

```
# Makefile for kernel

# Directories
u = /usr
i = $u/include
l = $u/lib
s = system

# Programs, flags, etc.
CC =      exec cc
CPP =     $l/cpp
LD =      $(CC) -o
CFLAGS =  -I$i
LDFLAGS = -i

HEAD = mpx.o
OBJS = start.o protect.o klib.o table.o kprintf.o main.o proc.o \
       i8259.o exception.o system.o clock.o utility.o debug.o
SYSTEM = system.a
LIBS = -ltimers

# What to make.
all: build
kernel build install: $(HEAD) $(OBJS)
    cd system && $(MAKE) -$(MAKEFLAGS) $@
    $(LD) $(CFLAGS) $(LDFLAGS) -o kernel \
    $(HEAD) $(OBJS) \
    $(SYSTEM) $(LIBS)
    install -S 0 kernel

clean:
    cd system && $(MAKE) -$(MAKEFLAGS) $@
    rm -f *.a *.o *~ *.bak kernel

depend:
    cd system && $(MAKE) -$(MAKEFLAGS) $@
    /usr/bin/mkdep "$(CC) -E $(CPPFLAGS)" *.c *.s > .depend

# How to build it
.s.o:
    $(CC) $(CFLAGS) -c -o $@ $<

.c.o:
    $(CC) $(CFLAGS) -c -o $@ $<

# Include generated dependencies.
klib.o: klib386.s klib88.s
mpx.o: mpx386.s mpx88.s
include .depend
```

```

/* This file contains the clock task, which handles time related functions.
 * Important events that are handled by the CLOCK include setting and
 * monitoring alarm timers and deciding when to (re)schedule processes.
 * The CLOCK offers a direct interface to kernel processes. System services
 * can access its services through system calls, such as sys_setalarm(). The
 * CLOCK task thus is hidden from the outside world.
 *
 * Changes:
 *   Oct 08, 2005   reordering and comment editing (A. S. Woodhull)
 *   Mar 18, 2004   clock interface moved to SYSTEM task (Jorrit N. Herder)
 *   Sep 30, 2004   source code documentation updated (Jorrit N. Herder)
 *   Sep 24, 2004   redesigned alarm timers (Jorrit N. Herder)
 *
 * The function do_clocktick() is triggered by the clock's interrupt
 * handler when a watchdog timer has expired or a process must be scheduled.
 *
 * In addition to the main clock_task() entry point, which starts the main
 * loop, there are several other minor entry points:
 *   clock_stop:      called just before MINIX shutdown
 *   get_uptime:      get realtime since boot in clock ticks
 *   set_timer:       set a watchdog timer (+)
 *   reset_timer:     reset a watchdog timer (+)
 *   read_clock:      read the counter of channel 0 of the 8253A timer
 *
 * (+) The CLOCK task keeps tracks of watchdog timers for the entire kernel.
 * The watchdog functions of expired timers are executed in do_clocktick().
 * It is crucial that watchdog functions not block, or the CLOCK task may
 * be blocked. Do not send() a message when the receiver is not expecting it.
 * Instead, notify(), which always returns, should be used.
 */

#include "kernel.h"
#include "proc.h"
#include <signal.h>
#include <minix/com.h>

/* Function prototype for PRIVATE functions. */
FORWARD _PROTOTYPE( void init_clock, (void) );
FORWARD _PROTOTYPE( int clock_handler, (irq_hook_t *hook) );
FORWARD _PROTOTYPE( int do_clocktick, (message *m_ptr) );
FORWARD _PROTOTYPE( void load_update, (void) );

/* Clock parameters. */
#define COUNTER_FREQ (2*TIMER_FREQ) /* counter frequency using square wave */
#define LATCH_COUNT 0x00 /* cc00xxxx, c = channel, x = any */
#define SQUARE_WAVE 0x36 /* ccaammmb, a = access, m = mode, b = BCD */
/* 11x11, 11 = LSB then MSB, x11 = sq wave */
#define TIMER_COUNT ((unsigned) (TIMER_FREQ/HZ)) /* initial value for counter*/
#define TIMER_FREQ 1193182L /* clock frequency for timer in PC and AT */

#define CLOCK_ACK_BIT 0x80 /* PS/2 clock interrupt acknowledge bit */

/* The CLOCK's timers queue. The functions in <timers.h> operate on this.
 * Each system process possesses a single synchronous alarm timer. If other
 * kernel parts want to use additional timers, they must declare their own
 * persistent (static) timer structure, which can be passed to the clock
 * via (re)set_timer().
 * When a timer expires its watchdog function is run by the CLOCK task.
 */
PRIVATE timer_t *clock_timers; /* queue of CLOCK timers */
PRIVATE clock_t next_timeout; /* realtime that next timer expires */

/* The time is incremented by the interrupt handler on each clock tick. */
PRIVATE clock_t realtime; /* real time clock */
PRIVATE irq_hook_t clock_hook; /* interrupt handler hook */

/*=====
 *                               clock_task                               *
 *=====*/
PUBLIC void clock_task()
{
/* Main program of clock task. If the call is not HARD_INT it is an error.
 */
    message m; /* message buffer for both input and output */

