13

System Initialization

Initialization is the last step of the design process. Designers should create a system by thinking about it in the executing state, postponing the details of how to get the system started. Thinking about initialization early has the same bad effect as worrying about optimization early: it tends to impose unnecessary constraints on the design and divert the designer's attention from important issues to trivial ones.

If initialization is the last step of design, why should this chapter take it up? We have chosen to introduce it here to show how the pieces discussed so far fit together before the reader loses sight of the fundamental system components. The most important thing to realize is that many microcomputer applications require no more than what we have at hand – a process manager to support concurrent computation, and the means to transmit information to and from running programs. In fact, several applications have been built on top of the ''minimal'' system we have already put together; the rest is just icing on the (layered) cake. So, it makes sense to take a look at how one might start such a system running. It also makes sense to consider initialization now because subsequent chapters describe pieces of the system that are more or less optional; this chapter will help explain why they can be included or ignored without affecting the lower-layer software. The discussion of initialization begins with a consideration of system termination.

13.1 Starting From Scratch

Everyone who has worked with a computer system knows that errant programs or malfunctions in the hardware lead to catastrophic failures, popularly called *crashes*. A crash occurs when the hardware attempts an invalid operation, caused because code or data in the operating system has been destroyed. Users also know that a crash means the contents of memory have been corrupted or lost, and that it will take considerable time (and perhaps a wizard) to restart the operating system. But users often do not understand or appreciate the restart mechanisms.

How can a machine, devoid of programs, spring into action and begin executing again? It cannot. Somehow a program must be deposited in memory before the machine can start. On the oldest machines, restarting was a painful process because a human operator entered the initial program through switches on the front panel. (Some microcomputer systems still use this method.) Switches were replaced by standard keyboards and later by special terminals that could feed in restart programs from paper tape. Now, large machines have attached micro- or mini-computer systems that load the initial program from tape or disk storage attached to the micro. (The microcomputer itself often has its initial program in Read-Only Memory, so it can restart without help from another machine.)

Using switches, keyboards, paper tape, or a microcomputer to load memory is a slow and tedious process; these techniques are only used to load the smallest possible startup program. Once a program has been loaded, the main CPU can execute the startup program which reads a larger program, usually from a specific location on a specific disk storage device. The CPU then branches to the larger program which reads the entire operating system into memory and branches to its beginning. Programs in the sequence that load ever larger programs are often called *bootstraps*, and the entire process is called *rebooting* the system. The terminology comes from the phrase, ‘‘pulling one’s self up by one’s bootstraps,’’ a seemingly impossible task. Other names for the process are *Initial Program Load* (IPL) and *cold start*.

The work of initialization does not end when the CPU begins executing operating system code. The system must initialize devices and system data structures like the semaphore table. It must also check for, and repair, damage to the linked lists and disk pointers in the file system. Most importantly, it must undergo metamorphosis, changing itself from a single program into an operating system capable of running multiple processes concurrently.

After a brief sketch of how PC-Xinu gets started, this chapter concentrates on explaining exactly what happens after the system begins execution. The main goal is to explain the steps necessary to transform the single, sequential program into an operating system that can support concurrent execution.

13.2 Booting PC-Xinu

The PC provides a development environment (editors, an assembler, a C compiler, a linker, and other tools) for creating and modifying the PC-Xinu source code. Compiling and assembling the source code and user program modules into object and library modules, and linking the modules together results in an executable program. This program is executed just like any other PC program. But once it begins, the entire interaction between PC-Xinu and the system hardware components is through the BIOS.

