

```

/* 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
 *
 * Changes:
 * Nov 24, 2004  simplified main() with system image (Jorrit N. Herder)
 * Aug 20, 2004  new prepare_shutdown() and shutdown() (Jorrit N. Herder)
 */

#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 *tp));

/*=====
 *          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;
    reg_t ktsb;              /* kernel task stack base */
    struct exec e_hdr;       /* for a copy of an a.out header */

/* Initialize the interrupt controller. */
    intr_init(1);

/* 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 */
    }
}

```

```

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 */
}

/* CODIGO PRACTICA 3 */
for (i=0;i<(NR_TASKS+NR_PROCS);i++)
    messg_sn[i] = 0;

/* 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.
*/

/* Task stacks.*/
ktsb = (reg_t) t_stack;

for (i=0; i < NR_BOOT_PROCS; ++i) {
    ip = &image[i];               /* process' attributes */
    rp = proc_addr(ip->proc_nr);   /* get process pointer */
    ip->endpoint = rp->p_endpoint; /* ipc endpoint */
    rp->p_max_priority = ip->priority; /* max scheduling priority */
    rp->p_priority = ip->priority; /* current priority */
    rp->p_quantum_size = ip->quantum; /* quantum size in ticks */
    rp->p_ticks_left = ip->quantum; /* current credit */
    strncpy(rp->p_name, ip->proc_name, P_NAME_LEN); /* set process name */
    (void) get_priv(rp, (ip->flags & SYS_PROC)); /* assign structure */
    priv(rp)->s_flags = ip->flags; /* process flags */
    priv(rp)->s_trap_mask = ip->trap_mask; /* allowed traps */
    priv(rp)->s_call_mask = ip->call_mask; /* kernel call mask */
    priv(rp)->s_ipc_to.chunk[0] = ip->ipc_to; /* restrict targets */
    if (iskerneln(proc_nr(rp))) { /* part of the kernel? */
        if (ip->stksize > 0) { /* HARDWARE stack size is 0 */
            rp->p_priv->s_stack_guard = (reg_t *) ktsb;
            *rp->p_priv->s_stack_guard = STACK_GUARD;
        }
        ktsb += ip->stksize; /* point to high end of stack */
        rp->p_reg.sp = ktsb; /* this task's initial stack ptr */
        text_base = kinfo.code_base >> CLICK_SHIFT;
            /* processes that are in the kernel */
        hdrindex = 0; /* all use the first a.out header */
    } else {
        hdrindex = 1 + i - NR_TASKS; /* servers, drivers, INIT */
    }
}

/* The bootstrap loader created an array of the a.out headers at
* absolute address 'aout'. Get one element to e_hdr.
*/
phys_copy(aout + hdrindex * A_MINHDR, vir2phys(&e_hdr),
          (phys_bytes) A_MINHDR);

/* Convert addresses to clicks and build process memory map */
text_base = e_hdr.a_syms >> CLICK_SHIFT;
text_clicks = (e_hdr.a_text + CLICK_SIZE - 1) >> CLICK_SHIFT;

```

```

if (!(e_hdr.a_flags & A_SEP)) text_clicks = 0;      /* common I&D */
data_clicks = (e_hdr.a_total + CLICK_SIZE-1) >> CLICK_SHIFT;
rp->p_memmap[T].mem_phys = text_base;
rp->p_memmap[T].mem_len  = text_clicks;
rp->p_memmap[D].mem_phys = text_base + text_clicks;
rp->p_memmap[D].mem_len  = data_clicks;
rp->p_memmap[S].mem_phys = text_base + text_clicks + data_clicks;
rp->p_memmap[S].mem_vir  = data_clicks; /* empty - stack is in data */

```

```

/* Set initial register values. The processor status word for tasks
 * is different from that of other processes because tasks can
 * access I/O; this is not allowed to less-privileged processes
 */

```

```

rp->p_reg.pc = (reg_t) ip->initial_pc;
rp->p_reg.psw = (iskernelp(rp)) ? INIT_TASK_PSW : INIT_PSW;

```

```

/* Initialize the server stack pointer. Take it down one word
 * to give crtso.s something to use as "argc".
 */

```

```

if (isusern(proc_nr(rp))) {      /* user-space process? */
    rp->p_reg.sp = (rp->p_memmap[S].mem_vir +
                  rp->p_memmap[S].mem_len) << CLICK_SHIFT;
    rp->p_reg.sp -= sizeof(reg_t);
}