```

```

int result;                                /* result returned by the handler */

init_clock();                              /* initialize clock task */

/* Main loop of the clock task.  Get work, process it.  Never reply. */
while (TRUE) {

    /* Go get a message. */
    receive(ANY, &m);

    /* Handle the request.  Only clock ticks are expected. */
    switch (m.m_type) {
    case HARD_INT:
        result = do_clocktick(&m);        /* handle clock tick */
        break;
    default:
        /* illegal request type */
        kprintf("CLOCK: illegal request %d from %d.\n", m.m_type, m.m_source);
    }
}

/*=====
*
*                               do_clocktick
*=====*/
PRIVATE int do_clocktick(m_ptr)
message *m_ptr;                          /* pointer to request message */
{
    /* Despite its name, this routine is not called on every clock tick.  It
    * is called on those clock ticks when a lot of work needs to be done.
    */

    /* A process used up a full quantum.  The interrupt handler stored this
    * process in 'prev_ptr'.  First make sure that the process is not on the
    * scheduling queues.  Then announce the process ready again.  Since it has
    * no more time left, it gets a new quantum and is inserted at the right
    * place in the queues.  As a side-effect a new process will be scheduled.
    */
    if (prev_ptr->p_ticks_left <= 0 && priv(prev_ptr)->s_flags & PREEMPTIBLE) {
        lock_dequeue(prev_ptr);           /* take it off the queues */
        lock_enqueue(prev_ptr);           /* and reinsert it again */
    }

    /* Check if a clock timer expired and run its watchdog function. */
    if (next_timeout <= realtime) {
        tmrs_exptimers(&clock_timers, realtime, NULL);
        next_timeout = clock_timers == NULL ?
            TMR_NEVER : clock_timers->tmr_exp_time;
    }

    /* Inhibit sending a reply. */
    return(EDONTREPLY);
}

/*=====
*
*                               init_clock
*=====*/
PRIVATE void init_clock()
{
    /* Initialize the CLOCK's interrupt hook. */
    clock_hook.proc_nr_e = CLOCK;

    /* Initialize channel 0 of the 8253A timer to, e.g., 60 Hz, and register
    * the CLOCK task's interrupt handler to be run on every clock tick.
    */
    outb(TIMER_MODE, SQUARE_WAVE);        /* set timer to run continuously */
    outb(TIMER0, TIMER_COUNT);            /* load timer low byte */
    outb(TIMER0, TIMER_COUNT >> 8);       /* load timer high byte */
    put_irq_handler(&clock_hook, CLOCK_IRQ, clock_handler);
    enable_irq(&clock_hook);              /* ready for clock interrupts */

    /* Set a watchdog timer to periodically balance the scheduling queues. */
    balance_queues(NULL);                  /* side-effect sets new timer */
}

```

```

/*=====
 *
 *                                clock_stop
 *=====*/
PUBLIC void clock_stop()
{
/* Reset the clock to the BIOS rate. (For rebooting.) */
    outb(TIMER_MODE, 0x36);
    outb(TIMER0, 0);
    outb(TIMER0, 0);
}

/*=====
 *
 *                                clock_handler
 *=====*/
PRIVATE int clock_handler(hook)
irq_hook_t *hook;
{
/* This executes on each clock tick (i.e., every time the timer chip generates
 * an interrupt). It does a little bit of work so the clock task does not have
 * to be called on every tick. The clock task is called when:
 *
 *      (1) the scheduling quantum of the running process has expired, or
 *      (2) a timer has expired and the watchdog function should be run.
 *
 * Many global global and static variables are accessed here. The safety of
 * this must be justified. All scheduling and message passing code acquires a
 * lock by temporarily disabling interrupts, so no conflicts with calls from
 * the task level can occur. Furthermore, interrupts are not reentrant, the
 * interrupt handler cannot be bothered by other interrupts.
 *
 * Variables that are updated in the clock's interrupt handler:
 *
 *      lost_ticks:
 *          Clock ticks counted outside the clock task. This for example
 *          is used when the boot monitor processes a real mode interrupt.
 *
 *      realtime:
 *          The current uptime is incremented with all outstanding ticks.
 *
 *      proc_ptr, bill_ptr:
 *          These are used for accounting. It does not matter if proc.c
 *          is changing them, provided they are always valid pointers,
 *          since at worst the previous process would be billed.
 */
    register unsigned ticks;

/* Acknowledge the PS/2 clock interrupt. */
    if (machine.ps_mca) outb(PORT_B, inb(PORT_B) | CLOCK_ACK_BIT);

/* Get number of ticks and update realtime. */
    ticks = lost_ticks + 1;
    lost_ticks = 0;
    realtime += ticks;

/* Update user and system accounting times. Charge the current process for
 * user time. If the current process is not billable, that is, if a non-user
 * process is running, charge the billable process for system time as well.
 * Thus the unbillable process' user time is the billable user's system time.
 */
    proc_ptr->p_user_time += ticks;
    if (priv(proc_ptr)->s_flags & PREEMPTIBLE) {
        proc_ptr->p_ticks_left -= ticks;
    }
    if (! (priv(proc_ptr)->s_flags & BILLABLE)) {
        bill_ptr->p_sys_time += ticks;
        bill_ptr->p_ticks_left -= ticks;
    }

/* Update load average. */
    load_update();

/* Check if do_clocktick() must be called. Done for alarms and scheduling.
 * Some processes, such as the kernel tasks, cannot be preempted.
 */
    if ((next_timeout <= realtime) || (proc_ptr->p_ticks_left <= 0)) {
        prev_ptr = proc_ptr;                /* store running process */
        lock_notify(HARDWARE, CLOCK);        /* send notification */
    }
}

