```
/* This file contains the main program of MINIX as well as its shutdown code.
 * The routine main() initializes the system and starts the ball rolling by
 * setting up the process table, interrupt vectors, and scheduling each task
 * to run to initialize itself.
 * The routine shutdown() does the opposite and brings down MINIX.
 *
 * The entries into this file are:
 *   main:              MINIX main program
 *   prepare_shutdown:  prepare to take MINIX down
 */
#include "kernel.h"
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <a.out.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/endpoint.h>
#include "proc.h"

/* Prototype declarations for PRIVATE functions. */
FORWARD _PROTOTYPE( void announce, (void));
FORWARD _PROTOTYPE( void shutdown, (timer_t *));

/*===========================================================================*
 *                                main                                       *
 *===========================================================================*/
PUBLIC void main()
{
/* Start the ball rolling. */
  struct boot_image *ip;        /* boot image pointer */
  register struct proc *rp;     /* process pointer */
  register struct priv *sp;     /* privilege structure pointer */
  register int i, s;
  int hdrindex;                 /* index to array of a.out headers */
  phys_clicks text_base;
  vir_clicks text_clicks, data_clicks, st_clicks;
  reg_t ktsb;                   /* kernel task stack base */
  struct exec e_hdr;            /* for a copy of an a.out header */

  /* Clear the process table. Anounce each slot as empty and set up mappings
   * for proc_addr() and proc_nr() macros. Do the same for the table with
   * privilege structures for the system processes.
   */
  for (rp = BEG_PROC_ADDR, i = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++i) {
        rp->p_rts_flags = SLOT_FREE;            /* initialize free slot */
        rp->p_nr = i;                           /* proc number from ptr */
        rp->p_endpoint = _ENDPOINT(0, rp->p_nr); /* generation no. 0 */
        (pproc_addr + NR_TASKS)[i] = rp;        /* proc ptr from number */
        memset(rp->p_mess_sent, 0, sizeof(rp->p_mess_sent)); /* sent message counter */
  }
  for (sp = BEG_PRIV_ADDR, i = 0; sp < END_PRIV_ADDR; ++sp, ++i) {
        sp->s_proc_nr = NONE;                   /* initialize as free */
        sp->s_id = i;                           /* priv structure index */
        ppriv_addr[i] = sp;                     /* priv ptr from number */
  }

  /* Set up proc table entries for processes in boot image.  The stacks of the
   * kernel tasks are initialized to an array in data space.  The stacks
   * of the servers have been added to the data segment by the monitor, so
   * the stack pointer is set to the end of the data segment.  All the
   * processes are in low memory on the 8086.  On the 386 only the kernel
   * is in low memory, the rest is loaded in extended memory.
   */

```

```
 66     /* Task stacks. */
 67     ktsb = (reg_t) t_stack;
 68
 69     for (i=0; i < NR_BOOT_PROCS; ++i) {
 70           int ci;
 71           bitchunk_t fv;
 72           ip = &image[i];                          /* process' attributes */
 73           rp = proc_addr(ip->proc_nr);             /* get process pointer */
 74           ip->endpoint = rp->p_endpoint;           /* ipc endpoint */
 75           rp->p_max_priority = ip->priority;       /* max scheduling priority */
 76           rp->p_priority = ip->priority;           /* current priority */
 77           rp->p_quantum_size = ip->quantum;        /* quantum size in ticks */
 78           rp->p_ticks_left = ip->quantum;          /* current credit */
 79           strncpy(rp->p_name, ip->proc_name, P_NAME_LEN); /* set process name */
 80           (void) get_priv(rp, (ip->flags & SYS_PROC));    /* assign structure */
 81           priv(rp)->s_flags = ip->flags;                  /* process flags */
 82           priv(rp)->s_trap_mask = ip->trap_mask;          /* allowed traps */
 83
 84           /* Initialize call mask bitmap from unordered set.
 85            * A single SYS_ALL_CALLS is a special case - it
 86            * means all calls are allowed.
 87            */
 88           if(ip->nr_k_calls == 1 && ip->k_calls[0] == SYS_ALL_CALLS)
 89                   fv = ~0;                /* fill call mask */
 90           else
 91                   fv = 0;                 /* clear call mask */
 92
 93           for(ci = 0; ci < CALL_MASK_SIZE; ci++)  /* fill or clear call mask */
 94                   priv(rp)->s_k_call_mask[ci] = fv;
 95           if(!fv)                  /* not all full? enter calls bit by bit */
 96                   for(ci = 0; ci < ip->nr_k_calls; ci++)
 97                           SET_BIT(priv(rp)->s_k_call_mask,
 98                                   ip->k_calls[ci]-KERNEL_CALL);
 99
100           priv(rp)->s_ipc_to.chunk[0] = ip->ipc_to;       /* restrict targets */
101           if (iskerneln(proc_nr(rp))) {           /* part of the kernel? */
102                   if (ip->stksize > 0) {          /* HARDWARE stack size is 0 */
103                           rp->p_priv->s_stack_guard = (reg_t *) ktsb;
104                           *rp->p_priv->s_stack_guard = STACK_GUARD;
105                   }
106                   ktsb += ip->stksize;    /* point to high end of stack */
107                   rp->p_reg.sp = ktsb;    /* this task's initial stack ptr */
108                   hdrindex = 0;           /* all use the first a.out header */
109           } else {
110                   hdrindex = 1 + i-NR_TASKS;      /* servers, drivers, INIT */
111           }
112
113           /* The bootstrap loader created an array of the a.out headers at
114            * absolute address 'aout'. Get one element to e_hdr.
