13 System Initialization

Introduction

- Many microcomputers require no more than what's been discussed in chapters 1 – 12: a process manager to support **concurrent computations**, and the means to transform information to and from **running programs**.
- It makes sense to consider initialization now, because subsequent chapters describe pieces of the system that are **more or less optional**.
- Chapters 1 – 13 describe a so-called **micro-kernel (system kernel)**, upon which basis a more complete and user-friendly operating system can be built. So a micro-kernel represents a "minimal" system – **the rest ist just icing on the (layered) cake**.

13.1 Starting from scratch

- A **crash** occurs when hardware executes an **invalid operation** caused because code or data in the operating system has been destroyed. A crash also means that the **contents of memory** have been **corrupted or lost**!
- How can a machine, devoid of valid 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 computers this happened **by hand**, using switches.
- Later, **standard keyboards** built to that purpose, i.e. special terminals, now micro- and mini-computers, were used to load the **initial program** from tape or disk storage attached to the micro.
- The microcomputer itself has its initial program in **read-only memory** encoded, so it can **restart** without help from other machines.

Starting from scratch

- Once the initial program has been loaded, the CPU can executed the startup program which reads a **larger program** (usually from a special location on a specific disk).
- Then the CPU branches to a larger program which reads the **entire operating system** into memory, and **branches to its beginning**. This process is called **rebooting** the system.
- When the CPU begins executing the operating system code, the system must **initialize devices and system data structures**: process table `proctab`, semaphore table `semaph[]`, the ready list, interrupt dispatch table `intmap[]`, device control blocks, memory, the sleep queue, and so on.
- It must also check for, and repair, **damage** to the linked lists and disk pointers in the system.

13.2 Booting Xinu

XINU is **downloaded** from another machine called its **host**:

- The host computer generates a **break condition** to halt the 11/2 processor by communicating over the console terminal line (described in the last paragraph of §2.1.11 on page 38).
- The 11/2 responds in **Octal Debugging Technique** (ODT) mode: It sends a prompt and recognizes commands to display and change memory locations and registers.
- The host loads an **initial program** in the 11/2 memory starting at location zero, and starts the 11/2 to execute it.

Booting Xinu

- The initial boot program (sent from host to the 11/2) **reads characters**, using polled I/O (also described in §2.1.11) and deposits them in memory starting at the highest location.
- When finished, the host sends a **break**, forcing the 11/2 into ODT mode.
- When ODT responds, the host starts the 11/2 executing the second sent **boot program**.
- The host and the second boot program **communicate**, with the host sending “packets” of bytes (one-at-a-time), and the boot program acknowledging receipt or requesting transmission.
- The host either tells the second bootstrap to branch to the **start of Xinu** or to halt and **wait for ODT commands**.
- In most cases, the host directs the second bootstrap to **branch to Xinu** and the 11/2’s CPU begins executing start (at 01000_8).

13.3 System startup

- The startup program described below creates the **environment** which a C-program expects; then it jumps to the C-procedure `nulluser`. Its start is located at 01000_8 – the first location beyond the interrupt vectors.
- Basically, startup **disables interrupts** and establishes an **empty stack**.
- Since **memory size is unknown** (the stack should start at the high-end of memory and grows downwards), finding the memory size is done by calling procedure `sizmem`.
- To guarantee that `stackptr` SP is valid while `sizmem` runs, startup sets `stackptr` to an **auxiliary stack** called `kernstk` (declared in file `panic.s` in §19), i.e. 300 decimal bytes, set up in the data area, and used when usual stack cannot be used.
- As its last instruction, start jumps to `nulluser`.

System startup

```
/* startup.s - start */

DISABLE     =     340               / PS to disable interrupts

/*--------------------------------------------------------------------------
/*  Xinu system entry point -- first location beyond interrupt vectors
/*--------------------------------------------------------------------------
      .global      start
start:
      mtps  $DISABLE          / Disable interrupts
      mov   $kernstk,sp       / set up temporary stack pointer
      jsr   pc,sizmem         / _maxaddr set to max address
      mov   _maxaddr,sp       / switch stack to high memory
      reset                   / reset bus
      clr   r5                / clear initial r5 for debugging
      clr   r4
      jmp   _nulluser         / Jump to C startup routine
```

13.4 Finding the size of memory

- The highest valid memory address is found by encoding the **highest possible memory address** in `_maxaddr` and then referencing successively smaller addresses until one is found **causing no exception**.
- To detect an invalid address, `sizmem` relies on the hardware – when nonexistent memory is referenced, the resulting error causes the CPU to trap through the **exception vector** at location 4 in memory (called the **bus exception**).
- `sizmem` sets an address in the exception vector so each time it references an illegal address the hardware **branches to location `sistrap`**.
- `sizmem` cannot destroy the **original data** in the bus exception vector, so it saves its contents upon entry, by pushing them onto the stack, and restores them later.
- In the code, the Label `sisloop` marks the point at which `sismem` **tests a memory location**. An exception trap occurs on each reference, causing a branch to `sistrap`, until a valid address has been reached.
- When a reference succeeds, `sismem` saves the address in **global variable** `maxaddr`, making it available to other system routines.

