages of using a semaphore is that it combines a counter, testing the counter and deciding whether to proceed or not all in an indivisible operation. In `getc()`, when a process passes through the `P()` statement, it is guaranteed to have a key in the input buffer. The process does not have to retry as in the case of using sleep/wakeup. However, when implementing input device drivers there is a practical problem, which must be handled properly. While a process waits for terminal inputs, it is usually interruptible. The process may be woken up or unblocked by a signal even if no key is entered. When the process resumes, it still tries to get a key. We must adjust the input buffer, e.g. by inserting a dummy key for the process to get. Otherwise, the buffer's head and tail pointers would be inconsistent, causing the next process to get wrong keys.

10.3.5. Interrupt Handlers Should Never Sleep or Block

In the KBD driver, the interrupt handler puts chars into the input buffer and process gets chars from the input buffer. It looks like a pipe but there is a major difference. In the pipe case, both ends are processes. In the KBD driver case, the write-end is not a process but an interrupt handler. Unlike a process, an interrupt handler should never sleep or become blocked, for the following reasons.

- (1). Sleeping or blocking applies only to a process, not to a piece of code. When an interrupt occurs, the current running process is diverted to handle the interrupt, but we do not know which process that is. If an interrupt handler sleeps or blocks, it would cause the interrupted process to sleep or block. The interrupted process may have nothing to do with the interrupt reason. It happens to be the running process when the interrupt occurs, and it is performing the interrupt processing service. Blocking such a good samaritan process logically does not make sense.
- (2). Blocking a process implies process switch. In all uniprocessor Unix-like systems, switch process in kernel mode is not allowed because the entire kernel is a critical region from process point of view.
- (3). If the interrupted process was executing in user mode, switch process is possible, but this would leave the interrupt handler in an unfinished state, rendering the interrupting device unusable for an unknown period of time. If the switched process terminates, the interrupt handler would never finish. This amounts to a lost interrupt, which confuses the device driver.
- (4). In a multiprocessor system, if the interrupted process has set a lock and the interrupt handler tries to acquire the same lock again, the process would lock itself out.

Therefore, an interrupt handler should only do wakeup or unblocking operations but should never sleep or become blocked. For instance, in the KBD driver the interrupt handler never waits. If the input buffer is full, it simply sounds a warning beep, discards the input char and returns.

10.3.6. Raw Mode and Cooked Mode

In the KBD driver, the interrupt handler deposits keys into the input buffer without echoing them. This has two effects. The first one is that, if no process is getting input keys and echoing them, the user would not see the keys as they are typed. The second effect is that the input buffer contains raw keys. For example, when a user types 'a', followed by backspace '\b' and then 'c', the normal interpretation is that the user intends to erase the last 'a' and replace it with 'c'. In raw mode, all three keys are entered into the input buffer. The interpretation of these keys and their visual effects on the display screen are left for the process to decide. Raw inputs are often necessary. For example, when executing a full screen text editor a process may need raw keys to move the cursor and process the edited text. If a process only needs input lines, as most processes executing in command-line mode do, the raw inputs must be processed or cooked into lines. Again, this is left for the process to decide outside of the KBD driver.

10.3.7. Foreground and Background Processes

The KBD driver works fine if only one process tries to get inputs from the keyboard. If several processes try to get inputs from the keyboard, the results may differ depending on the synchronization tools used. In Unix, wakeup() wakes up all processes sleeping on the same event without any specific order. If several processes are sleeping for inputs, each input key will wake up all such processes but only one process will get the key, so that each process may get a different key. The result is that no process can get a complete line,

which is clearly unacceptable. Using semaphores makes the situation even worse. If several processes try to get input keys, they are all blocked in the semaphore queue of the KBD driver. As input chars are entered, the processes are unblocked one at a time, each gets a different key. The result is the same as before; namely no process can get a complete line. In the simple KBD driver using sleep/wakeup for synchronization, sleeping processes are maintained in a FIFO sleepList. If we run several processes under MTX, only one process will get all the input keys. The reader may verify this and try to figure out why? It turns out that the problem can not be solved by the driver alone. Some protocol at a higher level is needed. In practice, when several processes share the same input device, e.g. a keyboard, only one of the processes is designated as the foreground process. All others, if any, are in the background. Only the foreground process is allowed to get inputs. A special fg command can be used to promote a background process to the foreground, which automatically demotes the original foreground process to background. In MTX, this is implemented by a lock semaphore in the KBD driver. Alternative ways to implement foreground and background processes are listed as programming exercises in the Problem section.

10.3.8. Improved KBD Driver

The following shows an improved keyboard driver, which handles upper case keys and some special keys. It uses semaphores for synchronization.

```

/*****
      An improved KBD driver: kbd.c file
*****/
#define KEYBD    0x60      // I/O port for keyboard data
#define PORT_B   0x61      // port_B of 8255
#define KBIT     0x80      // ack bit
#define NSCAN    58        // number of scan codes
#define KBLEN    64
#define BELL     0x07
#define F1       0x3B      // scan code of function keys
#define F2       0x3C
#define CAPSLOCK 0x3A      // scan code of special keys
#define LSHIFT   0x2A
#define RSHIFT   0x36
#define CONTROL  0x1D
#define ALT      0x38
#define DEL      0x53

#include "keymap.c"
struct kbd{
    char buf[KBLEN];
    int head, tail;
    SEMAPHORE data;          // semaphore between inth and process
    SEMAPHORE lock;          // ONE active process at a time
    struct proc *blist;      // (background) PROCS on the KBD
    int intr, kill, xon, xoff; // examples of KBD control keys
}kbd;
int alt, capslock, esc, shifted, control, arrowKey; // state variables

int kbd_init()

```

```

{
    esc=alt=control=shifted=capslock = 0; // clear state variables
    kbd.head = kbd.tail = 0;
    // define default control keys
    kbd.intr = 0x03;      // Control-C
    kbd.kill = 0x0B;      // Control-K
    kbd.xon  = 0x11;      // Control-Q
    kbd.xoff = 0x13;      // Control-S
    // initialize semaphores
    kbd.data.value = 0; kbd.data.queue = 0;
    kbd.lock.value = 1; kbd.lock.queue = 0;
    kbd.blist = 0;        // no background PROCs initially
    enable_irq(1);
    out_byte(0x20, 0x20);
}
/***** lower-half driver *****/
int kbhandler()
{
    int scode, value, c;
    // Fetch scab code from the keyboard hardware and acknowledge it.
    scode = in_byte(KEYBD);    // get scan code
    value = in_byte(PORT_B);    // strobe PORT_B to ack the char
    out_byte(PORT_B, value | KBIT);
    out_byte(PORT_B, value);

    if (scode == 0xE0)        // ESC key
        esc++;                // inc esc count by 1
    if (esc && esc < 2)        // only the first ESC key, wait for next code
        goto out;
    if (esc == 2){            // two 0xE0 means escape sequence key release
        if (scode == 0xE0)    // this is the 2nd ESC, real code comes next
            goto out;
        // with esc==2, this must be the actual scan code, so handle it
        scode &= 0x7F;        // leading bit off
        if (scode == 0x38){    // Right Alt
            alt = 0;
            goto out;
        }
        if (scode == 0x1D){    // Right Control release
            control = 0;
            goto out;
        }
        if (scode == 0x48)        // up arrow
            arrowKey = 0x0B;
        esc = 0;
        goto out;
    }
    if (scode & 0x80){ // key release: ONLY catch shift,control,alt
        scode &= 0x7F;        // mask out bit 7
        if (scode == LSHIFT || scode == RSHIFT)
            shifted = 0;        // released the shift key
        if (scode == CONTROL)
            control = 0;        // released the Control key
        if (scode == ALT)
            alt = 0;            // released the ALT key
        goto out;
    }
}

```

```

// from here on, must be key press
if (scode == 1) // Esc key on keyboard
    goto out;
if (scode == LSHIFT || scode == RSHIFT){
    shifted = 1; // set shifted flag
    goto out;
}
if (scode == ALT){
    alt = 1;
    goto out;
}
if (scode == CONTROL){
    control = 1;
    goto out;
}
if (scode == 0x3A){
    capslock = 1 - capslock; // capslock key acts like a toggle
    goto out;
}
if (control && alt && scode == DEL){
    printf("3-finger salute\n");
    goto out;
}
/***** Catch and handle F keys for debugging *****/
if (scode == F1){ do_F1(); goto out;}
if (scode == F2){ do_F2(); goto out;} // etc
// translate scan code to ASCII, using shift[ ] table if shifted;
c = (shifted ? shift[scode] : unshift[scode]);

// Convert all to upper case if capslock is on
if (capslock){
    if (c >= 'A' && c <= 'Z')
        c += 'a' - 'A';
    else if (c >= 'a' && c <= 'z')
        c -= 'a' - 'A';
}
if (control && (c=='c' || c=='C')){ // Control-C on PC are 2 keys
    //Control-C Key; send SIGINT(2) signal to processes on console;
    c = '\n'; // force a line, let procs handle SIG#2 when exit Kmode
}
if (control && (c=='d' || c=='D')){ // Control-D, these are 2 keys
    printf("Control-D: set code=4 to let process handle EOF\n");
    c = 4; // Control-D
}
/* enter the char in kbd.buf[ ] for process to get */
if (kbd.data.value == KBLEN){
    printf("%c\n", BELL);
    goto out; // kb.buf[] already FULL
}
kbd.buf[kbd.head++] = c;
kbd.head %= KBLEN;
V(&kbd.data);
out:
    out_byte(0x20, 0x20); // send EOI
}
/***** upper-half driver *****/
int getc()

```

```

{
    u8 c;
    if (running->fgground==0)
        P(&kbd.lock);          // only foreground proc can getc() from KBD
    P(&kbd.data);
    lock();
    c = kbd.buf[kbd.tail++]; kbd.tail %= KBLLEN;
    unlock();
    return c;
}

```

In the KBD structure of the improved keyboard driver, data is a semaphore with initial value 0. The semaphore lock (initial value=1) ensures that only one process can get inputs from the keyboard. In addition, the KBD structure also has a few other control variables. For example, intr defines the interrupt key, which generates a SIGINT(2) signal to all processes on the keyboard. It is set to Control-C by default. The kill key (Control-K) is for erasing an entire input line, etc. The control variables are usually managed by tty() and ioctl() syscalls. The improved KBD driver is intended to show how to detect and handle some of the special keys, which are explained below.

