19

Exception Handling and Support Routines

This chapter discusses the exception handling routines and the support routines that have been used throughout the system (e.g., *kprintf*). It concentrates on engineering and programming techniques. Although such details may seem trivial, building a reasonable set of support routines early can ease debugging significantly. So, though this chapter has been placed at the end of the book, implementers would do well to establish such software before tackling the system proper.

19.1 Exceptions, Traps, And Illegal Interrupts

Arranging an operating system to correctly associate device addresses, interrupt vector locations, and interrupt dispatchers is an annoying task that plagues most implementers. Often, mismatches between controller switch settings and coding cause devices to interrupt at vector locations other than those expected by the system. A similar problem occurs if a device is added to the bus without changing the operating system to accommodate it.

A related problem occurs when bugs in a program cause it to generate an exception. The hardware handles exceptions – events that occur when a program attempts an illegal operation – like interrupts. An exception might occur, for example, if a program generates an array subscript beyond the end of the array or if it attempts to divide by zero. In particular, the 8088 processor recognizes two run-time exceptions: divide by zero (interrupt type 0) and arithmetic overflow (interrupt type 4). The corresponding vectors are called *exception vectors*. On the 8088, arithmetic overflow is not treated as an exception unless the software performing arithmetic operations deliberately requests overflow exception trapping.

To inform the programmer of exceptions and unexpected interrupts, PC-Xinu can capture them and print a message on the console. It should be explained that early versions consisted of a series of *halt* statements – *panic* was expanded incrementally as the system was developed.

19.2 Initialization Of Interrupt Vectors

Chapter 13 explains how the exception and interrupt vectors are initialized by the code in file *initiali.c* by calling the vector initialization routine *mapinit*. Each exception vector is loaded with the address of procedure *_panic* and a code that indicates the interrupt type. The interrupt type is passed as a parameter (called the ‘‘minor device number’’ in Chapter 9) to *_panic*, so it can determine the source of the exception.

Procedure *_panic* prints information about the machine state, including the contents of registers and the top values on the stack, along with a message explaining the cause of the error. Commonly referred to as a *dump*, the machine state information is generally useless to programmers who work in high-level languages like C. However, it can give programmers trying to create a new operating system important clues about how the hardware is behaving, whether the stack is valid, and why the exception occurred.

To determine the values of machine registers and the stack, the exception handler must be coded using C pointer tricks to extract these values from the run-time stack. When entered, it restores interrupt vectors in preparation for a return to MS-DOS. Then, it proceeds to format and print a message giving a reason for the panic along with the saved values of the registers. If the stack pointer points to a valid memory location, the panic routine also prints the top few locations on the stack on the console. After printing the message, *_panic* halts PC-Xinu.

19.3 Implementation Of Panic

Routines *_panic* and *panic* comprise the panic handler, as shown in file *panic.c*. Although the *_panic* routine may seem long, remember that it must contend with many details pertaining to the run-time stack.

```
/* panic.c - panic, _panic */

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

```
/*------------------------------------------------------------------------
 *  panic  --  print error message and terminate PC-Xinu
 *------------------------------------------------------------------------
 */
int panic(cp)
char *cp;
{
        int i;
        int kprintf();
        int     ps;

        disable(ps);
        maprestore();                   /* restore interrupts to DOS    */
        restore(ps);
        kprintf("\n\n-- panic stop --\n%s\n\n",cp);
        if (numproc==0)
                kprintf("All user processes have completed\n");
        else
                kprintf("PC-Xinu terminated with %d process%s active\n",
                        numproc, ((numproc==1) ? "" : "es"));
        kprintf("Returning . . .\n\n");
        halt();                         /* return to caller             */
}

#define MAXSPRINT       6               /* max stack values to print    */

static int i;                           /* counter for printing stack   */
static char *ep;                        /* error message pointer        */
static struct pentry *pp;               /* pointer to this proc. entry  */
static int sssave, spsave;              /* user stack segment, pointer  */
static int far *sp;                     /* pointer to original stack    */

/*------------------------------------------------------------------------
 *  _panic  --  exception handler, called by exception interrupts
 *------------------------------------------------------------------------
 */
_panic(typ)
int typ;                                /* exception type               */
{
        maprestore();
        switch (typ) {
        case DB0VEC:
                ep = "Divide by zero";
                break;
```