```

```

/* Set ready. The HARDWARE task is never ready. */

```

```

if (rp->p_nr != HARDWARE) {
    rp->p_rts_flags = 0;          /* runnable if no flags */
    lock_enqueue(rp);           /* add to scheduling queues */
} else {
    rp->p_rts_flags = NO_MAP;    /* prevent from running */
}

```

```

/* Code and data segments must be allocated in protected mode. */

```

```

alloc_segments(rp);
}

```

```

#if ENABLE_BOOTDEV

```

```

/* Expect an image of the boot device to be loaded into memory as well.
 * The boot device is the last module that is loaded into memory, and,
 * for example, can contain the root FS (useful for embedded systems).
 */
hdrindex ++;
phys_copy(aout + hdrindex * A_MINHDR, vir2phys(&e_hdr), (phys_bytes) A_MINHDR);
if (e_hdr.a_flags & A_IMG) {
    kinfo.bootdev_base = e_hdr.a_syms;
    kinfo.bootdev_size = e_hdr.a_data;
}
}

```

```

#endif

```

```

/* MINIX is now ready. All boot image processes are on the ready queue.
 * Return to the assembly code to start running the current process.
 */
bill_ptr = proc_addr(IDLE);     /* it has to point somewhere */
announce();                     /* print MINIX startup banner */
restart();
}

```

```

/*=====
*
*          announce
*
*=====*/
PRIVATE void announce(void)
{
    /* Display the MINIX startup banner. */
    kprintf("\nMINIX %s.%s. "
           "Copyright 2006, Vrije Universiteit, Amsterdam, The Netherlands\n",
           OS_RELEASE, OS_VERSION);
#ifdef CHIP == INTEL
    /* Real mode, or 16/32-bit protected mode? */
    kprintf("Executing in %s mode.\n\n",
           machine.prot ? "32-bit protected" : "real");
#endif
}

/*=====
*
*          prepare_shutdown
*
*=====*/
PUBLIC void prepare_shutdown(how)
int how;
{
    /* This function prepares to shutdown MINIX. */
    static timer_t shutdown_timer;
    register struct proc *rp;
    message m;

    /* Send a signal to all system processes that are still alive to inform
    * them that the MINIX kernel is shutting down. A proper shutdown sequence
    * should be implemented by a user-space server. This mechanism is useful
    * as a backup in case of system panics, so that system processes can still
    * run their shutdown code, e.g, to synchronize the FS or to let the TTY
    * switch to the first console.
    */
#ifdef DEAD_CODE
    kprintf("Sending SIGKSTOP to system processes ... \n");
    for (rp=BEG_PROC_ADDR; rp<END_PROC_ADDR; rp++) {
        if (!isemptyp(rp) && (priv(rp)->s_flags & SYS_PROC) && !iskernelp(rp))
            send_sig(proc_nr(rp), SIGKSTOP);
    }
#endif

    /* Continue after 1 second, to give processes a chance to get scheduled to
    * do shutdown work. Set a watchdog timer to call shutdown(). The timer
    * argument passes the shutdown status.
    */
    kprintf("MINIX will now be shut down ... \n");
    tmr_arg(&shutdown_timer)->ta_int = how;
    set_timer(&shutdown_timer, get_uptime() + HZ, shutdown);
}

/*=====
*
*          shutdown
*
*=====*/
PRIVATE void shutdown(tp)
timer_t *tp;
{

```

```
/* This function is called from prepare_shutdown or stop_sequence to bring
 * down MINIX. How to shutdown is in the argument: RBT_HALT (return to the
 * monitor), RBT_MONITOR (execute given code), RBT_RESET (hard reset).
 */
int how = tmr_arg(tp)->ta_int;
ul6_t magic;

/* Now mask all interrupts, including the clock, and stop the clock. */
outb(INT_CTLMASK, ~0);
clock_stop();

if (mon_return && how != RBT_RESET) {
    /* Reinitialize the interrupt controllers to the BIOS defaults. */
    intr_init(0);
    outb(INT_CTLMASK, 0);
    outb(INT2_CTLMASK, 0);

    /* Return to the boot monitor. Set the program if not already done. */
    if (how != RBT_MONITOR) phys_copy(vir2phys(" "), kinfo.params_base, 1);
    level0(monitor);
}

/* Reset the system by jumping to the reset address (real mode), or by
 * forcing a processor shutdown (protected mode). First stop the BIOS
 * memory test by setting a soft reset flag.
 */
magic = STOP_MEM_CHECK;
phys_copy(vir2phys(&magic), SOFT_RESET_FLAG_ADDR, SOFT_RESET_FLAG_SIZE);
level0(reset);
}
```