(1). The KBD driver uses the state variables, esc, control, shifted, alt to keep track of the states of the driver. For example, esc=1 when it sees the first ESC (0xE0). With esc=2, the next scan code determines the actual special key, which also clears esc to 0.

(2). Key press or release: Key release is of interest only to shift, control and Alt keys because releasing such keys resets the state variables shifted, control and alt.

(3). Special key processing examples:

.Function Keys: These are the simplest to recognize. Some of the function keys may be used as hot keys to perform specific functions. For example, pressing the F1 key may invoke ps() to print the process status in kernel.

.Shift key: (LeftShift=0x2A, RightShift=0x36): When either shift key is pressed but not yet released, set shifted to 1 and use the shifted keymap to translate the input chars. Releasing either LSHIFT or RSHIFT key resets shifted to 0.

.Control key sequence: Control-C: Left CONTROL key (0x1D) down but not released, set control=1. Releasing the CONTROL key resets control to 0. With control=1 and a key press, determine the Control-key value.

.Esc Key sequence: Many special keys, e.g. arrow keys, Delete, etc. generate scan code E0xx, where E0 is the ESC key. After seeing 2 ESC keys, set esc=2. With esc=2, get the next key press, e.g. UP arrow=E048, to determine the special key value.

10.3.9. MTX10.vtk: Demonstration of Keyboard Driver

MTX10.vtk is a sample system which includes the display, timer and keyboard drivers. The keyboard driver handles upper and lower cases keys and some special keys. It also supports foreground and background processes. When MTX10.vtk boots up, it runs P1 as the foreground process in Umode. The reader may fork several background processes and use the fg command to change the active process on the keyboard. Figure 10.5 shows the screen of running MTX10.vtk.

```

QEMU
MTX starts in main()
init ....done
kbinit()
kbinit done
readyQueue = 1 --> NULL
loading /bin/u1 to segment 0x2000 .... done
Proc 0 forked a child 1 at segment=0x2000
timer init
=====
I am proc 1 in U mode: segment=0x2000 active=1
***** Menu *****
* ps cname kmode switch wait exit fork exec color fg *
*****
Command ? color
input color [rigiciy] :
=====
I am proc 1 in U mode: segment=0x2000 active=1
***** Menu *****
* ps cname kmode switch wait exit fork exec color fg *
*****
Command ? Function Key F2
Control-C Keys : Send INTR signal to process
enter CAPS and lower case keys
00:02:02

```

Figure 10.5. Display Screen of MTX10.vtk

10.4. Parallel Printer Driver

10.4.1 Parallel Port Interface

A PC usually has two parallel ports, denoted by LPT1 and LPT2, which support parallel printers. Each parallel port has a data register, a status register and a command register. Figure 10.6 shows the parallel port addresses and the register contents.

Parallel ports:	LPT1	LPT2						
DATA	0x378	0x3BC						
STATUS	0x379	0x3BD						
COMMAND	0x37A	0x3BE						
	7	6	5	4	3	2	1	0
Status :	NOTBUSY	-	NOPAPER	SELECT	NOERROR	-	-	-
Command:	-	-	-	EnableIRQ	INIT	SELECT	-	STROBE

Figure 10.6. Parallel Port Address and Registers

Before printing, the printer must be initialized once by:

- INIT : write 0x08 to COMMAND register first, then
- select : write 0x0C to COMMAND register.

To print, write a char to DATA register. Then strobe the printer once by writing a 1, followed by a 0, to bit 0 of the command register, which simulates a strobe pulse of width > 0.5 usec. For example, writing 0x1D = 00011101 followed by 0x1C = 00011100 strobcs the printer with interrupts enabled. When the printer is ready to accept the next char, it interrupts at IRQ7, which uses vector 15. In the interrupt handler, read the status register and check for error. If no error, send the next char to the printer and strobe it once again, etc.

10.4.2. A Simple Printer Driver

Like any interrupt-driven device driver, a printer driver consists of an upper-half part and a lower-half part, which share a common buffer area with control variables. The following shows a simple printer driver. We assume that each process prints one line at a time, which is controlled by a mutex semaphore. Process and interrupt handler share a circular buffer, `pbuf[PLEN]`, with head and tail pointers. To print a line, a process calls `prline()`, which calls `prchar()` to deposit chars into `pbuf`, waiting for room if necessary. It prints the first char if the printer is not printing. After writing the entire line to `pbuf`, the process waits for completion by `P(done)`. The interrupt handler will print the remaining chars from `pbuf`. When a line is completely printed, it `V(done)` to unblock the process, which `V(mutex)` to allow another process to print.

```
/****** C Code of a Simple Printer Driver *****/
#define NPR      1
#define PORT     0x378    // #define PORT 0x3BC for LPT2
#define STATUS   PORT+1
#define CMD      PORT+2
#define PLEN     128
#include "semaphore.c"    // SEMAPHORE type and P/V operations
struct para{           // printer data structure
    int port;           // I/O port address
    int printing;       // 1 if printer is printing
    char pbuf[PLEN];    // circular buffer
    int head, tail;
    SEMAPHORE mutex, room, done; // control semaphores
}printer[NPR];
pr_init()              // initialize or reset printer
{
    struct para *p;
    p = &printer[0];    // assume only one printer
    printf("pr_init %x\n", PORT);
    p->port = PORT;
    p->head = p->tail = 0;
    p->mutex.value = 1;   p->mutex.queue = 0;
    p->room.value = PLEN; p->room.queue = 0;
    p->done.value = 0;    p->done.queue = 0;
    /* initialize printer at PORT */
    out_byte(p->port+2, 0x08); // init
    out_byte(p->port+2, 0x0C); // int, init, select on
    enable_irq(7);
    p->printing = 0;      // is NOT printing now
}
int strobe(struct para *p)
{
    out_byte(p->port+2, 0x1D); // may need delay time here
    out_byte(p->port+2, 0x1C);
}
/****** Lower-half printer driver *****/
int phandler()          // interrupt handler
{
    u8 status, c;
    struct para *p = &printer[0];
    status = in_byte(p->port+1);
    //printf("printer interrupt status = %x\n", status);
    if (status & 0x08){    // check for noError status only
        if (p->room.value == PLEN){ // pbuf[] empty, nothing to print
```



```

        out_byte(p->port+2, 0x0C); // turn off printer interrupts
        V(&p->done);                // tell process print is DONE
        p->printing = 0;            // is no longer printing
        goto out;
    }
    c = p->pbuf[p->tail++];          // print next char
    p->tail %= PLEN;
    out_byte(p->port, c);
    strobe(p);
    V(&p->room);
    goto out;
}
// abnormal status: should handle it but ignored here
out: out_byte(0x20, 0x20);          // issue EOI
}
/***** Upper-half printer driver *****/
int prchar(char c)
{
    struct para *p = &printer[0];
    P(&p->room);                    // wait for room in pbuf[]
    lock();
    if (p->printing==0){            // print the char
        out_byte(p->port, c);
        strobe(p);
        p->printing = 1;
        unlock();
        return;
    }
    // already printing, enter char into pbuf[]
    p->pbuf[p->head++] = c;
    p->head %= PLEN;
    unlock();
}
int prline(char *line)
{
    struct para *p = &printer;
    P(&p->mutex);                    // one process prints LINE at a time
    while (*line)
        prchar(*line++);          // print chars
    P(&p->done);                    // wait until pbuf[ ] is DONE
    V(&p->mutex);                  // allow another process to print
}

```

10.4.3. MTX10.pr: Demonstration of Printer Driver

MTX10.pr demonstrates the printer driver. To test the printer driver on a virtual machine, configure the VM's parallel port to use a pseudo-terminal. Boot up MTX and run the pr command from user mode. Figure 10.7 shows the screen of running MTX10.pr on QEMU using /dev/pts/3 as the parallel port. For each interrupt, the printer driver displays a "printer interrupt" message. Eventually, the printer will be represented by a special file, /dev/lp0, in the MTX file system. In that case, writing to the special file invokes the printer driver.

```

23372 pts/3    00:00:00 bash
23387 pts/3    00:00:00 ps
root@wang:~# test
root@wang:~#

=====
I am proc 1 in U mode: segment=0x2000
***** Menu *****
* ps chname kmode switch wait exit fork exec pr *
*****
Command ? pr
input a line to print : test
line=test
printer interrupt: status=0x58
printer interrupt: status=0x58
printer interrupt: status=0x58
printer interrupt: status=0x58
printer interrupt: status=0x58
printer interrupt: status=0x58
printing done

```

Figure 10.7. Demonstration of Printer Driver

10.5. Serial Port Driver

10.5.1. RS232C Serial Protocol

RS232C is the standard protocol for serial communication. The RS232C protocol defines 25 signal lines between a Data Terminal Equipment (DTE) and a Data Communication Equipment (DCE). Usually, a computer is a DTE, a modem or serial printer is a DCE. Figure 10.8 shows the most commonly used signal lines and data flow directions of the serial interface.