```
        case SSTEPVEC:
                ep = "Single step";
                break;
        case BKPTVEC:
                ep = "Breakpoint";
                break;
        case OFLOWVEC:
                ep = "Arithmetic overflow";
                break;
        default:
                ep = "Unknown interrupt";
        }
        kprintf("\n\n-- panic stop -- \n");
        kprintf("panic: trap type %d (%s) \n",typ,ep);
        pp = &proctab[currpid];
        kprintf("process pid=%d name=%s \n",currpid,pp->pname);
        sys_getstk(&sssave,&spsave);    /* retrieve stack parameters    */
        FP_SEG(sp) = sssave;
        FP_OFF(sp) = spsave;
        sp += 13;                       /* actual top of user stack     */
        kprintf("state: ax=%04x bx=%04x cx=%04x dx=%04x",
                *(sp-6),*(sp-7),*(sp-8),*(sp-9));
        kprintf(" si=%04x di=%04x bp=%04x sp=%04x \n",
                *(sp-10),*(sp-11),*(sp-5),FP_OFF(sp));
        kprintf("       cs=%04x ds=%04x ss=%04x es=%04x",
                *(sp-2),*(sp-12),sssave,*(sp-13));
        kprintf(" ip=%04x fl=%04x \n",*(sp-3),*(sp-1));
        kprintf("stack dump:      \n");
        for ( i=0 ; i<MAXSPRINT ; i++, sp++ )
                kprintf("       sp=%04x *sp=%04x \n",FP_OFF(sp),*sp);
        kprintf("\nPC-Xinu terminated with %d process%s active \n",
                numproc, ((numproc==1) ? "" : "es"));
        kprintf("Returning . . . \n\n");
        halt();                         /* return to caller             */
}
```

Panic is called when an irrecoverable error condition arises (*e.g.*, running out of buffers for message passing). The string passed to *panic* is printed on the CONSOLE before the system is halted. *Panic* behaves almost exactly as *xdone*, except that *panic* does not make a *sleep* call.

Intcom calls *_panic* when an exception or unexpected interrupt occurs. What exactly will the stack contain when this call is made? Since *intcom* saves registers on the stack and saves the stack environment in *sssave:spsave*, *_panic* can print the exact state of the system when the interrupt occurred. The routine *sys_getstk* is used to retrieve the

sssave:spsave values (see Chapter 9). A careful reading of *intcom* will reveal that the stack contains information listed in Figure 19.1,

Stack Offset	Stack Contents
0	es
1	ds
2	di
3	si
4	dx
5	cx
6	bx
7	ax
8	bp
9	ret addr to intmap table, offset 4
10	IRET offset addr
11	IRET segment addr
12	IRET flags
13	user code

Figure 19.1 The stack established by *intcom*

Since exceptions and unexpected interrupts often occur as a result of a serious system failure, the contents of the stack or even system variables (such as the process table) may be suspect. Nevertheless, the stack dump produced by *_panic* should be useful to isolate where in the system the exception occurred.

19.4 Formatted Output

Throughout the system, procedures have called *printf* or *kprintf* to format and write data on the console terminal. Procedure *fprintf*, available from a library, formats and writes a string on a specified file or device. This section shows how these three procedures use a common formatting routine to perform their tasks.

Printf, *fprintf*, and *kprintf* each require as arguments a *format specification string* and a set of variables to be formatted. They convert the variables into a printable representation, according to the format specification, and write the result to a device. At the heart of these routines is a common procedure *doprnt* that actually performs the formatting. We start by looking at it.

```
/* doprnt.c - _doprnt */

#define MAXSTR  80
#define LOCAL   static
typedef unsigned int word;

/*------------------------------------------------------------------------
 *  _doprnt  --  format and write output using 'func' to write characters
 *------------------------------------------------------------------------
 */
_doprnt(fmt, args, func, farg)  /* Adapted by S. Salisbury, Purdue U.   */
char *fmt;                      /* Format string for printf             */
int *args;                      /* Arguments to printf                  */
int (*func)();                  /* Function to put a character          */
int farg;                       /* Argument to func                     */
{
        int     c;
        int     i;
        int     f;              /* The format character (comes after %) */
        char    *str;           /* running pointer in string            */
        char    string[MAXSTR]; /* the string str points to this output */
                                /*  from number conversion              */
        int     length;         /* length of string "str"               */
        char    fill;           /* fill character (' ' or '0')          */
        int     leftjust;       /* 0 = right-justified, else left-just. */
        int     longflag;       /* != 0 for long numerics               */
        int     fmax,fmin;      /* field specifications % MIN . MAX s   */
        int     leading;        /* no. of leading/trailing fill chars.  */
        char    sign;           /* set to '-' for negative decimals     */
        char    digit1;         /* offset to add to first numeric digit */

