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 so-called micro-kernel (system kernel).

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 means the contents of memory have been corrupted or lost

How can a machine, devoid of valid programs, spring into action and begin executing ?

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 are 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 restarts without help from other machines.

- Once the initial program has been loaded, the CPU can executed the startup program witch 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[]*, ready list, interrupt dispatch table *intmap[]*, device control blocks, memory.
- It must also check for and repair, damage to the linked lists and disk ptrs 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 11/2 processor
- 11/2 responds in *Octal Debugging Technique* (ODT) mode:
  - it sends a prompt and recognizes commands to display and change memory locations and registers
- Host loads initial program in 11/2 memory starting at location zero, and starts 11/2 executing it.
- Initial boot program (sent from host to 11/2) reads characters, using polled I/O and deposits them in memory starting at highest location
- When finished, host sends a break, forcing 11/2 into ODT mode
- When ODT responds, host starts executing on 11/2 sent boot program
- Host and second boot program communicate, with the host sending "packets" of bytes (once-at-a-time), and boot program acknowledging receipt or requesting transmission.
- Host (either) tells second bootstrap to branch to the start of Xinu (or to halt and wait for ODT commands)
- Host directs second bootstrap to branch to Xinu and CPU begins executing start (at 01000 octal).

14 System startup

- The startup program creates the environment which a C-program expects; then it jumps to the C-procedure *nulluser*
- Start is located at 01000 octal -the first location beyond the interrupt vectors.
- start disables interrupts and establishes a void stack.
- Since memory size is unknown (the stack should start at the high-end of memory and grows downward) finding the memory size is done by calling procedure at *sizmem*
- To guarantee that *stackptr* SP is valid while *sizmem* runs, startup sets *stackptr* to *kernstk*, 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*

See code below !

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

DISABLE = 340                   / PS to disable interrupts

/*------------------------------------------------
/* Xinu system entry point -- first location beyond
/* interrupt vectors
/*------------------------------------------------
        .globl  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
```

14.1 Finding the size of memory

The highest valid memory address is found by:

encoding the highest possible mem. address in *_maxaddr* and then referencing successively smaller addresses until one is found causing no exception

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

MAXADDR =       157776      / Maximum possible me. 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 --  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  / exception; 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
```

14.2 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 - start up - when the CPU jumps to *nulluser*

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

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

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

/* Declarations of major kernel variables */

struct  pentry  proctab[NPROC]; /* process table             */
                        /* next process slot to use in create */
int     nextproc;
struct  sentry  semaph[NSEM];   /* semaphore table           */
                      /* next semaphore slot to use in screate*/
int     nextsem;
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 ptocesses   */
int     currpid;          /* if of currently running process */
int     reboot = 0;       /* non-zero after first boot       */

/* real-time clock variables and sleeping processes
 * 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 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      */
int     clkruns;      /* set TRUE iff clock exists by setclkr */
#else                 /* no clock configured; be sure sleep   */
                      /* doesn't wait forever                 */
int     clkruns = FALSE;
#endif
                      /* head/tail of ready list (q indexes)  */
int     rdyhead,rdytail;
```

```
/************************************************************/
/**                        Note:                           **/
/** This is where the system begins after the C enviroment  **/
/** has been establisched. 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.                              **/
/**                                                         **/
/************************************************************/


/*------------------------------------------------------------
 * nulluser -- initialize system and becom the null process
 * (id = 0)
 *------------------------------------------------------------
 */
```

```
nulluser()                  /* babysit CPU when no one home  */
{
        char    ps;
        kprintf("\n\nXinu Version %s", VESION);
        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));
        enable();            /* enable interrupts             */

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

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

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

        numproc = 0;            /* initialize system variables   */
        nextproc = NPROC-1;
        nextsem = NSEM-1;
        nextqueue = NPROC;      /* q[0..NPROC-1] are processes   */
                                /* initialize free memory list   */
        memlist.mnext = mptr = (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 = PFREE;

                                /* initialize null process entry */
```

```
        pptr = &proctab[NULLPROC];
        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;

        for (i=0 ; i<NSEM ; i++) {  /* initialize semaphores */
                (sptr = &semaph[i])->sstate = SFREE;
                sptr->sqtail = 1 + (sptr->sqhead = newqueue());
        }
                                /* initialize ready list        */
        rdytail = 1 + (rdyhead=newqueue());

#ifdef  MEMMARK
        _mkinit();              /* initialize memory marking    */
#endif
#ifdef  RTCLOCK
        clkinit();              /* initialize r-t clock         */
#endif
#ifdef  Ndsk                    /* initialize disk buffers      */
        dskdbp= mkpool(DBUFSIZE,NDBUFF);
```

```
        dskrbp= mkpool(DREQSIZ,NDREQ);
#endif
        for (i =0 ; i<NDEVS ; i++) /* initialize devices    */
                init(i);
#ifdef  NNETS
        netinit();          /* initialize networks          */
#endif
        return(OK);
}
```

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*, to waits etc., using polled I/O

14.3 Transforming the program into a process

Where is the program turned into a process ?

- Procedure *sysinit* performs the system initialization.
  - It initialize the system's data structures, like the semaphore table, q-structure, process table, free memory list etc.
  - It initialize 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: filling in the process-table fields for process zero.
  - The real work is done by tow lines:
    assigning the process-table fields *PRCURR*, and
    `currpid = NULLPROC`
- Only after these values are assigned, rescheduling becomes possible: the program becomes a currently running process that *resched* can identify as process 0.

14.4 The map of low core

```
/* lowcore.s - (map of interrupt vectors in low part of mem.)*/

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

/*-------------------------------------------------------
/* absolute location 0 -- fixed interrupt and exception
/* vectors
/*-------------------------------------------------------

.       =       ooo^.           / panic because something jumped
        jmp     panic0          / to location zero
.       =       004^.           / bus error (e.g., malfunction)
        panic;  DISABLE+0
.       =       010^.           / illegal/revesed 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
      (repeated for all locations throug 774)
```