```

```

    }
    return(1);                                /* reenable interrupts */
}

/*=====
 *                               get_uptime                               *
 *=====*/
PUBLIC clock_t get_uptime()
{
    /* Get and return the current clock uptime in ticks. */
    return(realtime);
}

/*=====
 *                               set_timer                               *
 *=====*/
PUBLIC void set_timer(tp, exp_time, watchdog)
struct timer *tp;                            /* pointer to timer structure */
clock_t exp_time;                            /* expiration realtime */
tmr_func_t watchdog;                         /* watchdog to be called */
{
    /* Insert the new timer in the active timers list. Always update the
     * next timeout time by setting it to the front of the active list.
     */
    tmrs_settimer(&clock_timers, tp, exp_time, watchdog, NULL);
    next_timeout = clock_timers->tmr_exp_time;
}

/*=====
 *                               reset_timer                             *
 *=====*/
PUBLIC void reset_timer(tp)
struct timer *tp;                            /* pointer to timer structure */
{
    /* The timer pointed to by 'tp' is no longer needed. Remove it from both the
     * active and expired lists. Always update the next timeout time by setting
     * it to the front of the active list.
     */
    tmrs_clrtimer(&clock_timers, tp, NULL);
    next_timeout = (clock_timers == NULL) ?
        TMR_NEVER : clock_timers->tmr_exp_time;
}

/*=====
 *                               read_clock                              *
 *=====*/
PUBLIC unsigned long read_clock()
{
    /* Read the counter of channel 0 of the 8253A timer. This counter counts
     * down at a rate of TIMER_FREQ and restarts at TIMER_COUNT-1 when it
     * reaches zero. A hardware interrupt (clock tick) occurs when the counter
     * gets to zero and restarts its cycle.
     */
    unsigned count;

    outb(TIMER_MODE, LATCH_COUNT);
    count = inb(TIMER0);
    count |= (inb(TIMER0) << 8);

    return count;
}

/*=====
 *                               load_update                             *
 *=====*/
PRIVATE void load_update(void)
{
    ul6_t slot;
    int enqueued = -1, q;    /* -1: special compensation for IDLE. */
    struct proc *p;

    /* Load average data is stored as a list of numbers in a circular
     * buffer. Each slot accumulates _LOAD_UNIT_SECS of samples of
     * the number of runnable processes. Computations can then