        for(;;) {
                /* Echo characters until '%' or end of fmt string */
                while( (c = *fmt++) != '%' ) {
                        if( c == '\0' )
                                return;
                        (*func)(farg,c);
                }
                /* Echo "...%%..." as '%' */
                if( *fmt == '%' ) {
                        (*func)(farg,*fmt++);
                        continue;
                }
                /* Check for "%-..." == Left-justified output */
```

```
if (leftjust = ((*fmt=='-') ? 1 : 0) )
        fmt++;
/* Allow for zero-filled numeric outputs ("%0...) */
fill = (*fmt == '0') ? *fmt++ : ' ';
/* Allow for minimum field width specifier for %d,u,x,c,s*/
/* Also allow %* for variable width (%0* as well)     */
fmin = 0;
if( *fmt == '*' ) {
        fmin = *args++;
        ++fmt;
}
else while( '0' <= *fmt && *fmt <= '9' ) {
        fmin = fmin * 10 + *fmt++ - '0';
}
/* Allow for maximum string width for %s */
fmax = 0;
if( *fmt == '.' ) {
        if( *(++fmt) == '*' ) {
                fmax = *args++;
                ++fmt;
        }
        else while( '0' <= *fmt && *fmt <= '9' ) {
                fmax = fmax * 10 + *fmt++ - '0';
        }
}
/* Check for the 'l' option to force long numeric */
if( longflag = ((*fmt == 'l') ? 1 : 0) )
        fmt++;
str = string;
if( (f = *fmt++) == '\0' ) {
        (*func)(farg,'%');
        return;
}
sign = '\0';   /* sign == '-' for negative decimal */

switch( f ) {
case 'c' :
        string[0] = (char) *args;
        string[1] = '\0';
        fmax = 0;
        fill = ' ';
        break;

case 's' :
```

```
        str = (char *) *args;
        fill = ' ';
        break;

case 'D' :
        longflag = 1;
case 'd' :
        if(longflag) {
                if ( *(long  *)args < 0) {
                        sign = '-';
                        *(long *)args = -*(long *)args;
                }
        } else {
                if ( *args < 0 ) {
                        sign = '-';
                        *args = -*args;
                }
        }
        longflag--;
case 'U' :
        longflag++;
case 'u' :
        if ( longflag ) {
                digit1 = 0;
                /* negative longs in unsigned format    */
                /* can't be computed with long division */
                /* convert *args to "positive", digit1  */
                /* = how much to add back afterwards    */
                while(*(long *)args < 0) {
                        *(long *)args -= 1000000000L;
                        ++digit1;
                }
                _prtl10(*(long *)args, str);
                str[0] += digit1;
                ++args;
        } else
                _prt10(*args, str);
        fmax = 0;
        break;

case 'O' :
        longflag ++;
case 'o' :
        if ( longflag ) {
```

```
                        _prtl8(*(long *)args, str);
                        ++args;
                } else
                        _prt8(*args, str);
                fmax = 0;
                break;

        case 'X' :
                longflag++;
        case 'x' :
                if ( longflag ) {
                        _prtl16(*(long *)args, str);
                        ++args;
                } else
                        _prt16(*args, str);
                fmax = 0;
                break;

        default :
                (*func)(farg,f);
                break;
        }
        args++;
        for(length = 0; str[length] != '\0'; length++)
                ;
        if ( fmin > MAXSTR || fmin < 0 )
                fmin = 0;
        if ( fmax > MAXSTR || fmax < 0 )
                fmax = 0;
        leading = 0;
        if ( fmax != 0 || fmin != 0 ) {
                if ( fmax != 0 )
                        if ( length > fmax )
                                length = fmax;
                if ( fmin != 0 )
                        leading = fmin - length;
                if ( sign == '-' )
                        --leading;
        }
        if ( sign == '-' && fill == '0' )
                (*func)(farg, sign);
        if ( leftjust == 0 )
                for( i = 0; i < leading; i++ )
                        (*func)(farg,fill);
```