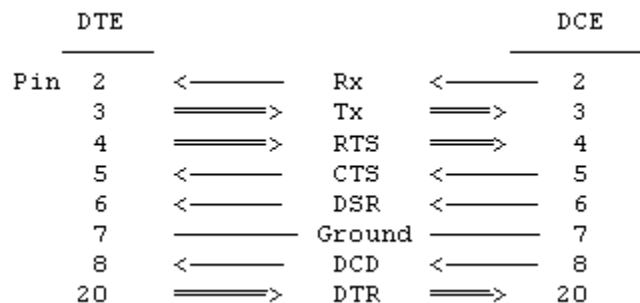


Figure 10.8. RS232C Signal Lines

To establish a serial connection between a DTE and a DCE, a sequence of events goes like this. First, the DTE asserts the DTR (Data Terminal Ready), which informs the DCE that the DTE is ready. The DCE responds by asserting DSR (Data Set Ready) and DCD (Data Carrier Detect), which informs the DTE that the DCE is also ready. When the DTE has data to send, it asserts RTS (Request To Send), which asks the DCE for permission to send. If the DCE is ready to receive data, it asserts CTS (Clear To Send). Then the DTE can send data through Tx while CTS is on. The DCE can send data through Rx to DTE anytime without asking for permission first.

10.5.2. Serial Data Format: Figure 10.9 shows the serial data format.

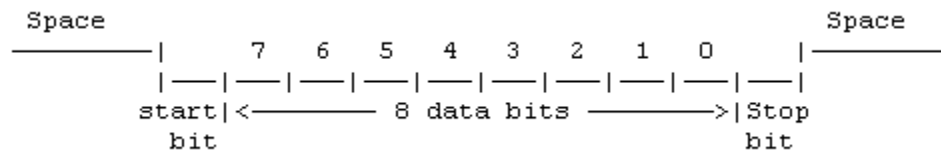


Figure 10.9. Serial Data Format

Bits timing is controlled by a (local) clock, which determines the baud rate. Technically, baud rate refers to the number of signal changes per second. For binary signals, baud rate is the same as bit rate. Each 8-bit data requires a start bit and at least one stop bit. With one stop bit and a baud rate of 9600, the data rate is 960 chars per second.

10.5.3. PC Serial Ports

Most PCs have two serial ports, denoted by COM1 and COM2. Figure 10.10 shows the serial port registers and their contents.

Index	Name	Function	COM1	COM2
0	DATA	Rx or Tx data	3F8	2F8
1	IER	Interrupt Enable	3F9	2F9
2	IIR	Interrupt ID	3FA	2FA
3	LCR	Line Control	3FB	2FB
4	MCR	Modem Control	3FC	2FC
5	LSR	Line Status	3FD	2FD
6	MSR	Modem Status	3FE	2FE

IER: bit0=RX enable, bit1=Tx enable
IIR: bits2-1=error(00), Tx(01) Rx(10), Modem(11)
LCR: number of data bits, stop bits, parity, etc.
MCR: bit0=DTR, bit1=RTS, bit2=enable hardware interrupt
LSR: bit0=Rx data ready, bit5=Tx register empty

Figure 10.10. Serial Ports Registers

10.5.4. Serial Terminals

PC's serial ports are DTE's by default. They are intended to connect to DCE's, such as modems, serial terminals and serial printers, etc. We cannot connect the serial ports of PCs directly because both are DTEs. In order for the connection to work, one must be a DCE, or at least each serial port thinks the other side is a DCE. There are two possible ways to do this.

(1). Configure one of the PC's serial port as a DCE. Then connect the two serial ports by an ordinary DTE to DCE cable, which is wired straight-through.

(2). Make both DTEs think the other side is a DCE by crossing some wires in the cable, as shown in Figure 10.11.

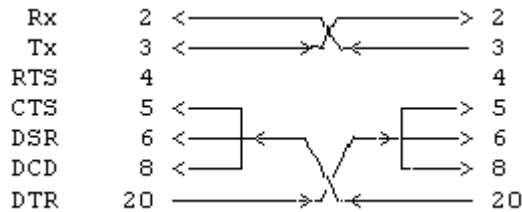


Figure 10.11. DTE-to-DTE Connection

Unlike the PC's console keyboard, which generates scan codes, a serial terminal generates ASCII code directly. The most widely used serial terminal standard is the VT100 protocol. In addition to ordinary ASCII keys, VT100 also supports special keys as escape sequence codes. For example, Cursor UP is ESC [A, Cursor Down is ESC [B, etc. A virtual serial terminal is a device which emulates a real serial terminal. There are many ways to connect a PC to a virtual serial terminal.

- . Connect to a PC running a program which emulates a serial terminal.
- . Enable serial port support of a PC emulator. For example,
QEMU: run QEMU with `-serial` option, e.g. `-serial mon:stdio -serial /dev/pts/2`
VMware/VirtualBox: configure VM to use virtual serial terminals.

It is noted that VMware's virtual serial ports may be sockets, which require a socket interface program, such as socat, for both input and output.

10.5.5. Serial Port Driver Design

10.5.5.1. The STTY structure

Each serial port is represented by a stty structure.

```
struct stty {
    // input section
    char inbuf[BUFLen]; int inhead, intail;
    SEMAPHORE inchars;
    // output section
    char outbuf[BUFLen]; int outhead, outtail;
    SEMAPHORE outspace;
    // Control section
    char erase, kill, intr, x_on, x_off, tx_on;
    // I/O port base address
    int port;
} stty[NSTTY]; // NSTTY = number of serial ports
```

The stty structure consists of three sections. The input section contains a circular buffer, inbuf, for storing input chars and a semaphore, inchars=0, for synchronization. The output section contains a circular buffer, outbuf, for storing outgoing chars, a semaphore, outspace=BUFLen, for synchronization and a tx_on flag for output interrupt enabled. The control section defines control chars, such as intr, echo, kill, etc. The I/O port field stores the serial port base address.

10.5.5.2. Serial Port Initialization

To initialize the serial ports, write values to their control registers in the following order.

```
0x80 to DATA : bit#7=1           // Use DATA, IER registers
0x00 to IER   : 0 value           // for divisor
0x0C to DATA : divisor=12        // divisor=12 for 9600 bauds
0x03 to LCR   : Line Control = 00000011 // no parity, 8_bit data
0x0B to MSR   : Modem Control= 00001011 // turn on IRQ, RTS, DTR
0x01 to IER   : Int Enablee  = 00000001 // Tx off, Rx on.
tx_on = 0;                      // Tx interrupt disabled
```

10.5.5.3. Lower-half Serial Port Driver

The lower-half of the serial port drivers is the interrupt handler, which is set up as follows.

- (1). Set interrupt vectors for COM1 (IRQ4) and COM2 (IRQ3) by
 set_vector(12, s0inth); // vector 12 for COM1
 set_vector(11, s1inth); // vector 11 for COM2

Install interrupt handler entry points by the INTTH macro in assembly:

```
_s0inth: INTTH s0handler      // entry points of
_s1inth: INTTH s1handler      // serial interrupt handlers
```

- (2). Write s0handler() and s1handler() functions, which call shandler(int port):
 int s0handler(){ shandler(0);}
 int s1handler(){ shandler(1);}

- (3). Serial Port Interrupt Handler Function in C:

```
int shandler(int port) // port=0 for COM1; 1 for COM2
{
    struct stty *tty = &stty[port];
    int IntID, LineStatus, ModemStatus, intType;
    IntID = in_byte(tty->port+IIR); // read InterruptID
    LineStatus = in_byte(tty->port+LSR); // read LineStatus
    ModemStatus = in_byte(tty->port+MSR); // read ModemStatus
    intType = IntID & 0x07;      // mask in lowest 3 bits of IntID
    switch(intType){
        case 0 : do_modem(tty);   break;
        case 2 : do_tx(tty);      break;
        case 4 : do_rx(tty);      break;
        case 6 : do_errors(tty);  break;
    }
    out_byte(0x20, 0x20);      // issue EOI to PIC Controller
}
```

Upon entry to shandler(), the port parameter identifies the serial port. The actions of the interrupt handler are as follows.

- . Read interrupt ID register to determine the interrupt cause.
- . Read line and modem status registers to check for errors.
- . Handle the interrupt.

. Issue EOI to signal end of interrupt processing.

In order to keep the driver simple, we shall ignore errors and modem status changes and consider only rx and tx interrupts for data transfer.