```

```
    * be made of the load average over variable periods, in the
    * user library (see getloadavg(3)).
    */
slot = (realtime / HZ / _LOAD_UNIT_SECS) % _LOAD_HISTORY;
if(slot != kloadinfo.proc_last_slot) {
    kloadinfo.proc_load_history[slot] = 0;
    kloadinfo.proc_last_slot = slot;
}

/* Cumulation. How many processes are ready now? */
for(q = 0; q < NR_SCHED_QUEUES; q++)
    for(p = rdy_head[q]; p != NIL_PROC; p = p->p_nextready)
        enqueued++;

kloadinfo.proc_load_history[slot] += enqueued;

/* Up-to-dateness. */
kloadinfo.last_clock = realtime;
}
```

```
#ifndef CONFIG_H
#define CONFIG_H

/* This file defines the kernel configuration. It allows to set sizes of some
 * kernel buffers and to enable or disable debugging code, timing features,
 * and individual kernel calls.
 *
 * Changes:
 *   Jul 11, 2005          Created. (Jorrit N. Herder)
 */

/* In embedded and sensor applications, not all the kernel calls may be
 * needed. In this section you can specify which kernel calls are needed
 * and which are not. The code for unneeded kernel calls is not included in
 * the system binary, making it smaller. If you are not sure, it is best
 * to keep all kernel calls enabled.
 */
#define USE_FORK            1      /* fork a new process */
#define USE_NEWMAP          1      /* set a new memory map */
#define USE_EXEC            1      /* update process after execute */
#define USE_EXIT            1      /* clean up after process exit */
#define USE_TRACE          1      /* process information and tracing */
#define USE_GETKSIG        1      /* retrieve pending kernel signals */
#define USE_ENDKSIG        1      /* finish pending kernel signals */
#define USE_KILL           1      /* send a signal to a process */
#define USE_SIGSEND        1      /* send POSIX-style signal */
#define USE_SIGRETURN      1      /* sys_sigreturn(proc_nr, ctxt_ptr, flags) */
#define USE_ABORT          1      /* shut down MINIX */
#define USE_GETINFO        1      /* retrieve a copy of kernel data */
#define USE_TIMES          1      /* get process and system time info */
#define USE_SETALARM       1      /* schedule a synchronous alarm */
#define USE_DEVIO          1      /* read or write a single I/O port */
#define USE_VDEVIO         1      /* process vector with I/O requests */
#define USE_SDEVIO         1      /* perform I/O request on a buffer */
#define USE_IRQCTL         1      /* set an interrupt policy */
#define USE_SEGCTL         1      /* set up a remote segment */
#define USE_PRIVCTL        1      /* system privileges control */
#define USE_NICE           1      /* change scheduling priority */
#define USE_UMAP           1      /* map virtual to physical address */
#define USE_VIRCOPY        1      /* copy using virtual addressing */
#define USE_VIRVCOPY       1      /* vector with virtual copy requests */
#define USE_PHYSCOPY       1      /* copy using physical addressing */
#define USE_PHYSVCOPY      1      /* vector with physical copy requests */
#define USE_MEMSET         1      /* write char to a given memory area */

/* Length of program names stored in the process table. This is only used
 * for the debugging dumps that can be generated with the IS server. The PM
 * server keeps its own copy of the program name.
 */
#define P_NAME_LEN         8

/* Kernel diagnostics are written to a circular buffer. After each message,
 * a system server is notified and a copy of the buffer can be retrieved to
 * display the message. The buffers size can safely be reduced.
 */
#define KMESS_BUF_SIZE     256

/* Buffer to gather randomness. This is used to generate a random stream by
 * the MEMORY driver when reading from /dev/random.
 */
#define RANDOM_ELEMENTS    32

/* This section contains defines for valuable system resources that are used
 * by device drivers. The number of elements of the vectors is determined by
 * the maximum needed by any given driver. The number of interrupt hooks may
 * be incremented on systems with many device drivers.
 */
#define NR_IRQ_HOOKS       16      /* number of interrupt hooks */
#define VDEVIO_BUF_SIZE    64      /* max elements per VDEVIO request */
#define VCOPY_VEC_SIZE     16      /* max elements per VCOPY request */

/* How many bytes for the kernel stack. Space allocated in mpx.s. */
#define K_STACK_BYTES      1024
```

```
/* This section allows to enable kernel debugging and timing functionality.
 * For normal operation all options should be disabled.
 */
#define DEBUG_SCHED_CHECK 0    /* sanity check of scheduling queues */
#define DEBUG_TIME_LOCKS 0    /* measure time spent in locks */
#endif /* CONFIG_H */
```



```

/* General macros and constants used by the kernel. */
#ifndef CONST_H
#define CONST_H

#include <ibm/interrupt.h>      /* interrupt numbers and hardware vectors */
#include <ibm/ports.h>          /* port addresses and magic numbers */
#include <ibm/bios.h>           /* BIOS addresses, sizes and magic numbers */
#include <ibm/cpu.h>           /* BIOS addresses, sizes and magic numbers */
#include <minix/config.h>
#include "config.h"

/* To translate an address in kernel space to a physical address. This is
 * the same as umap_local(proc_ptr, D, vir, sizeof(*vir)), but less costly.
 */
#define vir2phys(vir)    (kinfo.data_base + (vir_bytes) (vir))

/* Map a process number to a privilege structure id. */
#define s_nr_to_id(n)    (NR_TASKS + (n) + 1)

/* Translate a pointer to a field in a structure to a pointer to the structure
 * itself. So it translates '&struct_ptr->field' back to 'struct_ptr'.
 */
#define structof(type, field, ptr) \
    ((type *) (((char *) (ptr)) - offsetof(type, field)))

/* Translate an endpoint number to a process number, return success. */
#define isokendpt(e,p)  isokendpt_d((e),(p),0)
#define okendpt(e,p)    isokendpt_d((e),(p),1)

/* Constants used in virtual_copy(). Values must be 0 and 1, respectively. */
#define _SRC_    0
#define _DST_    1

/* Number of random sources */
#define RANDOM_SOURCES 16

/* Constants and macros for bit map manipulation. */
#define BITCHUNK_BITS    (sizeof(bitchunk_t) * CHAR_BIT)
#define BITMAP_CHUNKS(nr_bits) (((nr_bits)+BITCHUNK_BITS-1)/BITCHUNK_BITS)
#define MAP_CHUNK(map,bit) (map)[((bit)/BITCHUNK_BITS)]
#define CHUNK_OFFSET(bit) ((bit)%BITCHUNK_BITS)
#define GET_BIT(map,bit) ( MAP_CHUNK(map,bit) & (1 << CHUNK_OFFSET(bit)) )
#define SET_BIT(map,bit) ( MAP_CHUNK(map,bit) |= (1 << CHUNK_OFFSET(bit)) )
#define UNSET_BIT(map,bit) ( MAP_CHUNK(map,bit) &= ~(1 << CHUNK_OFFSET(bit)) )

#define get_sys_bit(map,bit) \
    ( MAP_CHUNK(map.chunk,bit) & (1 << CHUNK_OFFSET(bit)) )
#define set_sys_bit(map,bit) \
    ( MAP_CHUNK(map.chunk,bit) |= (1 << CHUNK_OFFSET(bit)) )
#define unset_sys_bit(map,bit) \
    ( MAP_CHUNK(map.chunk,bit) &= ~(1 << CHUNK_OFFSET(bit)) )
#define NR_SYS_CHUNKS    BITMAP_CHUNKS(NR_SYS_PROCS)

#if (CHIP == INTEL)

/* Program stack words and masks. */
#define INIT_PSW        0x0200    /* initial psw */
#define INIT_TASK_PSW  0x1200    /* initial psw for tasks (with IOPL 1) */
#define TRACEBIT        0x0100    /* OR this with psw in proc[] for tracing */
#define SETPSW(rp, new)    /* permits only certain bits to be set */ \
    ((rp)->p_reg.psw = (rp)->p_reg.psw & ~0xCD5 | (new) & 0xCD5)
#define IF_MASK 0x00000200
#define IOPL_MASK 0x003000

#if DEBUG_LOCK_CHECK
#define reallock(c, v)    { if (!(read_cpu_flags() & X86_FLAG_I)) { kinfo.relocking++; } el
se { intr_disable(); } }
#else
#define reallock(c, v)    intr_disable()
#endif

#define realunlock(c)    intr_enable()

/* Disable/ enable hardware interrupts. The parameters of lock() and unlock()