```
                if ( sign == '-' && fill == ' ' )
                        (*func)(farg, sign);
                for( i = 0 ; i < length ; i++ )
                        (*func)(farg, str[i]);
                if ( leftjust != 0 )
                        for( i = 0; i < leading; i++ )
                                (*func)(farg, fill);
        }
}


LOCAL  _prt10(num, str)
word    num;
char    *str;
{
        int     i;
        char    c, temp[6];

        temp[0] = '\0';
        for( i = 1; i <= 5; i++ ) {
                temp[i] = num % 10 + '0';
                num /= 10;
        }
        for( i = 5; temp[i] == '0'; i-- );
        if ( i == 0 )
                i++;
        while( i >= 0 )
                *str++ = temp[i--];
}

LOCAL  _prtl10(num, str)
long    num;
char    *str;
{
        int     i;
        char    c, temp[11];

        temp[0] = '\0';
        for( i = 1; i <= 10; i++ ) {
                temp[i] = num % 10 + '0';
                num /= 10;
        }
        for( i = 10; temp[i] == '0'; i-- );
        if ( i == 0 )
```

```
                i++;
        while( i >= 0 )
                *str++ = temp[i--];
}

LOCAL  _prt8(num, str)
word   num;
char   *str;
{
        int    i;
        char   c, temp[7];

        temp[0] = '\0';
        for( i = 1; i <= 6; i++ ) {
                temp[i] = (num & 07) + '0';
                num = (num >> 3) & 0037777;
        }
        temp[6] &= '1';
        for( i = 6; temp[i] == '0'; i-- );
        if ( i == 0 )
                i++;
        while( i >= 0 )
                *str++ = temp[i--];
}

LOCAL  _prtl8(num, str)
long    num;
char    *str;
{
        int     i;
        char    c, temp[12];

        temp[0] = '\0';
        for( i = 1; i <= 11; i++ ) {
                temp[i] = (num & 07) + '0';
                num = num >> 3;
        }
        temp[11] &= '3';
        for( i = 11; temp[i] == '0'; i-- );
        if ( i == 0 )
                i++;
        while( i >= 0 )
                *str++ = temp[i--];
}
```

```
LOCAL  _prt16(num, str)
word    num;
char    *str;
{
        int     i;
        char    c, temp[5];

        temp[0] = '\0';
        for( i = 1; i <= 4; i++ ) {
                temp[i] = "0123456789abcdef"[num & 0x0f];
                num = num >> 4;
      }
      for( i = 4; temp[i] =='0'; i-- );
      if ( i == 0 )
                i++;
      while( i >= 0 )
                *str++ = temp[i--];
}


LOCAL  _prtl16(num, str)
long    num;
char    *str;
{
        int     i;
        char    c, temp[9];

        temp[0] = '\0';
        for( i = 1; i <= 8; i++ ) {
                temp[i] = "0123456789abcdef"[num & 0x0f];
                num = num >> 4;
        }
        for( i = 8; temp[i] == '0'; i-- );
        if ( i == 0 )
                i++;
        while( i >= 0 )
                *str++ = temp[i--];
}
```

_Doprnt scans and writes the format string on its output, replacing occurrences of ‘‘%...’’ with a character representation of an argument. For example, it replaces occurrences of %o by the characters that give the octal representation of an argument, %d by characters for the decimal representation, and %x by characters for the hexadecimal

(base 16) representation. Whenever *_doprnt* manufactures a character to be written, whether from copying the format string or from the converted value of an argument, it calls a character-handling procedure to write the character. The flexibility of *_doprnt* arises because the character-handling procedure is an argument that can be different each time *_doprnt* is called. The next section shows some of the ways *_doprnt* can be used.

19.4.1 Printf And Fprintf

The three procedures *printf*, *fprintf*, and *kprintf* all use *_doprnt* to format values for output. The only difference is the character-handling routine they pass to *_doprnt*. *Printf* specifies *putc* as the character-handling routine, and the CONSOLE as the device to which characters should be written. When *_doprnt* invokes *putc*, characters are placed in the tty output buffer using the standard PC-Xinu tty device driver:

```
/* printf.c - printf */

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

/*------------------------------------------------------------------------
 *  printf  --  write formatted output to CONSOLE
 *------------------------------------------------------------------------
 */
printf(fmt,args)
char *fmt;
int args;
{
        int     putc();