(4). Input Interrupt Handler:

```
int do_rx(struct stty *tty)
{
    (1). char c = in_byte(tty->port); // get char from data register
    (2). if (tty->inchars.value >= BUFLen){ // if inbuf is full
        out_byte(tty, BEEP);          // sound BEEP=0x7
        return;
    }
    (3). if (c==0x3){                  // Control_C key
        send SIGINT(2) signal to processes;
        c = '\n';                      // force a line
    }
    tty->inbuf[tty->inhead++] = c; // put char into inbuf
    tty->inhead %= BUFLen;         // advance inhead
    (4). V(&tty->inchars); // inc inchars and unblock sgetc()
}
```

In `do_rx()`, it only catches Control_C as the interrupt key, which sends a SIGINT(2) signal to the process on the terminal and forces a line for the unblocked process to get. It considers other inputs as ordinary keys. The circular input buffer is used in the same way as in the keyboard driver. Similar to the keyboard driver case, we assume that there is a process running on the serial terminal. So `do_rx()` only enters raw inputs into the input buffer. The process will cook the raw inputs into lines and echo the chars.

(5). Output Interrupt Handler:

```
int do_tx(struct stty *tty)
{
    (1). if (no more chars in outbuf[]){ // no more outputs
        turn_off Tx interrupt;
        return;
    }
    (2). char c = tty->outbuf[tty->outtail++]; // output next char
        tty->outtail %= BUFLen;
        out_byte(tty->port, c);
    (3). V(&tty->outspace);
}
```

In `do_tx()`, if there are no more chars to output, it turns off Tx interrupt and returns. Otherwise, it outputs the next char from outbuf and V the outspace semaphore to unblock any process that may be waiting for room in outbuf.

10.5.5.4. Upper-half Serial Port Driver

The upper-half driver functions are `sgetc()` and `sputc()`, which are called by processes to input/output chars from/to the serial port.

```

char sgetc(struct stty *tty)    // return a char from serial port
{
    char c;
    P(&tty->inchars);           // wait if no input char yet
    lock();                     // disable interrupts
    c = tty->inbuf[tty->intail++];
    tty->intail %= BUFLLEN;
    unlock();                   // enable interrupts
    return c;
}
int sputc(struct stty *tty, char c)
{
    P(&tty->outspace);           // wait for space in outbuf[]
    lock();                     // disable interrupts
    tty->outbuf[tty->outhead++] = c;
    tty->outhead %= BUFLLEN;
    if (!tty->tx_on)
        enable_tx(tty);        // turn on tty's tx interrupt;
    unlock();                   // enable interrupts
}

```

When a process calls `sputc()`, it first waits for room in the output buffer. Then it deposits the char into `outbuf` and turns on Tx interrupt, if necessary. The interrupt handler, `do_tx()`, will write out the chars from `outbuf` on successive Tx interrupts. When the interrupt for the last char occurs, the interrupt handler turns off Tx interrupt.

10.5.6. MTX10.serial: Demonstration of Serial Port Driver

MTX10.serial demonstrates the serial port driver. Figure 10.12 shows the screen of running MTX10.serial under QEMU. To test the serial port driver, configure QEMU to use a pseudo-terminal, e.g. `/dev/pts/3`, as serial port. In the pseudo-terminal, enter `sleep 10000` to suspend the Linux `sh` process. After booting up MTX, serial port I/O can be done in either Kmode or Umode. In Kmode the commands are

```

i : get a line from serial port;
o : send a line to serial port

```

In Umode, the commands are `sin` for input a line and `sout` for output a line. On inputs, it prints a rx interrupt for each char. On outputs, it prints a tx interrupt for each line. In the demonstration system, we assume that no process is running on the serial terminal to echo the inputs. In order to let the user see the input chars, `do_rx()` uses a small echo buffer to echo the inputs and it also handles the backspace key. The reader may consult the driver code for details. In the final versions of MTX, we shall use the serial port driver to support multiple user logins from serial terminals.

```

Serial Port Ready
serial line from Umode
test
|

=====
I am task 1 in U mode: segment=0x2000
***** Menu *****
* ps cname kmode switch wait exit fork exec sin sout *
*****
Command ? sout
outline = serial line from Umode
tx: done with a line
=====
I am task 1 in U mode: segment=0x2000
***** Menu *****
* ps cname kmode switch wait exit fork exec sin sout *
*****
Command ? sin
getline rx:c=t rx:c=e rx:c=s rx:c=t rx:c=
rx: has a line uiline=test
=====

```

Figure 10.12. Demonstration of Serial Port Driver

10.6. Floppy Disk Driver

10.6.1. FDC Controller

The PC's floppy disk controller (FDC) uses the NEC μ PD765 or Intel 82072A floppy disk controller chip. For basic FD operations, they use the same programming interface. The FDC has several registers at the I/O port addresses shown in Figure 10.13.

Register	Primary	7	6	5	4	3	2	1	0
DOR	0x3F2	motor-on: B A				IRQ-DMA ~RESET DRIVE			
STATUS	0x3F4	RDY DIR DMA BSY				drive is seeking			
DATA	0x3F5	Data reg : write commands, read results							
ST0-ST2	0x3F5	Result status; read from DATA register							

Figure 10.13. FDC Registers

The status bits in ST0-ST2 are listed in the driver code. The reader may also consult FDC documentations [FDC] for more information. The FDC uses channel 2 DMA of the ISA bus for data transfer. When executing a R/W command the sequence of operations is as follows.

10.6.2. FDC R/W Operation Sequence

- (1). Turn on the drive motor with the IRQ_DMA bit set, wait until drive motor is up to running speed, which can be done by either a delay or a timer request. Also, set a timer

request to turn off the drive motor with an appropriate delay time, e.g. 2 seconds, after the command is finished.

(2). If needed, reset, recalibrate the drive and seek to the right cylinder.

(3). Set up the DMA controller for data transfer.

(4). Poll the main status register until drive is ready to accept command.

(5). Write a sequence of command bytes to the DATA register. For example, in a read/write sector operation, the 9 command bytes are:

opcode, head+drive, cyl, head, sector, // drive=00,01,10,11

bytes_per_sector, sector_per_track, gap_length, data_length.

For a specific drive type, the last 4 command bytes are always the same and can be regarded as constants.

(6). After executing the command, the FDC will interrupt at IRQ6. When a FD interrupt occurs, if the command has a result phase, the driver can read the result status ST0-ST2 from DATA register. Some commands do not have a result phase e.g. recalibrate and seek, which require an additional sense command to be sent in order to read the status bytes ST0-ST2.

(7). Wait for FDC interrupt; send a sense command if needed. Read results from DATA register to get status ST0-ST2 and check for error. If error, retry steps (2)-(7).

(8). If no error, the command is finished. Start next command. If no more commands in 2 seconds, turn off the drive motor by timer.

10.6.3. FD Driver Design

(1). Floppy drive data structure: Each drive is represented by a floppy structure.

```
struct floppy {    // drive struct, one per drive
    u16 opcode;    // FDC_READ or FDC_WRITE
    u16 cylinder;  // cylinder number
    u16 sector;    // sector : counts from 1 NOT 0
    u16 head;      // head number
    u16 count;     // byte count (BLOCK_SIZE)
    u16 address;   // virtual address of data area
    u16 curcyl;    // current cylinder number
    u8  results[7]; // each cmd may generate up to 7 result bytes
    u8  calibration; // drive is CALIBRATED or UNCALIBRATED flag
} floppy[NFDC];    // NFDC = 1
```

In order to keep the driver simple, we assume that the FD driver supports only one 1.44 MB drive for READ/WRITE operations. It should be quite easy to extend the driver to support more drives and other operations, such as format, etc.

(2). FD semaphores: the FD driver uses two semaphores for synchronization:

```
SEMAPHORE fdio=1; // process execute FD driver one at a time,
SEMAPHORE fdsem=0; // for process to wait for FD interrupts.
```

(3). fd_init(): initialize the FD structure, including semaphores and status flags. Write 0x0C to DOR register, which turns A: drives's motor off and sets the drive with interrupts enabled and using DMA.

```

int fd_init()
{
    struct floppy *fp = floppy[0];
    fdio.value = 1; fdsem.queue = 0;
    fdsem.value = fdsem.queue = 0;
    fp->curcyl = 0; fp->calibration = 0;
    need_reset = 0;
    motor_status = 0x0C;    // 0x0C;          (0x0D for B drive)
    out_byte(DOR, 0x0C);    // DOR=00001100 (0x0D for B drive)
}

```

(4). Upper-half FD driver: the upper-half of the FD driver includes almost the entire driver's C code. In the FD driver, the process runs in several stages, which are interleaved by FD interrupts. After issuing a command, it waits for interrupt by P(fdsem). When the interrupt occurs, the FD interrupt handler does V(fdsem) to unblock the process, which reads and checks the result status and continues to the next stage, etc.

(5). Lower-half FD driver: The FD interrupt handler is very simple. All it does is to issue V(fdsem) to unblock the process. Then it issues EOI and returns.

(6). FD motor on/off control: The motor status is maintained in a motor_status variable. After turning on the drive motor, the process sets a timer request of 20 ticks and waits on the fdsem semaphore for the drive motor to get up to speed. After 20 ticks, the timer interrupt handler unblocks the process. For each read/write operation, it also sets 120 timer ticks to turn off the motor, which is done by the timer interrupt handler.