Before PC-Xinu may be started, the host operating system MS-DOS must be booted following the steps described in the previous section. In the case of the PC, the ROM BIOS contains bootstrap code which is executed when the system is turned on or when the *Ctrl-Alt-Del* keys are pressed. (Note in the latter case that the BIOS keyboard ISR

must be active and interrupts enabled for the keys to be recognized.) The BIOS bootstrap code initializes hardware devices, checks system integrity and memory size, and reads into RAM memory another small bootstrap program from disk. This second bootstrap program reads in the remaining parts of MS-DOS and executes its startup code. PC-Xinu begins life as a program executed from within MS-DOS. In a sense, booting PC-Xinu ‘‘from scratch’’ involves the following steps:

BIOS bootstrap –> disk bootstrap –> MS-DOS –> PC-Xinu

13.3 System Startup

When PC-Xinu first begins, the CPU starts executing C run-time initialization code linked from the standard C library. Basically, it must establish a valid stack, process command-line parameters, and determine the size of available memory. A single program, not an operating system, is running when the CPU calls the C procedure *main* and begins executing it. It is this program that initializes important data structures, devices, semaphores and processes. The code is found in the file *initiali.c*: If there is drama in the system, it lies here, where the transformation from program to system takes place.

```
/* initialize.c - main, sysinit */

#include <dos.h>
#include <conf.h>
#include <kernel.h>
#include <io.h>
#include <proc.h>
#include <sem.h>
#include <mem.h>
#include <q.h>
#include <mark.h>
#include <butler.h>
#include <bios.h>
#include <kbdio.h>

#ifdef  Ntty
#include <tty.h>
int     winofcur;               /* define the current input window      */
struct tty      tty[Ntty];      /* window buffers and mode control      */
#endif

#ifdef  Ndsk
#include <disk.h>
#endif

#ifdef  Nmf
#include <mffile.h>
struct mfblk    mftab[Nmf];
#endif

/* Declarations of major kernel variables */

struct  pentry  proctab[NPROC]; /* process table                        */
int     nextproc;               /* next process slot to use in create   */
struct  sentry  semaph[NSEM];   /* semaphore table                      */
int     nextsem;                /* next semaphore slot to use in screate*/
struct  qent    q[NQENT];       /* q table (see queue.c)                */
int     nextqueue;              /* next slot in q structure to use      */

struct  mblock  memlist;        /* list of free memory blocks           */
char    *end;                   /* beginning of free memory             */
char    *maxaddr;               /* end of free memory                   */

/* PC-specific variables */
```

```
int     nmaps;                  /* no. of active intmap entries         */

/* active system status */

int     numproc;                /* number of live user processes        */
int     currpid;                /* id of currently running process      */

/* real-time clock variables and sleeping process queue pointers        */

long    tod;                    /* time-of-day (tics since startup)     */
int     defclk;                 /* non-zero, then deferring clock count */
int     clkdiff;                /* deferred clock ticks                 */
int     slnempty;               /* FALSE if the sleep queue is empty    */
int     *sltop;                 /* address of key part of top entry in  */
                                /* the sleep queue if slnonempty==TRUE  */
int     clockq;                 /* head of queue of sleeping processes  */
int     preempt;                /* preemption counter.  Current process */
                                /* is preempted when it reaches zero;   */
                                /* set in resched; counts in ticks      */

/* miscellaneous system variables */

int     butlerpid;              /* pid of butler process                */
int     rdyhead,rdytail;        /* head/tail of ready list (q indexes)  */

#define NULLNM  "**NULL**"      /* null process name                    */

/************************************************************************/
/***                             NOTE:                                ***/
/***                                                                  ***/
/***   This is where the system begins after the C environment has    ***/
/***   been established.  Interrupts are initially DISABLED, and      ***/
/***   must eventually be enabled explicitly.  This routine turns     ***/
/***   itself into the null process after initialization.  Because    ***/
/***   the null process must always remain ready to run, it cannot    ***/
/***   execute code that might cause it to be suspended, wait for a   ***/
/***   semaphore, or put to sleep, or exit.  In particular, it must   ***/
/***   not do I/O unless it uses kprintf for console output.          ***/
/***                                                                  ***/
/************************************************************************/

/*------------------------------------------------------------------------
 *  main  --  initialize system and become the null process (id==0)
```

```
 *------------------------------------------------------------------------
 */
main(argc,argv)                         /* babysit CPU when no one home */
int argc;
char *argv[];
{
        int     xmain();                /* user's main procedure        */
        int     butler();               /* BUTLER process               */
        int     ps;                     /* save processor flags         */
        int     pcx;                    /* reschedule flag              */

        while ( kbdgetc() != NOCH )     /* eat remaining kbd chars      */
                ;
        kprintf("Initializing . . .\n");