```

```
* are used when debugging is enabled. See debug.h for more information.
*/
#define lock(c, v)      realloc(c, v)
#define unlock(c)      realunlock(c)

/* Sizes of memory tables. The boot monitor distinguishes three memory areas,
 * namely low mem below 1M, 1M-16M, and mem after 16M. More chunks are needed
 * for DOS MINIX.
 */
#define NR_MEMS          8

#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)
/* M68000 specific constants go here. */
#endif /* (CHIP == M68000) */

#endif /* CONST_H */
```

```

/* This file implements kernel debugging functionality that is not included
 * in the standard kernel. Available functionality includes timing of lock
 * functions and sanity checking of the scheduling queues.
 */

```

```

#include "kernel.h"
#include "proc.h"
#include "debug.h"
#include <limits.h>

```

```

#if DEBUG_TIME_LOCKS          /* only include code if enabled */

```

```

/* Data structures to store lock() timing data. */
struct lock_timingdata timingdata[TIMING_CATEGORIES];
static unsigned long starttimes[TIMING_CATEGORIES][2];

```

```

#define HIGHCOUNT      0
#define LOWCOUNT       1

```

```

void timer_start(int cat, char *name)
{
    static int init = 0;
    unsigned long h, l;
    int i;

    if (cat < 0 || cat >= TIMING_CATEGORIES) return;

    for(i = 0; i < sizeof(timingdata[0].names) && *name; i++)
        timingdata[cat].names[i] = *name++;
    timingdata[0].names[sizeof(timingdata[0].names)-1] = '\0';

    if (starttimes[cat][HIGHCOUNT]) { return; }

    if (!init) {
        int t, f;
        init = 1;
        for(t = 0; t < TIMING_CATEGORIES; t++) {
            timingdata[t].lock_timings_range[0] = 0;
            timingdata[t].resets = timingdata[t].misses =
                timingdata[t].measurements = 0;
        }
    }

    read_tsc(&starttimes[cat][HIGHCOUNT], &starttimes[cat][LOWCOUNT]);
}

```

```

void timer_end(int cat)
{
    unsigned long h, l, d = 0, binsize;
    int bin;

    read_tsc(&h, &l);
    if (cat < 0 || cat >= TIMING_CATEGORIES) return;
    if (!starttimes[cat][HIGHCOUNT]) {
        timingdata[cat].misses++;
        return;
    }
    if (starttimes[cat][HIGHCOUNT] == h) {
        d = (l - starttimes[cat][1]);
    } else if (starttimes[cat][HIGHCOUNT] == h-1 &&
        starttimes[cat][LOWCOUNT] > 1) {
        d = ((ULONG_MAX - starttimes[cat][LOWCOUNT]) + 1);
    } else {
        timingdata[cat].misses++;
        return;
    }
    starttimes[cat][HIGHCOUNT] = 0;
    if (!timingdata[cat].lock_timings_range[0] ||
        d < timingdata[cat].lock_timings_range[0] ||
        d > timingdata[cat].lock_timings_range[1]) {
        int t;
        if (!timingdata[cat].lock_timings_range[0] ||
            d < timingdata[cat].lock_timings_range[0])
            timingdata[cat].lock_timings_range[0] = d;
    }
}