Finding the size of memory

```
/* sizmem.s - sizmem */

MAXADDR     =     157776            / Maximum possible mem. address
DISABLE     =     340               / PS to     disable     interrupts
ENABLE      =     000               / PS to     enable interrupts
EXCPPC      =     4                 / exception vector address for
EXCPPS      =     6                 / "memory out of range" errors

/*--------------------------------------------------------------------------
/*  sizmem  --  size memory, placing highest valid address in _maxaddr
/*--------------------------------------------------------------------------
      .globl      sizmem
sizmem:
      mfps  -(sp)             / save incoming PS
      mtps  $DISABLE          / disable interrupts
      mov   r0,-(sp)          / save registers used
      mov   *$EXCPPC,-(sp)    / save old contents of
      mov   *$EXCPPS,-(sp)    /  exception vector
      mov   $siztrap,*$EXCPPC / set up vector to catch memory
      mov   $DISABLE,*$EXCPPS / exceptions; disable interrupts
      mov   $MAXADDR,r0       / set r0 to highest possible loc
sizloop:
      mov   (r0),(r0)         / reference what r0 points to
      mov   r0,_maxaddr       / no interrupt - memory exists.
      mov   (sp)+,*$EXCPPS    / restore exception vector
      mov   (sp)+,*$EXCPPC
      mov   (sp)+,r0          / restore r0
      mtps  (sp)+             / restore PS
      rts   pc                / return to caller
siztrap:
      add   $4,sp             / pop interrupted PC and PS
      sub   $2,r0             / move to next lower address
      jbr   sizloop           / try again
```

13.5 Initializing system data structures

- `startup.s` and `sizmem.s` do nothing more than **create a valid run-time environment** for C by setting the `stackptr` to the highest valid memory address.
- A single program is running – startup – when the CPU jumps to `nulluser`. This is the program that initializes important datastructures, devices, semaphores, and processes. The code is found in file `initialize.c` below, where the transformation from program to system takes place.
- **If there is drama in the system, it lies here!**

Initializing system data structures

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <sem.h>
#include <sleep.h>
#include <mem.h>
#include <tty.h>
#include <q.h>
#include <io.h>
#include <disk.h>
#include <network.h>

extern  int     main();                 /* address of user's main prog  */

/* 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      */
int     *maxaddr;               /* max memory address (set by sizmem)   */
#ifdef  NDEVS
struct  intmap  intmap[NDEVS];  /* interrupt dispatch table             */
#endif
struct  mblock  memlist;        /* list of free memory blocks           */
#ifdef  Ntty
struct  tty     tty[Ntty];      /* SLU buffers and mode control         */
#endif

/* active system status */

int     numproc;                /* number of live user processes        */
int     currpid;                /* id of currently running process      */
int     reboot = 0;             /* non-zero after first boot            */
```

Initializing system data structures

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

#ifdef  RTCLOCK
int     count6;                 /* counts in 60ths of a second 6-0     */
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 slnempty==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     */
int     clkruns;                /* set TRUE iff clock exists by setclkr */
#else
int     clkruns = FALSE;        /* no clock configured; be sure sleep  */
#endif                          /*   doesn't wait forever              */

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


/***********************************************************************/
/***                            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 polled output.          ***/
/***                                                                 ***/
/***********************************************************************/
```

Initializing system data structures

```
/*------------------------------------------------------------------------
 *  nulluser  -- initialize system and become the null process (id==0)
 *------------------------------------------------------------------------
 */
nulluser()                              /* babysit CPU when no one home */
{
        int     userpid;
        char    ps;

        kprintf("\n\nXinu Version %s", VERSION);
        if (reboot++ < 1)
                kprintf("\n");
        else
                kprintf("   (reboot %d)\n", reboot);
        sysinit();                      /* initialize all of Xinu */
        kprintf("%u real mem\n",(unsigned)maxaddr+(unsigned)sizeof(int));
        kprintf("%u avail mem\n",
                (unsigned)maxaddr-(unsigned)(&end)+(unsigned)sizeof(int));
        kprintf("clock %sabled\n\n", clkruns==1?"en":"dis");
        enable();                       /* enable interrupts */

        /* start a process executing the user's main program */

        resume(
                userpid = create(main,INITSTK,INITPRIO,INITNAME,1,0);
        );
        while (TRUE) {                  /* run forever without actually */
                pause();                /*  executing instructions      */
        }
}
```