(7). Main FD driver function: read/write a disk block to an address in kernel space:

```

int fdrw(u16 rw, u16 blk, char *addr)
{
    struct floppy *fp = floppy[0];
    int r, i;
    P(&fdio);    // one process executes fd_rw() at a time
    motor_on();  // turn on motor if needed, set 2 sec. turn off time
    fp->opcode = rw;    // opcode
    r = (2*blk) % CYL_SIZE;    // compute CHS
    fp->cylinder = (2*blk) / CYL_SIZE;
    fp->head = r / TRK_SIZE;
    fp->sector = (r % TRK_SIZE) + 1;    // sector counts from 1, NOT 0
    fp->count = BLOCK_SIZE;    // block size
    fp->address = addr;    // address
    for (i=0; i<MAX_RETRY; i++){    // try MAX_RETRY times
        // First check to see if FDC needs reset
        if (need_reset) // printf("fd reset\n");
            fd_reset();
        // May also need to recalibrate, especially after a reset
        if (fp->calibration == UNCALIBRATED)
            calibrate(fp);
        // Seek to the correct cylinder if needed
        if (fp->curcyl != fp->cylinder)
            seek(fp);
        // Set up DMA controller
        setup_dma(fp);
    }
}

```

```

        // write commands to FDC to start RW
        r = commands(fp);
        if (r==OK) break;
    }
    if (i>=MAX_FD_RETRY)
        printf("FDC error\n");// error, most likely hardware failure
    V(&fdio);                // unlock fdio semaphore
}

```

fdrw() is the main FD driver function. It reads/writes a 1K block (2 sectors) at the linear block number, blk, into the virtual address addr (in Kmode). The reason of reading only 2 sectors is because the FD driver is intended to work with the MTX file system, which uses 1KB I/O buffers for both FD and HD. When a process enters fdwr(), it first locks the fdio semaphore to prevent other process from executing the same FD driver. The fdio lock is released only after the current process has finished the read/write operation. Then it calls motor_on(), which turns on the drive motor and sets the motor-off time to 2 seconds. Then it computes the CHS values of the blk. Then it executes the retry loop up to MAX_FD_RETRY (5) times. In the retry loop, it first ensures that the drive does not need reset, has been calibrated and seek to the right cylinder so that the drive is ready to accept commands. Then it calls setup_dma(), which programs the DMA controller for the intended data transfer. It writes the command bytes to the DATA register. Then it waits for interrupt by P(fdsem). When the expected interrupt occurs, the FD interrupt handler issues V(fdsem) to unblock the process, which continues to read the result status and checks for error. For a recalibrate or seek command, it issues a sense command before reading the status ST0-ST2. Error conditions in the recalibration and seek stages are checked but ignored. Such errors would most likely lead to a R/W error eventually. So we only focus on checking the result status after a R/W command. Any error at this stage causes the driver to re-execute the retry loop. When the R/W operation finishes, either successfully or due to error, the process unlocks the fdio semaphore.

10.6.4. Implications on I/O Buffers in MTX

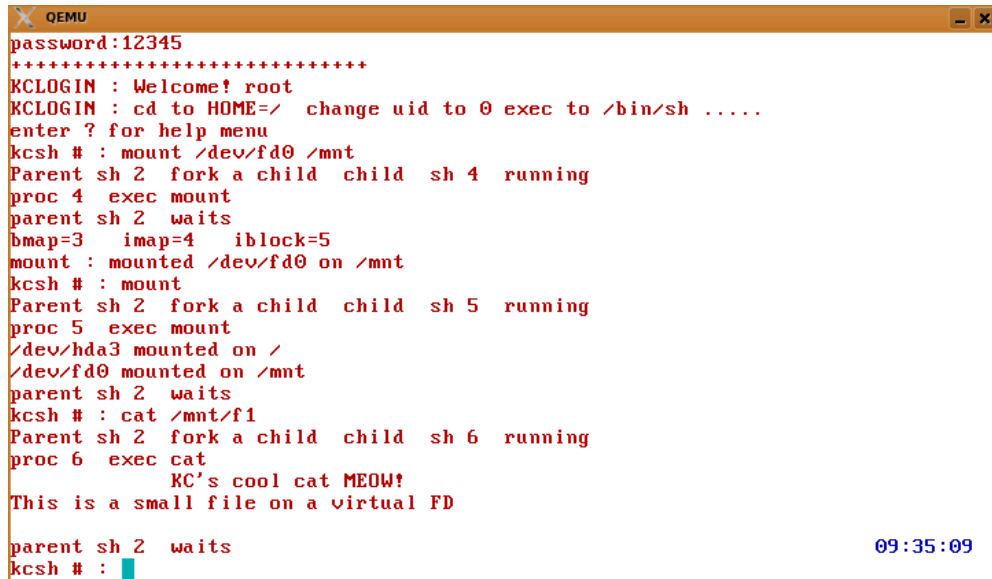
MTX uses I/O buffers for block devices, which include both FD and HD. For HD, all writes to disk blocks use the delay-write policy. When a process writes a disk block, it writes data to a buffer assigned to the disk block, which is marked as dirty for delay-write. A dirty buffer is written out to disk asynchronously only when it is to be reassigned to a different disk block. For removable devices, such as FD disks, delay-write is undesirable. For this reason, the FD driver uses synchronous write to ensure that data are written to the FD disk directly.

10.6.5. Demonstration of FD Driver

MTX10.fd demonstrates the FD driver. Since the FD driver takes over FD interrupts, it is not possible to run MTX on a FD image and operate on a FD at the same time. For this reason, MTX10.fd runs only on HD. The reader may test it on virtual machines by the following steps.

- (1). Run the sh script mk PARTITION qemu|vmware, e.g. mk 3 qemu, which installs a complete rmtx system to partition 3 of a virtual HD named vdisk.
- (2). Run `qemu -had vdisk -fda fd` # fd is a FD image containing an EXT2 FS
- (3). Boot up MTX from partition 3. Mount the virtual FD by `mount /dev/fd0 /mnt`
- (4). Then copy files to/from the mounted FS. When finished, `umount /dev/fd0`.

Figure 10.14 shows the screen of running MTX10.fd. It first mounts the virtual FD. Then it cat the contents of a file from the mounted FD. The reader may also run the cp command to copy files to/from the FD.



```

QEMU
password:12345
*****
KCLOGIN : Welcome! root
KCLOGIN : cd to HOME=/ change uid to 0 exec to /bin/sh .....
enter ? for help menu
kcsh # : mount /dev/fd0 /mnt
Parent sh 2 fork a child child sh 4 running
proc 4 exec mount
parent sh 2 waits
bmap=3 imap=4 iblock=5
mount : mounted /dev/fd0 on /mnt
kcsh # : mount
Parent sh 2 fork a child child sh 5 running
proc 5 exec mount
/dev/hda3 mounted on /
/dev/fd0 mounted on /mnt
parent sh 2 waits
kcsh # : cat /mnt/f1
Parent sh 2 fork a child child sh 6 running
proc 6 exec cat
KC's cool cat MEOW!
This is a small file on a virtual FD
parent sh 2 waits
kcsh # :
09:35:09

```

Figure 10.14. Demonstration of FD driver and mount

10.7. IDE Hard disk Driver

There are two kinds of hard disk interface; Parallel ATA (PATA) and Serial ATA (SATA). PATA is also known as IDE. SATA stands for serial ATA interface, which uses SCSI like packets and is different from IDE interface. In some PCs, SATA can be configured to be backward compatible with PATA but not vice versa. Older PATA uses PIO (Programmed I/O). Newer PATA, e.g. ATA-3, may use either PIO or DMA. In order to use DMA the PC must be in protected mode to access the PCI bus. In general, DMA is faster and more suited to transferring large amounts of data. For small amount of data, e.g. in usual file system operations, it is actually better to use PIO due to its simplicity. This section presents IDE hard disk drivers using PIO. In order to illustrate different driver designs, we shall show three different versions of the HD driver. The first driver relies entirely on the process, which actively controls each r/w operation. In the second driver, process and interrupt handler cooperate, each performs part of the r/w operation. In the third version, the HD driver is an integral part of the block device I/O buffer management subsystem of the MTX kernel. It works with a disk I/O queue and supports both synchronous (blocking) read and asynchronous (non-blocking) write operations.