        _doprnt(fmt, &args, putc, CONSOLE);
        return(OK);
}
```

Fprintf also uses *putc* as the character-handling procedure, but passes it the device specified by its first argument instead of the CONSOLE device:

```
/* fprintf.c - fprintf */

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

/*------------------------------------------------------------------------
 *  fprintf  --  write formatted output on a specified device
 *------------------------------------------------------------------------
 */
fprintf(dev,fmt,args)
int dev;
char *fmt;
int args;
{
        int     putc();

        _doprnt(fmt, &args, putc, dev);
        return(OK);
}
```

19.4.2 Kprintf

Kprintf does much more than provide a convenient way to format a string: it operates independent of the rest of the system. To understand why such a tool should be built as early as possible, consider how difficult it is to debug an output driver if there is no way to print information. Even after the console I/O system is running, *printf* cannot be used to debug other interrupt handlers, because it uses *putc* which may eventually *wait* for the buffer semaphore. Thus, a special-purpose output routine is needed that will never *wait*; *kprintf* serves that purpose.

To print formatted strings without enabling interrupts, *kprintf* indirectly calls the BIOS *wtty* routine through *kputc*. Recall from Chapter 2 that *kputc* makes an elementary translation of a newline into a return/newline combination. Also, the definition of *kputc* has a dummy parameter designed precisely for use with *_doprnt*. Since *kputc* does not monitor the keyboard, output during *kprintf* cannot be stopped using the Ctrl-S key.

The code for *kprintf* is found in file *kprintf.c*.

```
/* kprintf.c - kprintf */

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

```
/*-------------------------------------------------------------------------
 *  kprintf  --  kernel printf: formatted, unbuffered output to the screen
 *-------------------------------------------------------------------------
 */
int kprintf(fmt, args)          /* Derived by Bob Brown, Purdue U.      */
char *fmt;
{
        void    kputc();
        int     pcx;

        xdisable(pcx);
        _doprnt(fmt, &args, kputc, CONSOLE);
        xrestore(pcx);
        return(OK);
}
```

19.5 The Butler Process

As we have observed on a number of other occasions, interrupt service routines cannot perform activities that can result in suspension of the current process. On the other hand, there are interrupt-driven events that may result in system activities that can cause process suspension. Rather than having the interrupt service routine perform these activities, the ISR must trigger a process to perform these activities in its behalf. Following the strategy outlined for keyboard interrupt processing in Chapter 12, we use message passing to communicate between the interrupt service routine and the server process. The interrupt service routine sends the process a message that the event has occurred, and the server process performs the required activity.

The *butler* process, created at system initialization, is designed to respond to events that are not device-specific. One event triggers system termination; other events result in the display of system activities suitable for debugging purposes. The file *butler.h* defines those messages the butler process will recognize.

```
/* butler.h */

/* butler demon process definitions for housekeeping and other chores   */

#define BTLRNAME        "*BUTLER*"      /* process name                 */
#define BTLRSTK         512             /* butler process stack size    */
#define BTLRPRIO        1000            /* butler priority              */

extern  int     butler();               /* butler process code          */
extern  int     butlerpid;              /* butler process pid for msgs  */

#define MSGKILL         -1              /* kill xinu                    */
#define MSGPSNAP        1               /* print a process snapshot     */
#define MSGTSNAP        2               /* print a tty snapshot         */
#define MSGDSNAP        3               /* print a disk snapshot        */

extern  int     psnap();
extern  int     tsnap();
extern  int     dsnap();
```

The code for the *butler* process is simple. Like the keyboard process, the *butler* is designed as an infinite loop. After receiving a message, the *butler* performs the action determined by the value of the message. The code is in file *butler.c*.

```
/* butler.c - butler */

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

/*------------------------------------------------------------------------
 *  butler  --  general housekeeping process.  Responds to messages and
 *      takes appropriate actions.  Messages recognized are:
 *              MSGKILL         - kill the system
 *              MSGPSNAP        - print a process table snapshot
 *              MSGTSNAP        - print tty structure snapshot
 *              MSGPSNAP        - print disk snapshot
 *------------------------------------------------------------------------
 */

PROCESS butler()
{
        int     pcx;
```

```
        int     msg;

        for (;;) {
                msg = receive();
                xdisable(pcx);
                switch (msg) {          /* wait for & get the message   */
                case MSGKILL:
                        xrestore(pcx);
                        xdone();
                        break;          /* can never get here           */
                case MSGPSNAP:
                        psnap();
                        break;
#ifdef Ntty
                case MSGTSNAP:
                        tsnap();
                        break;
#endif
#ifdef Ndsk
                case MSGDSNAP:
                        dsnap();
                        break;
#endif
                }
                kprintf("Press any key to continue . . .");
                kgetc(0);
                kprintf("\n");
                xrestore(pcx);
        }
}
```