Initializing system data structures

```
/*------------------------------------------------------------------------
 *  sysinit  --  initialize all Xinu data structures and devices
 *------------------------------------------------------------------------
 */
LOCAL   sysinit()
{
        int     i, j;
        struct  pentry  *pptr;
        struct  sentry  *sptr;
        struct  mblock  *mptr;

        numproc  = 0;                   /* initialize system variables */
        nextproc = NPROC-1;
        nextsem  = NSEM-1;
        nextqueue= NPROC;               /* q[0..NPROC-1] are processes */

        memlist.mnext = mptr =          /* initialize free memory list */
          (struct mblock *) roundew(&end);
        mptr->mnext = (struct mblock *)NULL;
        mptr->mlen = truncew((unsigned)maxaddr-NULLSTK-(unsigned)&end);

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

        pptr = &proctab[NULLPROC];      /* initialize null process entry */
        pptr->pstate = PRCURR;
        for (j=0; j<7; j++)
                pptr->pname[j] = "prnull"[j];
        pptr->plimit = ( (int)maxaddr ) - NULLSTK;
        pptr->pbase = maxaddr;
        pptr->paddr = nulluser;
        pptr->pargs = 0;
        currpid = NULLPROC;
```

Initializing system data structures

```
        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 */
#endif
#ifdef  RTCLOCK
        clkinit();                      /* initialize r.t.clock */
#endif
#ifdef  Ndsk
        dskdbp= mkpool(DBUFSIZ,NDBUFF); /* initialize disk buffers */
        dskrbp= mkpool(DREQSIZ,NDREQ);
#endif
        for ( i=0 ; i<NDEVS ; i++ )     /* initialize devices */
                init(i);
#ifdef  NNETS                           /* initialize networks */
        netinit();
#endif
        return(OK);
}
```

13.6 nulluser

- `nulluser` is simple: It calls `sysinit` to do the **initialization**. When `sysinit` returns, the running program has been made into **process 0**, with interrupts disabled, and no other process existing.
- After printing a few messages, `nulluser` **enables interrupts** and calls `create` to start a process running the **user's main program**.
- Because the process executing `nulluser` has become the null-process, it **cannot** exit, sleep, wait for a semaphore, or suspend itself.
- After initialization is completed, and a process has been created to execute the user's main program, the `nulluser` process just becomes an **infinite loop**, capable of reacting to interrupts and giving `resched` a process to schedule to when no user process are ready to run.
- `nulluser` requires a **special print procedure** `kprintf`, using polled I/O, because normal I/O processing would put it in the process state `PRWAIT`, and this is forbidden for the nulluser, because its process state should be *current* or *ready* (see above). `kprintf` is dicussed in §19.4.2.

13.7 Transforming the program into a process

Where is the **program** turned into a **process**?

- Procedure `sysinit` performs the **system initialization**:
  - It initializes the system's data structures, like the semaphore table, q-structure, process table, free memory list etc.
  - It initializes clock routines by calling `clockinit`.
  - Finally, it calls `init` once for each system device.
  - Procedure `init` calls the device initialization routines indirectly through `devtab`.
- Most interesting: it fills in the **process-table fields for process zero**. The real work is done by two lines: assigning the process-table fields `PRCURR`, and `currpid = NULLPROC`!
- Only after these values have been assigned, **rescheduling becomes possible**: then the program becomes a currently running process that `resched` can identify as process 0.

13.8 The map of low core

- The interrupt vectors are all initialized to a **default value** before interrupt vectors for specific devices are filled in.
- Unexpected interrupts trap to procedure `panic`, which **prints a message** on the console and **halts** the 11/2.
- None of the initialization code we have seen until now fills in exception or interrupt vector locations other than the clock vector and the I/O vectors specified in the device switch table.
- **How are these locations initialized**? The remaining vector addresses are initialized statically, before execution begins, to an appropriate entry point int `panic`. File `lowcore.s` contains the code.

The map of low core

```
/* lowcore.s - (map of interupt vectors in low part of memory) */

DISABLE     =     340                / PS to     disable     interrupts
ENABLE      =     000                / PS to     enable interrupts

/*-------------------------------------------------------------------------
/*  absolute location 0  -- fixed interrupt and exception vectors
/*-------------------------------------------------------------------------
.     =     000^.              / panic because something jumped
      jmp   panic0             /          to location zero
.     =     004^.              / bus error (e.g., malfunction)
      panic;      DISABLE+0
.     =     010^.              / illegal/reserved instruction
      panic;      DISABLE+1
.     =     014^.              / BPT instruction and T bit
      panic;      DISABLE+2
.     =     020^.              / Input/Output trap
      panic;      DISABLE+3
.     =     024^.              / Power fail
      panic;      DISABLE+4
.     =     030^.              / EMT instruction, emulator trap
      panic;      DISABLE+5
.     =     034^.              / TRAP instruction
      panic;      DISABLE+6
.     =     040^.
      panic;      DISABLE+7
.     =     044^.
      panic;      DISABLE+7

(repeat for all locations through 774)

.     =     770^.
      panic;      DISABLE+7
.     =     774^.
      panic;      DISABLE+7
```