115            */
116           phys_copy(aout + hdrindex * A_MINHDR, vir2phys(&e_hdr),
117                                                   (phys_bytes) A_MINHDR);
118           /* Convert addresses to clicks and build process memory map */
119           text_base = e_hdr.a_syms >> CLICK_SHIFT;
120           text_clicks = (e_hdr.a_text + CLICK_SIZE-1) >> CLICK_SHIFT;
121           data_clicks = (e_hdr.a_data+e_hdr.a_bss + CLICK_SIZE-1) >> CLICK_SHIFT;
122           st_clicks= (e_hdr.a_total + CLICK_SIZE-1) >> CLICK_SHIFT;
123           if (!(e_hdr.a_flags & A_SEP))
124           {
125                   data_clicks= (e_hdr.a_text+e_hdr.a_data+e_hdr.a_bss +
126                           CLICK_SIZE-1) >> CLICK_SHIFT;
127                   text_clicks = 0;           /* common I&D */
128           }
129           rp->p_memmap[T].mem_phys = text_base;
130           rp->p_memmap[T].mem_len  = text_clicks;
```

```
131           rp->p_memmap[D].mem_phys = text_base + text_clicks;
132           rp->p_memmap[D].mem_len  = data_clicks;
133           rp->p_memmap[S].mem_phys = text_base + text_clicks + st_clicks;
134           rp->p_memmap[S].mem_vir  = st_clicks;
135           rp->p_memmap[S].mem_len  = 0;
136
137           /* Set initial register values.  The processor status word for tasks
138            * is different from that of other processes because tasks can
139            * access I/O; this is not allowed to less-privileged processes
140            */
141           rp->p_reg.pc = (reg_t) ip->initial_pc;
142           rp->p_reg.psw = (iskernelp(rp)) ? INIT_TASK_PSW : INIT_PSW;
143
144           /* Initialize the server stack pointer. Take it down one word
145            * to give crtso.s something to use as "argc".
146            */
147           if (isusern(proc_nr(rp))) {             /* user-space process? */
148                   rp->p_reg.sp = (rp->p_memmap[S].mem_vir +
149                                   rp->p_memmap[S].mem_len) << CLICK_SHIFT;
150                   rp->p_reg.sp -= sizeof(reg_t);
151           }
152
153           /* Set ready. The HARDWARE task is never ready. */
154           if (rp->p_nr == HARDWARE) RTS_LOCK_SET(rp, NO_PRIORITY);
155           RTS_LOCK_UNSET(rp, SLOT_FREE); /* remove SLOT_FREE and schedule */
156
157           /* Code and data segments must be allocated in protected mode. */
158           alloc_segments(rp);
159     }
160
161   #if SPROFILE
162     sprofiling = 0;       /* we're not profiling until instructed to */
163   #endif /* SPROFILE */
164   #if CPROFILE
165     cprof_procs_no = 0;  /* init nr of hash table slots used */
166   #endif /* CPROFILE */
167
168     /* MINIX is now ready. All boot image processes are on the ready queue.
169      * Return to the assembly code to start running the current process.
170      */
171     bill_ptr = proc_addr(IDLE);           /* it has to point somewhere */
172     announce();                           /* print MINIX startup banner */
173     restart();
174   }
175
176   /*===========================================================================*
177    *                                announce                                   *
178    *===========================================================================*/
179   PRIVATE void announce(void)
180   {
181     /* Display the MINIX startup banner. */
182     kprintf("\nMINIX %s.%s. "
183   #ifdef _SVN_REVISION
184           "(" _SVN_REVISION ")\n"
185   #endif
186         "Copyright 2006, Vrije Universiteit, Amsterdam, The Netherlands\n",
187         OS_RELEASE, OS_VERSION);
188   }
189
190   /*===========================================================================*
191    *                                prepare_shutdown                           *
192    *===========================================================================*/
193   PUBLIC void prepare_shutdown(how)
194   int how;
195   {
```

```
196   /* This function prepares to shutdown MINIX. */
197     static timer_t shutdown_timer;
198     register struct proc *rp;
199     message m;
200
201     /* Send a signal to all system processes that are still alive to inform
202      * them that the MINIX kernel is shutting down. A proper shutdown sequence
203      * should be implemented by a user-space server. This mechanism is useful
204      * as a backup in case of system panics, so that system processes can still
205      * run their shutdown code, e.g, to synchronize the FS or to let the TTY
206      * switch to the first console.
207      */
208   #if DEAD_CODE
209     kprintf("Sending SIGKSTOP to system processes ...\n");
210     for (rp=BEG_PROC_ADDR; rp<END_PROC_ADDR; rp++) {
211         if (!isemptyp(rp) && (priv(rp)->s_flags & SYS_PROC) && !iskernelp(rp))
212             send_sig(proc_nr(rp), SIGKSTOP);
213     }
214   #endif
215
216     /* Continue after 1 second, to give processes a chance to get scheduled to
217      * do shutdown work.  Set a watchog timer to call shutdown(). The timer
218      * argument passes the shutdown status.
219      */
220     kprintf("MINIX will now be shut down ...\n");
221     tmr_arg(&shutdown_timer)->ta_int = how;
222     set_timer(&shutdown_timer, get_uptime() + HZ, shutdown);
223   }
224
225   /*===========================================================================*
226    *                                shutdown                                   *
227    *===========================================================================*/
228   PRIVATE void shutdown(tp)
229   timer_t *tp;
230   {
231   /* This function is called from prepare_shutdown or stop_sequence to bring
232    * down MINIX. How to shutdown is in the argument: RBT_HALT (return to the
233    * monitor), RBT_MONITOR (execute given code), RBT_RESET (hard reset).
234    */
235     intr_init(INTS_ORIG);
236     clock_stop();
237     arch_shutdown(tmr_arg(tp)->ta_int);
238   }
239
240
```