10.7.1. IDE Interface

A PC usually has two IDE channels, denoted as IDE0 and IDE1. Each IDE channel can support two devices, known as the master and slave. A hard disk drive is usually the master device of IDE0. Each IDE channel has a set of fixed I/O port addresses. Figure 10.15 lists the port addresses of IDE0 and their contents.

```
Control Register:
  0x3F6 = 0x80 (0000 1RE0): R=reset, E=0=enable interrupt
Command Block Registers:
  0x1F0 = Data Port
  0x1F1 = Error
  0x1F2 = Sector Count
  0x1F3 = LBA low byte
  0x1F4 = LBA mid byte
  0x1F5 = LBA hi byte
  0x1F6 = 1B1D TOP4LBA: B=LBA,D=driv
  0x1F7 = Command/status
                                7       6       5       4       3       2       1       0
Status Register(0x1F7)= BUSY READY FAULT SEEK DRQ CORR IDDEX ERROR
Error Register (0x1F1)=  BBK   UNC   MC   IDNF MCR ABRT  TONF AMNF
The error bits are: BBK=Bad Block,UNC=Uncorrectable data error,
MC=Media Changed,IDNF=ID mark Not Found,MCR=Media Change Requested,
ABRT=Aborted,TONF=Track 0 Not Found,AMNF=Address Mark Not Found
```

Figure 10.15. IDE Ports and Registers

10.7.2. IDE R/W Operation Sequence

- (1). Initialize HD: Write 0x08 to Control Register (0x3F6): (E bit=0 to enable interrupt).
- (2). Read Status Register (0x1F7) until drive is not Busy and READY;
- (3). Write sector count, LBA sector number, drive (master=0x00, slave=0x10) to command registers (0x1F2-0x1F6).
- (4). Write READ|WRITE command to Command Register (0x1F7).
- (5). For a write operation, wait until drive is READY and DRQ (drive request for data). Then, write data to data port.
- (6). Each (512-byte) sector R|W generates an interrupt. I/O can be done in two ways:
 - (7a). Process waits for each interrupt. When a sector R/W completes, interrupt handler unblocks process, which continues to read/write the next sector of data from/to data port.
 - (7b). Process starts R/W. For write (multi-sectors) operation, process writes the first sector of data, then waits for the FINAL status interrupt. Interrupt handler transfers remaining sectors of data on each interrupt. When R/W of all sectors finishes, it unblocks the process. This scheme is better because it does not unblock or wakeup processes unnecessarily.
- (8). Error Handling: After each R/W operation or interrupt, read status register. If status.ERROR bit is on, detailed error information is in the error register (0x1F1). Recovery from error may need HD reset.

10.7.3. A Simple HD Driver

Shown below is a simple HD driver based on 10.7.2.(7a). For each `hd_rw()` call, the process starts the r/w of one sector and blocks itself on the `hd_sem` semaphore, waiting for interrupts. When an interrupt occurs, the interrupt handler simply unblocks the process, which continues to r/w the next sector. The `hd_mutex` semaphore is used to ensure that processes execute `hd_rw()` one at a time.

```
/****** A Simple IDE hard disk driver using PIO *****/
#define HD_DATA      0x1F0  // data port for R/W
#define HD_ERROR     0x1F1  // error register
#define HD_SEC_COUNT 0x1F2  // R/W sector count
#define HD_LBA_LOW   0x1F3  // LBA low  byte
#define HD_LBA_MID   0x1F4  // LBA mid  byte
#define HD_LBA_HI    0x1F5  // LBA high byte
#define HD_LBA_DRIVE 0x1F6  // 1B1D0000=>B=LBA,D=drive=>0xE0 or 0xF0
#define HD_CMD       0x1F7  // command : R=0x20 W=0x30
#define HD_STATUS    0x1F7  // status register
#define HD_CONTROL    0x3F6  // 0x08(0000 1RE0):Reset,E=1:NO interrupt
/* HD disk controller command bytes. */
#define HD_READ      0x20  // read
#define HD_WRITE     0x30  // write
#define BAD          -1    // return BAD on error
struct semaphore hd_mutex; // for procs hd_rw() ONE at a time
struct semaphore hd_sem;   // for proc to wait for IDE interrupts
// read_port() reads count words from port to (segment, offset)
int read_port(u16 port, u16 segment, u16 *offset, u16 count)
{ int i;
  for (i=0; i<count; i++)
    put_word(in_word(port), segment, offset++);
}
// write_port() writes count words from (segment, offset) to port
int write_port(u16 port, u16 segment, u16 *offset, u16 count)
{ int i;
  for (i=0; i<count; i++)
    out_word(port, get_word(segment, offset++));
}
int delay(){ }
int hd_busy() {return in_byte(HD_STATUS) & 0x80;} // test BUSY
int hd_ready(){return in_byte(HD_STATUS) & 0x40;} // test READY
int hd_drq()  {return in_byte(HD_STATUS) & 0x08;} // test DRQ
int hd_reset()
{ /****** HD software reset sequence *****/
  ControlRegister(0x3F6)=(0000 1RE0); R=reset, E=0:enable interrupt
  Strobe R bit from HI to LO; with delay time in between:
    Write 0000 1100 to ControlReg; delay();
    Write 0000 1000 to ControlReg; wait for notBUSY & no error
  *****/
  out_byte(0x3F6, 0x0C);    delay();
  out_byte(0x3F6, 0x08);    delay();
  if (hd_busy() || cd_error()) {
    printf("HD reset error\n"); return(BAD);
  }
  return 0;    // return 0 means OK
}
```

```

int hd_error() // test for error
{
    int r;
    if (in_byte(0x1F7)& 0x01){ // status.ERROR bit on
        r = in_byte(0x1F1); // read error register
        printf("HD error=%x\n", r); return r;
    }
    return 0; // return 0 if no error
}

int hd_init()
{
    printf("hd_init\n");
    hd_mutex.value = 1;
    hd_mutex.queue = 0;
    hd_sem.value = hd_sem.queue = 0;
}

int hdhandler()
{
    printf("hd interrupt! ");
    V(&hd_sem); // unblock process
    out_byte(0xA0, 0x20); // send EOI
    out_byte(0x20, 0x20);
}

int set_ide_regs(int rw, int sector, int nsectors)
{
    while(hd_busy() && !hd_ready()); // wait until notBUSY & READY
    printf("write to IDE registers\n");
    out_byte(0x3F6, 0x08); // control = 0x08; interrupt
    out_byte(0x1F2, nsectors); // sector count
    out_byte(0x1F3, sector); // LBA low byte
    out_byte(0x1F4, sector>>8); // LBA mid byte
    out_byte(0x1F5, sector>>16); // LBA high byte
    out_byte(0x1F6, ((sector>>24)&0x0F) | 0xE0); // use LBA for drive 0
    out_byte(0x1F7, rw); // READ | WRITE command
}

int hd_rw(u16 rw, u32 sector, char *buf, u16 nsectors)
{
    int i;
    P(&hd_mutex); // procs execute hd_rw() ONE at a time
    hd_sem.value = hd_sem.queue = 0;
    set_ide_regs(rw, sector, nsector); // set up IDE registers
    // ONE interrupt per sector read|write; transfer data via DATA port
    for (i=0; i<nsectors; i++){ // loop for each sector
        if (rw==HD_READ){
            P(&hd_sem); // wait for interrupt
            if (err = hd_error())
                break;
            read_port(0x1F0, getds(), buf, 256); // 256 2-byte words
            buf += 512;
        }
        else{ // for HD_WRITE, must wait until notBUSY and DRQ=1
            while (hd_busy() && !hd_drq());
            write_port(0x1F0, getds(), buf, 256);
            buf += 512;
            P(&hd_sem); // wait for interrupt
            if (hd_error())
                break;
        }
    }
}

```

```

    }                                // end loop
    V(&hd_mutex);                    // release hd_mutex lock
    if (hd_error()) return BAD;
    return 0;
}

```

10.7.4. Improved HD Driver

The second HD driver is based on 10.7.2.(7b). In this case, the process sets up a data area containing disk I/O parameters, such as R|W operation, starting sector, number of sectors, etc. Then it starts R|W of the first sector and blocks until all the sectors are read or written. On each interrupt, the interrupt handler performs R|W of the remaining sectors. When all sectors R|W are finished, it unblocks the process on the last interrupt.

```

/***** HD driver for synchronous disk I/O *****/
/** HD I/O parameters common to hd_rw() and interrupt handler */
u16 opcode;          // HD_READ | HD_WRITE
char *bufPtr;        // pointer to data buffer
int ICOUNT;          // sector count
u16 herror           // error flag
int hdhandler()      // HD interrupt handler
{
    printf("HD interrupt ICOUNT=%d\n", ICOUNT);
    if (herror = hd_error()) // check for error
        goto out;
    // ONE interrupt per sector read|write; transfer data via DATA port
    if (opcode==HD_READ){
        read_port(0x1F0, getds(), bufPtr, 512);
        bufPtr += 512;
    }
    else{ // HD_WRITE
        if (ICOUNT > 1){
            unlock(); // allow interrupts
            write_port(0x1F0, getds(), bufPtr, 512);
            bufPtr += 512;
        }
    }
    if (--ICOUNT == 0){
        printf("HD inth: V(hd_sem)\n");
        V(&hd_sem);
    }
out:
    if (herror && ICOUNT)
        V(&hd_sem); // must unblock process even if ICOUNT > 0
    out_byte(0xA0, 0x20); // send EOI
}
int hd_rw(u16 rw, u32 sector, char *buf, u16 nsectors)
{
    P(&hd_mutex); // one proc at a time executes hd_rw()
    hd_sem.value = hd_sem.queue = 0;
    printf("hd_rw: setup I/O paremeters for interrupt handler\n");
    opcode = rw; // set opcode
    bufPtr = buf; // pointer to data buffer
    ICOUNT = nsectors; // nsectors to R/W
    herror = 0; // initialize herror to 0
}

```



```

    set_ide_regs(rw, sector, sectors); // set up IDE registers
    if (rw==HD_WRITE){ // must wait until notBUSY and DRQ=1
        while (hd_busy() && !hd_drq());
        write_port(0x1F0, getds(), buf, 512);
        bufPtr += 512;
    }
    P(&hd_sem); // block until r/w of ALL nsectors are done
    V(&hd_mutex); // unlock hd_mutex lock
    return hderror;
}