19.5.1 The Ctrl-Break Event

The *Ctrl-Break* interrupt has interrupt type *CBRKINT* defined in *bios.h*. PC-Xinu is designed to terminate itself upon receipt of this interrupt, which is triggered by pressing the Ctrl-Break combination on the keyboard. Recall that this interrupt is vectored to the routine *cbrkint* in *initiali.c*. The code for *cbrkint* is trivial.

```
/* cbrkint.c - cbrkint */

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

/*------------------------------------------------------------------------
 *  cbrkint  --  send a MSGKILL message to the butler
 *------------------------------------------------------------------------
 */

INTPROC cbrkint()
{
        sendf(butlerpid,MSGKILL);
}
```

Recall that the *butler* process translates the *MSGKILL* message sent by *cbrkint* into a call to *xdone*. Note too that *cbrkint* forces the message to be sent using *sendf*.

19.5.2 System Snapshots

For debugging or status reporting purposes, it is desirable to be able to display run-time information about the contents of system tables or information on the status of devices. The information displayed on the screen is called a ‘‘snapshot’’ because it captures the system state at an instant in time. System snapshots are provided in PC-Xinu for the process table and for tty (window) and disk devices. These snapshots are taken by the *butler* upon receipt of appropriate messages sent by the keyboard server process *ttyiproc*. The code is in files *psnap.c*, *tsnap.c*, and *dsnap.c* shown here.

```
/* psnap.c - psnap */

/*------------------------------------------------------------------------
 *  psnap  --  print a snapshot of the process table
 *------------------------------------------------------------------------
 */

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

#define CTRLC   3                       /* ASCII code for control-C     */
```

```
char *sttab[8] = {
        "???",
        "current",
        "free",
        "ready",
        "receiving",
        "sleeping ",
        "suspended",
        "waiting"
};

#define STATE(s) sttab[(s>0&&s<8)?s:0]

static struct psnap {
        int     state;
        int     prio;
        char    name[10];
        char    *pbase;
        int     plen;
        char    *pregs;
        int     ctx[4];
} stab[NPROC];

static struct qent qtab[NQENT];

static int pmark[NPROC];
static char qname[4];

#define next(i) qtab[(i)].qnext
#define prev(i) qtab[(i)].qprev
#define key(i)  qtab[(i)].qkey

int psnap()
{
        int     i,j;
        int     s;
        int     ps;
        register struct pentry *pptr;
        register struct psnap *sptr;
        int     *addr;

        disable(ps);
        /* retrieve a copy of the process table */
        for ( i=0; i<NPROC; i++ ) {
```

```
        pptr = &proctab[i];
        sptr = &stab[i];
        sptr->state = pptr->pstate;
        sptr->prio = pptr->pprio;
        strcpy(sptr->name,pptr->pname);
        sptr->pbase = pptr->pbase;
        sptr->plen = pptr->plen;
        sptr->pregs = pptr->pregs;
        if ( pptr->pstate != PRCURR ) {
                addr = (int *) pptr->pregs;
                for ( j=0 ; j<4 ; j++ )
                        sptr->ctx[j] = *addr++;
        }
}
/* retrieve a copy of the q structure */
for ( i=0; i<NQENT; i++ )
        qtab[i] = q[i];
restore(ps);

/* now print the process table */
pprint();

kprintf("Press any key to continue . . .");
if ( kgetc(0) == CTRLC )
        return;
kprintf("\n");

/* now print the queues */
for ( i=0; i<NPROC; i++ )
        pmark[i] = 0;
/* print the semaphore queues */
i = NPROC;
for ( j=0; j<NSEM; j++ ) {
        qname[0] = 'S';
        qname[1] = '0'+(j/10);
        qname[2] = '0'+(j%10);
        qname[3] = 0;
        qprint(i,qname,PRWAIT,0);
        i += 2;
}
qprint(i,"RDY",PRREADY,1);
i += 2;
qprint(i,"SLP",PRSLEEP,1);
for ( i=0 ; i<NPROC; i++ )
```

```
                if ( pmark[i] == 0 ) {
                    s = stab[i].state;
                    if ( s == PRWAIT || s == PRREADY || s == PRSLEEP )
                        kprintf("?%02d: %s %s\n",i,stab[i].name,STATE(s));
                }
}

static pprint()
{
        int     i,j;
        int     s;
        struct  pentry *pptr;
        struct  psnap *sptr;

        kprintf("\n id state      prio name        sp base  top");
        kprintf("   di   si flag   bp\n");
        for ( i=0; i<NPROC; i++ ) {
                sptr = &stab[i];        /* pointer to process entry     */
                if ( (s=sptr->state) != PRFREE ) {
                        kprintf("%3d %-9s %5d %s",
                                i,STATE(s),
                                sptr->prio,sptr->name);
                        kprintf(" %04x %04x %04x",(int)sptr->pregs,
                                (int)sptr->pbase,
                                sptr->plen+(int)sptr->pbase);
                        if ( s != PRCURR )
                                for (j=0; j<4; j++ )
                                        kprintf(" %04x",sptr->ctx[j]);
                        kprintf("\n");
                }
        }
}

static qprint(i,qnam,state,keyprint)
int i;
char *qnam;
int state;
int keyprint;
{
        int     ni,pi;
        int     s;

        if ( prev(i) != EMPTY )
                kprintf("%s: nonempty qprev header\n",qnam);
```

```
        if ( next(i+1) != EMPTY )