```

```

        if (!timingdata[cat].lock_timings_range[1] ||
            d > timingdata[cat].lock_timings_range[1])
            timingdata[cat].lock_timings_range[1] = d;
        for(t = 0; t < TIMING_POINTS; t++)
            timingdata[cat].lock_timings[t] = 0;
        timingdata[cat].binsize =
            (timingdata[cat].lock_timings_range[1] -
             timingdata[cat].lock_timings_range[0])/(TIMING_POINTS+1);
        if (timingdata[cat].binsize < 1)
            timingdata[cat].binsize = 1;
        timingdata[cat].resets++;
    }
    bin = (d-timingdata[cat].lock_timings_range[0]) /
        timingdata[cat].binsize;
    if (bin < 0 || bin >= TIMING_POINTS) {
        int t;
        /* this indicates a bug, but isn't really serious */
        for(t = 0; t < TIMING_POINTS; t++)
            timingdata[cat].lock_timings[t] = 0;
        timingdata[cat].misses++;
    } else {
        timingdata[cat].lock_timings[bin]++;
        timingdata[cat].measurements++;
    }

    return;
}

#endif /* DEBUG_TIME_LOCKS */

#if DEBUG_SCHED_CHECK /* only include code if enabled */

#define PROCLIMIT 10000

PUBLIC void
check_runqueues(char *when)
{
    int q, l = 0;
    register struct proc *xp;

    for (xp = BEG_PROC_ADDR; xp < END_PROC_ADDR; ++xp) {
        xp->p_found = 0;
        if (l++ > PROCLIMIT) { panic("check error", NO_NUM); }
    }

    for (q=0; q < NR_SCHED_QUEUES; q++) {
        if (rdy_head[q] && !rdy_tail[q]) {
            kprintf("head but no tail: %s", when);
            panic("scheduling error", NO_NUM);
        }
        if (!rdy_head[q] && rdy_tail[q]) {
            kprintf("tail but no head: %s", when);
            panic("scheduling error", NO_NUM);
        }
        if (rdy_tail[q] && rdy_tail[q]->p_nextready != NIL_PROC) {
            kprintf("tail and tail->next not null: %s", when);
            panic("scheduling error", NO_NUM);
        }
        for(xp = rdy_head[q]; xp != NIL_PROC; xp = xp->p_nextready) {
            if (!xp->p_ready) {
                kprintf("scheduling error: unready on runq: %s\n", when);

                panic("found unready process on run queue", NO_NUM);
            }
            if (xp->p_priority != q) {
                kprintf("scheduling error: wrong priority: %s\n", when);

                panic("wrong priority", NO_NUM);
            }
            if (xp->p_found) {
                kprintf("scheduling error: double scheduling: %s\n", when);
                panic("proc more than once on scheduling queue", NO_NUM);
            }
            xp->p_found = 1;

```

```
    if (xp->p_nextready == NIL_PROC && rdy_tail[q] != xp) {
        kprintf("scheduling error: last element not tail: %s\n", when);
        panic("scheduling error", NO_NUM);
    }
    if (l++ > PROCLIMIT) panic("loop in schedule queue?", NO_NUM);
}

for (xp = BEG_PROC_ADDR; xp < END_PROC_ADDR; ++xp) {
    if (! isemptyp(xp) && xp->p_ready && ! xp->p_found) {
        kprintf("scheduling error: ready not on queue: %s\n", when);
        panic("ready proc not on scheduling queue", NO_NUM);
        if (l++ > PROCLIMIT) { panic("loop in proc.t?", NO_NUM); }
    }
}

#endif /* DEBUG_SCHED_CHECK */
```

```

#ifndef DEBUG_H
#define DEBUG_H

/* This header file defines all debugging constants and macros, and declares
 * some variables. Certain debugging features redefine standard constants
 * and macros. Therefore, this header file should be included after the
 * other kernel headers.
 */

#include <ansi.h>
#include "config.h"

/* Enable prints such as
 * . send/receive failed due to deadlock or dead source or dead destination
 * . trap not allowed
 * . bogus message pointer
 * . kernel call number not allowed by this process
 *
 * Of course the call still fails, but nothing is printed if these warnings
 * are disabled.
 */
#define DEBUG_ENABLE_IPC_WARNINGS          0

/* It's interesting to measure the time spent withing locked regions, because
 * this is the time that the system is deaf to interrupts.
 */
#if DEBUG_TIME_LOCKS

#define TIMING_POINTS          20          /* timing resolution */
#define TIMING_CATEGORIES      20
#define TIMING_NAME           10

/* Definition of the data structure to store lock() timing data. */
struct lock_timingdata {
    char names[TIMING_NAME];
    unsigned long lock_timings[TIMING_POINTS];
    unsigned long lock_timings_range[2];
    unsigned long binsize, resets, misses, measurements;
};

/* The data is declared here, but allocated in debug.c. */
extern struct lock_timingdata timingdata[TIMING_CATEGORIES];

/* Prototypes for the timing functionality. */
_PROTOTYPE( void timer_start, (int cat, char *name) );
_PROTOTYPE( void timer_end, (int cat) );

#define locktimestart(c, v) timer_start(c, v)
#define locktimeend(c) timer_end(c)
#else
#define locktimestart(c, v)
#define locktimeend(c)
#endif /* DEBUG_TIME_LOCKS */

/* This check makes sure that the scheduling queues are in a consistent state.
 * The check is run when the queues are updated with ready() and unready().
 */
#if DEBUG_SCHED_CHECK
_PROTOTYPE( void check_runqueues, (char *when) );
#endif /* DEBUG_SCHED_CHECK */

/* The timing and checking of kernel locking requires a redefine of the lock()
 * and unlock() macros. That's done here. This redefine requires that this
 * header is included after the other kernel headers.
 */
#if (DEBUG_TIME_LOCKS || DEBUG_LOCK_CHECK)
# undef lock
# define lock(c, v)      do { realloc(c, v); locktimestart(c, v); } while(0)
# undef unlock
# define unlock(c)      do { locktimeend(c); realunlock(c); } while(0)
#endif

#endif /* DEBUG_H */