```

10.7.5. HD Driver with I/O Queue

In an operating system, read operations are usually synchronous, meaning that a process must wait for the read operation to complete. However, write operations are usually asynchronous. In the MTX kernel, all HD write operations are delay writes. After issuing a write request, the process continues without waiting for the write operation to complete. Actual writing to the HD may take place much later by the I/O buffer subsystem. Since the interrupt handler also issues R/W operations, `hd_rw()` cannot contain any sleep or P operations which would block the caller. The algorithm of the MTX HD driver is shown below.

```

----- MTX HD driver: Shared Data Structure -----
    struct request{
        struct request *next; // next request pointer
        int    opcode;        // READ|WRITE
        u32    sector;        // start sector (or block#)
        u16    nsectors;      // number of sectors to R/W
        char   buf[BLKSIZE];  // data area
        SEMAPHORE io_done;    // initial value = 0
    }; I/O_queue = a (FIFO) queue of I/O requests;

----- MTX HD Driver Upper-half Process) -----
hd_read()
{
    1. construct an "I/O request" with SEMAPHORE request.io_done=0;
    2. hd_rw(request);
    3. P(request.io_done); // "wait" for READ completion
    4. read data from request.buf;
}

hd_write()
{
    1. construct an "I/O request";
    2. write data to request.buf;
    3. hd_rw(request); // do not "wait" for completion
}

hd_rw(I/O_request)
{
    1. enter request into (FIFO) I/O_queue;
    2. if (fisrt in I/O_queue)
        start_io(request); // issue actual I/O to HD
}

```

```

----- MTX HD Driver Lower-half -----
InterruptHandler()
{
    request = first request in I/O_queue;
    1. while (request.nsectors){
        transfer data;
        request.nsectors--;
        return;
    }
    2. request = dequeue(I/O_queue); // 1st request from I/O queue
    if (request.opcode==READ)        // READ is synchronous:
        V(request.io_done);          // unblock waiting process
    else
        release the request buffer; // dealy write; release buffer
    3. if (!empty(I/O_queue))        // if I/O_queue not-empty
        start_io(first request in I/O_queue); // start next I/O
}

```

Details of the MTX HD driver will be shown later when we discuss I/O buffer management for block devices in Chapter 12.

10.7.6. MTX10.hd: Demonstration of IDE Driver

MTX10.hd demonstrates the IDE driver. It implements the second HD driver described in Section 10.7.4. After booting up MTX10.hd, enter the hd command from Umode. It issues a syscall(10) to execute the hd() function in kernel. In hd(), it first calls hd_rw() to read the HD's MBR sector to find out the start sectors of the partitions. It writes 10 lines of text to block 0 of partition 1. Then it reads the same block back to verify that the write operations are completed successfully. For each read/write operation, the driver displays messages to show the interactions between the process and the interrupt handler. The reader may modify the hd() code to choose an appropriate free partition to read/write.

10.8. ATAPI Driver

CDROM drives use the ATAPI (AT Attachment Packet Interface) protocol [ATA, 1996], which is an extension of PATA. At the hardware level, ATAPI allows ATAPI devices, e.g. CD/DVD drives, to be connected directly to the IDE bus. All the I/O port addresses are the same as in PATA. However, some of the command registers are redefined in ATAPI. The major difference between PATA and ATAPI is that ATAPI uses SCSI-like packets to communicate with ATAPI devices. A packet consists of 12 bytes, which are written to the data port when the drive is ready to accept the packet. Once a packet is issued, interactions between the host and the device are governed by the ATAPI protocol.

10.8.1. ATAPI Protocol

Figure 10.16 shows the I/O ports of the ATAPI interface.

```
// ATAPI I/O ports: registers 0x172,0x174,0x175 are REDEFINED in ATAPI
#define CD_DATA      0x170    // data port for R/W
#define CD_ERROR     0x171    // error register/write=features register
#define CD_INTREAs  0x172    // Interrupt reason (see below)
#define CD_NO_USE    0x173    // not used by ATAPI
#define CD_LBA_MID   0x174    // byte_count LOW byte
#define CD_LBA_HI    0x175    // byte_count HI byte
#define CD_DRIVE     0x176    // drive select (0x00=master,0x10=slave)
#define CD_CMD       0x177    // command: 0xA0 (packet command)
#define CD_STATUS    0x177    // status register (see below)
```

Figure 10.16. ATAPI I/O Ports

The status (0x177) and error (0x171) registers are the same as in PATA, except that some of the bits, especially those in the error register, are redefined for ATAPI devices.

	7	6	5	4	3	2	1	0
status reg : 0x177 =	BUSY	REDY	FAULT	SEEK	DRQ	CO	-	ERR
error reg : 0x171 =	sense key error MCR ABRT EOM ILI							
sense key error=notReady,hardware,illegal request,unit attention								
MCR=media change,ABRT=abort,EOM=end of media,ILI=illegal length								

Figure 10.17. ATAPI Status and Error Registers

The interrupt reason register (0x172) is new in ATAPI. When an ATAPI interrupt occurs, IO indicates the data direction (0 = toDevice, 1 = toHost) and CoD indicates whether the request is for Command packet (0) or data (1). The meaning of the (IO,CoD,DRQ) bits are shown in Figure 10.18.

Interrupt reason register = 0x172; with DRQ in status register 0x177

7	6	5	4	3	2	1	0	status
reserved		Rel		IO	CoD	DRQ	(IO:0=toDev,1=toHost;CoD:0=data,1=command)	
				—	—	—		
				0	0	1	ready for data from host (PIO write data)	
				0	1	1	ready to accept command packet	
				1	0	1	data to host ready (PIO read data)	
				1	1	0	final status interrupt	

Figure 10.18. ATAPI Interrupt Reason Register

The byte count registers (0x174-175) are used in 2 ways. When issuing a command packet to a device, byte count is the number of bytes needed by the host. After a DRQ interrupt, byte count is the actual number of bytes transferred by the device. A device may use several data transfer operations, each identified by a separate DRQ interrupt, to satisfy the needed byte count. On each DRQ interrupt, the host must examine the interrupt reason. If the interrupt reason is (10), the host must continue to read data until it sees the final status interrupt (11) with DRQ=0.

10.8.2. ATAPI Operation Sequence

The following shows the operation sequence of the ATAPI protocol.

- (1). host : poll for BSY=0 and DRQ=0; then write to feature, byteCount, drive registers (0x171,0x174,0x176); then, write packet command 0xA0 to command register (0x177).
- (2). drive: set BSY, prepares for command packet transfer. Then set (IO,coD) to (01) and assert DRQ, cancel BSY.
- (3). host : when DRQ=1, write 12 command bytes to Data Register 0x170; then wait for drive interrupt;
- (4). drive: clear DRQ, set BSY, read feature & byteCount Regs. For PIO mode, put byte count into byteCount regs, set (IO,coD)=(10), DRQ=1, BSY=0, then generate interrupt INTRQ.
- (5). host : interrupt handler: read status register (to clear INTRQ) to get DRQ. DRQ=1 means data ready for PIO. Read data port by actual byte count.
- (6). drive: After data read by host, clears DRQ=0. If more data, set BSY=1, repeat (4) to (6) until all needed bytes are transferred.
- (7). drive: When all needed data are transferred, issue final status interrupt by BSY=0, (IO,Cod,DRQ)=(110) and INTRQ.
- (8). host : final status interrupt: read status register and error register.