                kprintf("%s: nonempty qnext header\n",qnam);
        for ( ni=next(i), pi=i ; ni < NPROC ; pi=ni, ni=next(ni) ) {
                kprintf("%s: pid=%02d",qnam,ni);
                if ( ni < 0 ) {
                        kprintf(" ?queue index corrupted!\n");
                        break;
                }
                kprintf(" %s",stab[ni].name);
                if ( keyprint )
                        kprintf(" key=%d",key(ni));
                if ( pmark[ni]++ != 0 ) {
                        kprintf(" ?process on another queue!\n");
                        break;
                }
                s = stab[ni].state;
                if ( s != state )
                        kprintf(" ?illegal state (%d=%s)",s,STATE(s));
                if ( prev(ni) != pi )
                        kprintf(" ?invalid qprev index (%d)",prev(ni));
                kprintf("\n");
        }
        if ( prev(ni) != pi ) {
                kprintf("%s: ?invalid qprev index (%d)",qnam,prev(ni));
        }
        if ( ni != i+1 )
                kprintf("%s: ?queue terminates incorrectly (%d)",qnam,ni);
        if ( pi != i )
                kprintf("----\n");
}


/* tsnap.c - tsnap */

/*------------------------------------------------------------------------
 *  tsnap  --  print a summary of tty statistics
 *------------------------------------------------------------------------
 */

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

static struct   tsnap {
```

```
        int     ihead;                  /* input queue head             */
        int     itail;                  /* input queue tail             */
        int     isem;                   /* input semaphore value        */
        int     icnt;                   /* input count                  */
        int     ohead;                  /* output queue head            */
        int     otail;                  /* output queue tail            */
        int     osem;                   /* output semaphore value       */
        int     ocnt;                   /* output count                 */
        int     odsnd;                  /* output defer count           */
        int     ehead;                  /* echo queue head              */
        int     etail;                  /* echo queue tail              */
        int     ecnt;                   /* echo count                   */
        int     wstate;
} ttab[Ntty];

tsnap()
{
        int     i,ps;
        struct  tsnap   *tabp;
        struct  tty     *ttyp;

        kprintf("\ncurrent window = %d\n",winofcur);
        kprintf("%2s %5s %5s %5s %5s  %5s %5s %5s %5s %5s  %5s %5s %5s\n",
                "wn","ihead","itail"," isem"," icnt","ohead","otail",
                " osem"," ocnt","odsnd","ehead","etail"," ecnt");
        disable(ps);
        for (i=0 ; i<Ntty ; i++) {
                ttyp = &tty[i];
                tabp = &ttab[i];
                if ( (tabp->wstate=ttyp->wstate) > 0 ) {
                        tabp->ihead = ttyp->ihead;
                        tabp->itail = ttyp->itail;
                        tabp->isem  = semaph[ttyp->isem].semcnt;
                        tabp->icnt  = ttyp->icnt;
                        tabp->ohead = ttyp->ohead;
                        tabp->otail = ttyp->otail;
                        tabp->osem  = semaph[ttyp->osem].semcnt;
                        tabp->ocnt  = ttyp->ocnt;
                        tabp->odsnd = ttyp->odsend;
                        tabp->ehead = ttyp->ehead;
                        tabp->etail = ttyp->etail;
                        tabp->ecnt  = ttyp->ecnt;
                }
        }
```

```
        restore(ps);
        for (i=0 ; i<Ntty ; i++) {
                tabp = &ttab[i];
                if ( tabp->wstate > 0 )
                kprintf(
                "%2d %5d %5d %5d %5d  %5d %5d %5d %5d %5d  %5d %5d %5d\n",
                        i,tabp->ihead,tabp->itail,tabp->isem,tabp->icnt,
                        tabp->ohead,tabp->otail,tabp->osem,tabp->ocnt,
                        tabp->odsnd,tabp->ehead,tabp->etail,tabp->ecnt);
        }
}


/* dsnap.c - dsnap */

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

/*------------------------------------------------------------------------
 *  dsnap  --  print a snapshot of disk requests
 *------------------------------------------------------------------------
 */
dsnap()
{
        struct  dsblk   *dsptr;
        struct  dreq    *drptr;
        int     ps;
        int     i,j;

        disable(ps);
        for (i=0 ; i<Ndsk ; i++) {
                dsptr = &dstab[i];
                if ( dsptr->dsprocnum < 0 )
                        continue;
                drptr = dsptr->dreqlst;
                kprintf("disk=%d, server pid=%d\n",i,dsptr->dsprocnum);
                if ( (drptr=dsptr->dreqlst) != DRNULL) {
                        j = 0;
                        kprintf("  no.  dba pid  buf op\n");
                        while ( drptr != DRNULL ) {
                                kprintf("  %2d. %4d %3d %04x %2d\n",
                                        j++,drptr->drdba,drptr->drpid,
                                        drptr->drbuff,drptr->drop);
                                drptr = drptr->drnext;
```

```
                        }
                } else
                        kprintf("  <empty request queue>\n");
        }
        restore(ps);
}
```