        xdisable(pcx);
        disable(ps);
        if ( sysinit() == SYSERR ) {    /* initialize all of PC-Xinu    */
                kprintf("PC-Xinu initialization error\n");
                maprestore();
                restore(ps);
                exit(1);
        }
        restore(ps);
        kprintf("\nPC-Xinu Version %s\n", VERSION);
        kprintf("%u real mem\n",(word)maxaddr);
        kprintf("%u base addr\n",(word)end);
        kprintf("%u avail mem\n", maxaddr-end);
        kprintf("\nHit any key to continue . . . ");
        kgetc(0);                       /* wait for keyboard input      */
        scrollup(0,0x184f,0,7);         /* clear the screen             */
        putcsrpos(0,0);                 /* home the cursor              */
        xrestore(pcx);

        /* start the butler process                                     */
        resume( butlerpid=create(butler,BTLRSTK,BTLRPRIO,BTLRNAME,0) );

        /* start up the user process                                    */
        resume( create(xmain,INITSTK,INITPRIO,"xmain",2,argc,argv) );

        while (TRUE)            /* run forever without actually */
                pause();        /*  executing instructions      */
}

/*------------------------------------------------------------------------
```

```
 *  sysinit  --  initialize all PC-Xinu data structures and devices
 *------------------------------------------------------------------------
 */

LOCAL sysinit()
{
        int     i,j;
        struct  pentry  *pptr;
        struct  sentry  *sptr;
        struct  mblock  *mptr;
        char    *malloc();              /* C memory allocator           */
        char    *realloc();
#ifndef TURBOC
        char    *sys_stknpb();
#endif
        int     xdone();                /* terminate xinu               */
        word    sizmem;                 /* memory sizing                */
        char    *namep;                 /* null process name pointer    */
        struct  devsw   *dvptr;         /* pointer to devtab entry      */

        nmaps=0;                        /* no. of active intmap entries */
        numproc = 0;                    /* initialize system variables  */
        nextproc = NPROC-1;
        nextsem = NSEM-1;
        nextqueue = NPROC;              /* q[0..NPROC-1] are processes  */

#ifdef TURBOC
        sizmem = coreleft() - MDOS;
        if ( (end=malloc(sizmem)) == NULL )
                return(SYSERR);
#else
        for (   sizmem = MMAX;
                sizmem >= MMIN && (end=malloc(sizmem)) == NULL;
                sizmem -= MBLK )
                        ;
        if ( sizmem < MMIN )
                return(SYSERR);
        sizmem -= MDOS;                 /* save some for MSDOS          */
        if ( (end=realloc(end,sizmem)) == NULL )
                return(SYSERR);
#endif
        maxaddr = end+sizmem;           /* top of free memory           */
        end = (char *) roundew( (word)end );
        maxaddr = (char *) truncew( (word)maxaddr );
```

```
        memlist.mnext = mptr =          /* initialize free memory list  */
                (struct mblock *) end;
        mptr->mnext = (struct mblock *) NULL;
        mptr->mlen = maxaddr-end;

        for (i=0 ; i<NPROC ; i++)       /* initialize process table     */
                proctab[i].pstate = PRFREE;

        pptr = &proctab[NULLPROC];      /* initialize null process entry*/
        pptr->pstate = PRCURR;
        pptr->pprio = 0;
#ifdef TURBOC
        pptr->pbase = maxaddr;          /* null process pbase stack ptr */
#else
        pptr->pbase = sys_stknpb();     /* null process pbase stack ptr */
#endif
        namep=NULLNM;
        for (j=0; j<PNMLEN; j++)
                pptr->pname[j] = (*namep ? *namep++ : ' ');
        pptr->pname[PNMLEN] = '\0';
        pptr->paddr = main;
        pptr->pargs = 0;
        currpid = NULLPROC;
#ifndef TURBOC
        sys_stkinit();                  /* initialize run-time stacks   */
#endif
        for (i=0 ; i<NSEM ; i++) {      /* initialize semaphores        */
                (sptr = &semaph[i])->sstate = SFREE;
                sptr->sqtail = 1 + (sptr->sqhead = newqueue());
        }

        rdytail = 1 + ( rdyhead=newqueue() ); /* initialize ready list  */

#ifdef  MEMMARK
        _mkinit();                      /* initialize memory marking    */
#else
        pinit();                        /* initialize ports             */
        poolinit();                     /* initialize the buffer pools  */
#endif

        clkinit();                      /* initialize real-time clock   */

#ifdef  Ndsk
        dskdbp= mkpool(DBUFSIZ, NDBUFF); /* initialize disk buffers     */
```

```
        dskrbp= mkpool(DREQSIZ, NDREQ);
#endif

#ifdef  Ntty
        winofcur = 0;                   /* initialize current window    */
#endif

        mapinit(DB0VEC, _panic, DB0VEC);        /* divide by zero       */
        mapinit(SSTEPVEC, _panic, SSTEPVEC);    /* single step          */
        mapinit(BKPTVEC, _panic, BKPTVEC);      /* breakpoint           */
        mapinit(OFLOWVEC, _panic, OFLOWVEC);    /* overflow             */
        mapinit(CBRKVEC, cbrkint, CBRKVEC);     /* ctrl-break           */

#ifdef  NDEVS
        for ( i=0 ; i<NDEVS ; i++ ) {   /* initialize devices           */
                dvptr = &devtab[i];
                if ( dvptr->dvivec && mapinit(dvptr->dvivec,
                                dvptr->dviint, dvptr->dvminor) == SYSERR )
                        return(SYSERR);
                if (dvptr->dvovec && mapinit(dvptr->dvovec,
                                dvptr->dvoint, dvptr->dvminor) == SYSERR )
                        return(SYSERR);
                init(i);                /* initialize the device        */
        }
#endif

        if ( mapinit( CLKVEC | BIOSFLG, clkint, 0 ) == SYSERR )
                return(SYSERR);

        return(OK);
}
```