10.8.3. ATAPI Driver Code

```
// ATAPI registers 0x172,0x174,0x175 are REDEFINED in ATAPI
#define CD_DATA      0x170 // data port for R/W
#define CD_ERROR     0x171 // error register/write=features register
#define CD_INTREDA   0x172 // Read:interrupt reason register with DRQ
#define CD_NO_USE    0x173 // not used by ATAPI
#define CD_LBA_MID   0x174 // byte count LOW byte bits0-7
#define CD_LBA_HI    0x175 // byte count HI byte bits 8-15
#define CD_DRIVE     0x176 // drive select
#define CD_CMD       0x177 // command : R=0x20 W=0x30
#define CD_STATUS    0x177 // status register
/***** (1) utility functions *****/
int cd_busy() { return (in_byte(0x177) & 0x80);}
// BUSY and READY are complementary; only need cd_busy()
int cd_ready(){ return (in_byte(0x177) & 0x40); }
int cd_DRQ() { return (in_byte(0x177) & 0x08);}
int cd_error()
{
    int r = in_byte(0x177); // read status REG
    if (r & 0x01){ // if status.ERROR bit on
        r = in_byte(0x171); // read error REG
        printf("CD RD ERROR = %x\n", r);
        return r;
    }
    return 0;
}
/***** (2). Common database of ATAPI driver *****/
int cderror;
char *bufPtr;
u16 byteCount;
u16 procSegment; // calling process Umode segment
/***** (3) ATAPI Driver Upper-half (Process) *****/
int getSector(u32 sector, char *buf, u16 nsector)
{
    int i; u16 *usp; u8 *cp;
    printf("CD/DVD getSector : sector=%l\n", sector);
}
```

```

cderror = 0; // clear error flag
bufPtr = buf; // pointer to data area
procSegment = running->uss; // if buf is in Umode
cd_sem.value = cd_sem.queue = 0; // initialize cd_sem to 0
/***** READ 10 packet layout *****/
      0  1  | 2 3 4 5 | 6 | 7 8 9 | 10 11 |
      .byte 0x28 0 | LBA | 0 | len | 0 0 |
*****/
// create packet: packet.lba[4]; low-byte=MSB hi-byte=LSB
zeropkt();
pkt[0] = 0x28; // READ 10 opcode in pkt[0]
cp = (u8 *)&sector; // sector LBA in pkt[2-5]
pkt[2] = *(cp+3); // low byte = MSB
pkt[3] = *(cp+2); // LBA byte 1
pkt[4] = *(cp+1); // LBA byte 2
pkt[5] = *cp; // high byte= LSB
pkt[7] = 0; // pkt[7-8]=number of sectors to READ
pkt[8] = nsector; // nsector's LSB
out_byte(0x376, 0x08); // nInt=0 => enable drive interrupt
out_byte(0x176, 0x00); // device : 0x00=master;
//(1). poll for notBUSY and then ready:
while (cd_busy() || cd_DRQ()); // wait until notBUSY and DRQ=0
// 2. write to features regs 0x171, byteCount, drive registers
out_byte(0x171, 0); // feature REG = PIO (not DMA)
// write host NEEDED byte_count into byteCount regs 0x174-0x175
out_byte(0x174, (nsector*2048) & 0xFF); // Low byte
out_byte(0x175, (nsector*2048) >> 8); // high byte
// 3. write 0xA0 to command register 0x177
//printf("write 0xA0 to cmd register\n");
out_byte(0x177, 0xA0); // drive starts to process packet
// wait until notBUSY && READY && DRQ is on
while (cd_busy() || !cd_DRQ()); // wait until notBUSY and DRQ=1
//printf("write 6 words of packet to 0x170\n");
usp = &pkt;
for (i=0; i<6; i++)
    out_word(0x170, *usp++);
P(&cd_sem); // process "wait" for FINAL status interrupt
return cderror;
}
/***** (4). ATAPI Driver Lower-half: Interrupt Handler *****/
int cdhandler()
{
    u8 reason, status, err; u16 byteCount;
    //read status, interrupt reason and error registers; get status.DRQ:
    status = in_byte(0x177);
    reason = in_byte(0x172) & 0x03;
    err = in_byte(0x171);
    if (status & 0x01){ // status.ERROR bit on ==> set cderror flag
        cderror = 1;
        V(&cd_sem); // unblock proc
        if (err & 0x08) // err=xxxxMyyy : M=need media change => ignore
            printf("media change error\n");
        //printf("CD error : status=%x err=%x\n", status, err);
        goto out;
    }
    if ((reason==0x01) && (status & 0x08)){ // 011=ready for command
        //printf("CD ready for commands\n");
    }
}

```

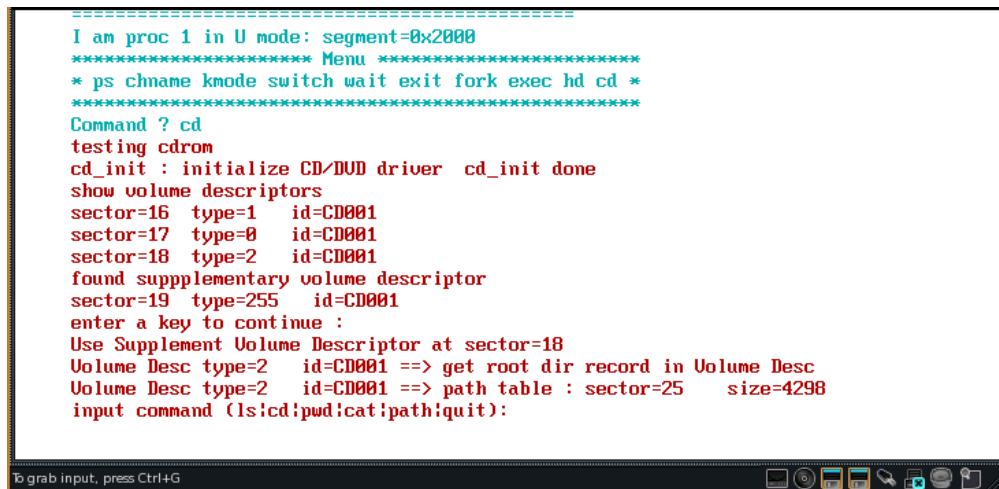
```

        goto out;
    }
    if ((reason==0x02) && (status & 0x08) && !(status & 0x80)){
        //printf("CD data ready for PIO ");
        byteCount=(in_byte(0x175) << 8 ) | (in_byte(0x174));
        //printf("byteCount=%d\n", byteCount);
        /***** PIO data to Kmode or Umode *****/
        read_port(0x170, getds(), bufPtr, byteCount);    // PIO to KMODE
        //read_port(0x170, procSegment, bufPtr, byteCount); // to Umode
        goto out;
    }
    if (reason==0x03 && !(status & 0x08)){ // 110: FINAL status interrupt
        //printf("final status interrupt:status=%x:V(cd_sem)\n", status);
        V(&cd_sem);
    }
out:
    out_byte(0xA0, 0x20);    // EOI to slave 8259 PIC
    out_byte(0x20, 0x20);    // EOI to master PIC
}

```

10.8.4. MTX10.hd.cd: Demonstration of ATAPI Driver

MTX10.hd.cd is a sample MTX system with both IDE and ATAPI drivers. First, run mk to install MTX to a FD image. Then run QEMU or VMware on the FD image. Configure QEMU or VMware with either a real or virtual CDROM containing an iso9660 file system. When MTX boots up, enter the cd command to test the ATAPI driver. The driver testing program, cd.c, supports such operations as ls, cd, pwd and cat, which allow the user to navigate the iso9660 file system tree on the CDROM. Figure 10.19 shows the screen of running MTX10.hd.cd



```

=====
I am proc 1 in U mode: segment=0x2000
***** Menu *****
* ps cname kmode switch wait exit fork exec hd cd *
*****
Command ? cd
testing cdrom
cd_init : initialize CD/DVD driver  cd_init done
show volume descriptors
sector=16 type=1 id=CD001
sector=17 type=0 id=CD001
sector=18 type=2 id=CD001
found supplementary volume descriptor
sector=19 type=255 id=CD001
enter a key to continue :
Use Supplement Volume Descriptor at sector=18
Volume Desc type=2 id=CD001 ==> get root dir record in Volume Desc
Volume Desc type=2 id=CD001 ==> path table : sector=25 size=4298
input command (ls:cd:pwd:cat:path:quit):

```

Figure 10.19. Demonstration of HD and CD drivers

Problems

1. The console display driver only handles the special chars `\n`, `\r` and `\b`. Modify it to handle the tab char, `\t`. Each `\t` should be expanded to, e.g. 8 spaces. Alternatively, each `\t` should advance the Cursor to the beginning of the next display boundary.
2. Modify the console display driver to support scroll-down. Use either a function key or the up-arrow key to initiate the scroll-down action.
3. Virtual Console: virtual consoles, which are often called the poor man's X-windows, provide logical display screens for different processes. In this scheme, each PROC has a 2000 words area for saving the current display screen in the video RAM. During process switch (e.g. by a function key), save current PROC's screen and restore the saved screen of the new process. The visual effect is as if the display is switched from one PROC to another. Implement virtual consoles for processes in MTX.
4. Keyboard driver Programming problems
 - (1). Use Control-C key to send a `SIGINT`(2) signal to all processes associated with the keyboard. Install handler function to ignore the signal or as a software interrupt.
 - (2). Use Control-K as the kill key, which erases the current input line.
 - (3). Catch F2, F3 as hot keys to display sleeping PROCs, semaphore values and queues.
 - (4). Implement `uV(semaphore *s, PROC *p)`, which unblocks a process p from a semaphore queue.
 - (5). Use process status, e.g. `STOPPED`, to implement background processes.
 - (6). Modify the `fg` command to `fg pid`, which promotes a specific process to the foreground.
 - (7). Command history: Save executed commands in a buffer. Use up-down arrow keys to recall a previously executed command, with possible editing, for execution again.
5. Serial Ports Programming Problems

The serial port driver operates in the "half-duplex" mode, in which it does not echo input chars as they are received. In full-duplex mode, the driver echoes each input char, including special char handling. Modify the serial port driver for full-duplex mode operation.
6. FD driver programming:

Extend the FD driver to support two FD drives.
7. The HD driver works for the master drive of IDE0. Modify the HD driver to support two IDE0 drives.
8. The ATAPI driver works for the second IDE master drive. Extend it to support two IDE1 drives.

References

1. ATA: "ATA Interface for CD-ROMs", rev 2.6, SFF Committee, Jan., 1996.
2. ATAPI: "ATA/ATAPI Command Set (ATA8-ACS)", 2006,
<http://www.t13.org/documents/uploadeddocuments/docs2006/d1699r3f-ata8-ac.pdf>
3. FDC: "Floppy Disk Controller", http://wiki.osdev.org/Floppy_Disk_Controller
4. PATA: Parallel ATA: http://en.wikipedia.org/wiki/Parallel_ATA
5. SATA: "Serial ATA: A Comparison with Ultra ATA Technology". Seagate Technology, 2012
6. SVGA: http://en.wikipedia.org/wiki/Super_video_graphics_array
7. T13 AT Attachment (1998). AT Attachment Interface for Disk Drives (ATA-1)
8. VDC: Motorola 6845 VDC: <http://www.tinyvga.com/6845>