Each of the above snapshot routines extracts information from the appropriate system structures, stores it in temporary locations, and displays the information on the console. Interrupts are disabled while the data is extracted to avoid a change of state which could occur as the result of an interrupt. Observe that the *butler* is always identified as the current process in the *psnap* routine, because the *butler* is running when the process table is searched.

The keyboard process *ttyiproc* sends a snapshot request to the *butler* upon receipt of the *Ctrl-F1*, *Ctrl-F2*, or *Ctrl-F3* key, triggering a process, tty or disk snapshot, respectively. It is a simple matter to add snapshot functionality or to change the keys which trigger the snapshot activity.

19.6 Summary

Trapping and identifying unexpected interrupts and exceptions such as division by zero are important because they help isolate bugs that would otherwise be difficult to catch. Hence, building error detection routines early is essential, even if the routines are crude.

Another essential debugging tool consists of an output routine that masks system interrupt handling for use in kernel debugging. In our case, the routine is called *kprintf*; it provides a way to print the values of variables and constants without relying on the conventional output device driver. *Kprintf* is especially useful for printing messages from within interrupt routines, because the usual buffered output driver cannot be used there.

Displaying system and device tables at run-time can help in monitoring system performance and in identifying problems. The *ttyiproc* keyboard server process recognizes special keys and sends messages to the *butler* process, which prints system snapshots on the console. The *butler* also handles system termination, a task that must be handled outside of an interrupt service routine.

FOR FURTHER STUDY

Most books consider the issue of protection along with that of exception processing. Habermann [1976] and Calingaert [1982] are examples. Papers by Dennis and Van Horn [1966], Lampson [1969], and Fabry [1974] all consider protection mechanisms.

EXERCISES

19.1 Replace *panic* by a set of routines that merely halts the processor with an error code in register AX. How much space can you save?

19.2 How many locations does *panic* require on the user's stack to display status information about an exception or illegal interrupt?

19.3 Design a mechanism that allows the executing process to catch exceptions.

19.4 Design another interface for *_doprnt* that formats output and places it in a string.

19.5 What do you think is the probability of a ''divide by zero'' exception in a BIOS routine? Can you guarantee that one will never occur?

19.6 Is it possible to identify the process that is current when the process snapshot *psnap* is requested using the *Ctrl-F1* key?