The procedure *main* itself is exceedingly simple. It disables interrupts and calls procedure *sysinit* to do the initialization. When *sysinit* returns, it has made the running program into process 0, but interrupts remain disabled and no other processes exist. After printing a few introductory messages, *main* enables interrupts, creates the *BUTLER* process, and calls *create* to start a process running the user's main program *xmain*.

Because the process executing *main* has become the null process, it cannot exit, sleep, wait for a semaphore, or suspend itself. If the initialization routine needed to perform any of these actions, it would have created another process to be the null process, but that seems unnecessary. Once initialization is complete and a process has been created to execute the user's program, the null process just falls into an infinite loop, giving *resched* a process to schedule when no user processes are ready to run.

The null process is slightly more sophisticated than it may seem. Conceptually, it executes an infinite loop. Actually, the loop invokes the macro *pause*, which expands to a call to the assembly language routine *sys_wait* defined in *eidi.asm*. *Sys_wait* enables interrupts and executes the special 8088 machine instruction *HLT*, which stops the processor but leaves interrupts enabled. After an interrupt has been processed, the CPU starts executing the code immediately following the *pause* and continues around the loop until it encounters the *pause* again. Pausing the CPU when there are no computations to perform minimizes interference between the CPU and other devices using the system bus, because it prevents the CPU from fetching instructions from memory. On some systems, such minimization is important when devices like disks are performing DMA transfer because it means the transfer will take less time.