```

```

/* This file contains a simple exception handler.  Exceptions in user
 * processes are converted to signals. Exceptions in a kernel task cause
 * a panic.
 */

#include "kernel.h"
#include <signal.h>
#include "proc.h"

/*=====
 *                               exception                               *
 *=====*/
PUBLIC void exception(vec_nr)
unsigned vec_nr;
{
/* An exception or unexpected interrupt has occurred. */

    struct ex_s {
        char *msg;
        int signum;
        int minprocessor;
    };
    static struct ex_s ex_data[] = {
        { "Divide error", SIGFPE, 86 },
        { "Debug exception", SIGTRAP, 86 },
        { "Nonmaskable interrupt", SIGBUS, 86 },
        { "Breakpoint", SIGEMT, 86 },
        { "Overflow", SIGFPE, 86 },
        { "Bounds check", SIGFPE, 186 },
        { "Invalid opcode", SIGILL, 186 },
        { "Coprocessor not available", SIGFPE, 186 },
        { "Double fault", SIGBUS, 286 },
        { "Coprocessor segment overrun", SIGSEGV, 286 },
        { "Invalid TSS", SIGSEGV, 286 },
        { "Segment not present", SIGSEGV, 286 },
        { "Stack exception", SIGSEGV, 286 }, /* STACK_FAULT already used */
        { "General protection", SIGSEGV, 286 },
        { "Page fault", SIGSEGV, 386 }, /* not close */
        { NIL_PTR, SIGILL, 0 }, /* probably software trap */
        { "Coprocessor error", SIGFPE, 386 },
    };
    register struct ex_s *ep;
    struct proc *saved_proc;

    /* Save proc_ptr, because it may be changed by debug statements. */
    saved_proc = proc_ptr;

    ep = &ex_data[vec_nr];

    if (vec_nr == 2) { /* spurious NMI on some machines */
        kprintf("got spurious NMI\n");
        return;
    }

    /* If an exception occurs while running a process, the k_reenter variable
     * will be zero. Exceptions in interrupt handlers or system traps will make
     * k_reenter larger than zero.
     */
    if (k_reenter == 0 && !iskernelp(saved_proc)) {
#ifdef 0
        {
            kprintf(
                "exception for process %d, pc = 0x%x:0x%x, sp = 0x%x:0x%x\n",
                proc_nr(saved_proc),
                saved_proc->p_reg.cs, saved_proc->p_reg.pc,
                saved_proc->p_reg.ss, saved_proc->p_reg.sp);
            kprintf("edi = 0x%x\n", saved_proc->p_reg.di);
        }
#endif
        cause_sig(proc_nr(saved_proc), ep->signum);
        return;
    }
}