13.4 Transforming The Program Into A Process

Procedure *sysinit* performs the tough part of system initialization. It initializes the system data structures like the semaphore table, the process table, and the free memory list. Obtaining a large block of free memory for this list is the most involved part. The standard C procedure *malloc* is called to obtain the largest possible memory block; the size of the allocated memory block is reduced slightly to allow for additional *malloc* allocations used by C library routines to access MS-DOS facilities. The global variables *end* and *maxaddr* are set to the beginning of the allocated block (the end of data space plus the C system stack) and the maximum free space address, respectively. These are rounded up and truncated (to ensure that the addresses lie on byte boundaries that are multiples of four, required by the *getmem* and *freemem* procedures), and the free memory list is initialized.

Finally, *sysinit* sets up the *intmap* table. First it sets up error handlers for divide by zero and overflow conditions, and the *Ctrl-Break* handler for system termination. Then it calls *mapinit* for all those devices defined in *devtab* whose interrupts must be revectored and calls *init* once for each device in the system. Recall that procedure *init*, in turn, calls the actual device initialization routines indirectly through *devtab*.

The most interesting piece of initialization code occurs about half-way through *sysinit* when it fills in the process table entry for process zero. Most of the process table fields, like the process name field, are merely dressing to make debugging easier. The real work is done by only two lines that assign the process state field *PRCURR* and the current process id variable, *currpid*, the index of the null process. Until these two values are in place, rescheduling would be impossible. Once they have been assigned, however, the program becomes a currently running process that *resched* can identify as process 0. All that remains is to initialize the other pieces of the system so that all services are available before *nulluser* starts a process executing the user's program.

Sysinit also determines the base of the null process stack in its process table entry and does additional compiler-specific stack initialization if necessary. Stack routines for PC-Xinu compiled with the Microsoft C compiler are in the file *stack.asm*:

```
; stack.asm - _sys_stknpb, _sys_stkinit

        include dos.asm

        dseg
; MS-DOS C compiler stack limit register
        extrn   STKHQQ:word
        endds

        pseg

        public  _sys_stknpb
        public  _sys_stkinit

;------------------------------------------------------------------------
; _sys_stknpb  --  return null process pbase stack base pointer
;------------------------------------------------------------------------
; char *sys_stknpb()
_sys_stknpb     proc    near
        mov     ax,STKHQQ               ; get lower stack limit
        ret
_sys_stknpb     endp

;------------------------------------------------------------------------
; _sys_stkinit  --  run-time stack initialization
;------------------------------------------------------------------------
; void sys_stkinit()
_sys_stkinit    proc    near
        mov     STKHQQ,0                ; set stack base to zero
        ret
_sys_stkinit    endp

        endps

        end
```

Sys_stknpb sets the null process stack base, and sys_stkinit essentially disables stack overflow checking generated by the C compiler.

Code generated by the Turbo C compiler does not do run-time stack checking, and consequently stack initialization does not need to be done in this case.

13.5 Summary

Initialization is the last step of system design; it should be postponed to avoid changing the design simply to make it easier. We have discussed initialization here because it shows how the components designed so far can form a usable system.

FOR FURTHER STUDY

Many books comment on system startup. Both Habermann [1976] and Calingaert [1982] touch on the subject. One of the few detailed examples of system startup can be found in Madnick and Donovan [1974], which describes the IBM System/360 ‘‘cold start’’ procedure.

EXERCISES

13.1 Is the order of initialization important for the process table, semaphore table, memory free list, devices, and ready list?

13.2 Explain, by tracing through the procedures involved, what would go wrong if *main* did not disable interrupts before calling *sysinit*.

13.3 Create a disk from which PC-Xinu can be loaded and executed using the PC bootstrap loader. Alternatively, write a bootstrap loader that loads and executes PC-Xinu from a PC-Xinu formatted disk.