```

```
/* Exception in system code. This is not supposed to happen. */
if (ep->msg == NIL_PTR || machine.processor < ep->minprocessor)
    kprintf("\nIntel-reserved exception %d\n", vec_nr);
else
    kprintf("\n%s\n", ep->msg);
kprintf("k_reenter = %d", k_reenter);
kprintf("process %d (%s)", proc_nr(saved_proc), saved_proc->p_name);
kprintf("pc = %u:0x%x", (unsigned) saved_proc->p_reg.cs,
        (unsigned) saved_proc->p_reg.pc);

panic("exception in a kernel task", NO_NUM);
}
```



```
#ifndef GLO_H
#define GLO_H

/* Global variables used in the kernel. This file contains the declarations;
 * storage space for the variables is allocated in table.c, because EXTERN is
 * defined as extern unless the _TABLE definition is seen. We rely on the
 * compiler's default initialization (0) for several global variables.
 */
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif

#include <minix/config.h>
#include "config.h"

/* Variables relating to shutting down MINIX. */
EXTERN char kernel_exception; /* TRUE after system exceptions */
EXTERN char shutdown_started; /* TRUE after shutdowns / reboots */

/* Kernel information structures. This groups vital kernel information. */
EXTERN phys_bytes aout; /* address of a.out headers */
EXTERN struct kinfo kinfo; /* kernel information for users */
EXTERN struct machine machine; /* machine information for users */
EXTERN struct kmessages kmess; /* diagnostic messages in kernel */
EXTERN struct randomness krandom; /* gather kernel random information */
EXTERN struct loadinfo kloadinfo; /* status of load average */

/* Process scheduling information and the kernel reentry count. */
EXTERN struct proc *prev_ptr; /* previously running process */
EXTERN struct proc *proc_ptr; /* pointer to currently running process */
EXTERN struct proc *next_ptr; /* next process to run after restart() */
EXTERN struct proc *bill_ptr; /* process to bill for clock ticks */
EXTERN char k_reenter; /* kernel reentry count (entry count less 1) */
EXTERN unsigned lost_ticks; /* clock ticks counted outside clock task */

#if (CHIP == INTEL)

/* Interrupt related variables. */
EXTERN irq_hook_t irq_hooks[NR_IRQ_HOOKS]; /* hooks for general use */
EXTERN irq_hook_t *irq_handlers[NR_IRQ_VECTORS]; /* list of IRQ handlers */
EXTERN int irq_actids[NR_IRQ_VECTORS]; /* IRQ ID bits active */
EXTERN int irq_use; /* map of all in-use irq's */

/* Miscellaneous. */
EXTERN reg_t mon_ss, mon_sp; /* boot monitor stack */
EXTERN int mon_return; /* true if we can return to monitor */
EXTERN int do_serial_debug;
EXTERN int who_e, who_p; /* message source endpoint and proc */

/* VM */
EXTERN phys_bytes vm_base;
EXTERN phys_bytes vm_size;
EXTERN phys_bytes vm_mem_high;

/* Variables that are initialized elsewhere are just extern here. */
extern struct boot_image image[]; /* system image processes */
extern char *t_stack[]; /* task stack space */
extern struct segdesc_s gdt[]; /* global descriptor table */

EXTERN _PROTOTYPE( void (*level0_func), (void) );
#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)
/* M68000 specific variables go here. */
#endif

#endif /* GLO_H */
```

```

/* This file contains routines for initializing the 8259 interrupt controller:
 *   put_irq_handler: register an interrupt handler
 *   rm_irq_handler: deregister an interrupt handler
 *   intr_handle:     handle a hardware interrupt
 *   intr_init:       initialize the interrupt controller(s)
 */

#include "kernel.h"
#include "proc.h"
#include <minix/com.h>

#define ICW1_AT      0x11      /* edge triggered, cascade, need ICW4 */
#define ICW1_PC      0x13      /* edge triggered, no cascade, need ICW4 */
#define ICW1_PS      0x19      /* level triggered, cascade, need ICW4 */
#define ICW4_AT_SLAVE 0x01      /* not SFNM, not buffered, normal EOI, 8086 */
#define ICW4_AT_MASTER 0x05     /* not SFNM, not buffered, normal EOI, 8086 */
#define ICW4_PC_SLAVE 0x09     /* not SFNM, buffered, normal EOI, 8086 */
#define ICW4_PC_MASTER 0x0D    /* not SFNM, buffered, normal EOI, 8086 */

#if _WORD_SIZE == 2
typedef _PROTOTYPE( void (*vecaddr_t), (void) );

FORWARD _PROTOTYPE( void set_vec, (int vec_nr, vecaddr_t addr) );

PRIVATE vecaddr_t int_vec[] = {
    int00, int01, int02, int03, int04, int05, int06, int07,
};

PRIVATE vecaddr_t irq_vec[] = {
    hwint00, hwint01, hwint02, hwint03, hwint04, hwint05, hwint06, hwint07,
    hwint08, hwint09, hwint10, hwint11, hwint12, hwint13, hwint14, hwint15,
};
#else
#define set_vec(nr, addr)      ((void)0)
#endif

/*=====
 *                               intr_init                               *
 *=====*/
PUBLIC void intr_init(mine)
int mine;
{
    /* Initialize the 8259s, finishing with all interrupts disabled. This is
     * only done in protected mode, in real mode we don't touch the 8259s, but
     * use the BIOS locations instead. The flag "mine" is set if the 8259s are
     * to be programmed for MINIX, or to be reset to what the BIOS expects.
     */
    int i;

    intr_disable();

    if (machine.protected) {
        /* The AT and newer PS/2 have two interrupt controllers, one master,
         * one slaved at IRQ 2. (We don't have to deal with the PC that
         * has just one controller, because it must run in real mode.)
         */
        outb(INT_CTL, machine.ps_mca ? ICW1_PS : ICW1_AT);
        outb(INT_CTLMASK, mine ? IRQ0_VECTOR : BIOS_IRQ0_VEC);
        outb(INT_CTLMASK, (1 << CASCADE_IRQ));          /* ICW2 for master */
        outb(INT_CTLMASK, ICW4_AT_MASTER);              /* ICW3 tells slaves */
        outb(INT_CTLMASK, ~(1 << CASCADE_IRQ));          /* IRQ 0-7 mask */
        outb(INT2_CTL, machine.ps_mca ? ICW1_PS : ICW1_AT);
        outb(INT2_CTLMASK, mine ? IRQ8_VECTOR : BIOS_IRQ8_VEC);
        outb(INT2_CTLMASK, CASCADE_IRQ);                 /* ICW2 for slave */
        outb(INT2_CTLMASK, ICW4_AT_SLAVE);              /* ICW3 is slave nr */
        outb(INT2_CTLMASK, ~0);                          /* IRQ 8-15 mask */

        /* Copy the BIOS vectors from the BIOS to the Minix location, so we
         * can still make BIOS calls without reprogramming the i8259s.
         */
    }
}

#if IRQ0_VECTOR != BIOS_IRQ0_VEC
    phys_copy(BIOS_VECTOR(0) * 4L, VECTOR(0) * 4L, 8 * 4L);
#endif

```

```

#endif
#if IRQ8_VECTOR != BIOS_IRQ8_VEC
    phys_copy(BIOS_VECTOR(8) * 4L, VECTOR(8) * 4L, 8 * 4L);
#endif
} else {
    /* Use the BIOS interrupt vectors in real mode. We only reprogram the
     * exceptions here, the interrupt vectors are reprogrammed on demand.
     * SYS_VECTOR is the Minix system call for message passing.
     */
    for (i = 0; i < 8; i++) set_vec(i, int_vec[i]);
    set_vec(SYS_VECTOR, s_call);
}
}

/*=====
 *                               put_irq_handler                               *
 *=====*/
PUBLIC void put_irq_handler(hook, irq, handler)
irq_hook_t *hook;
int irq;
irq_handler_t handler;
{
    /* Register an interrupt handler. */
    int id;
    irq_hook_t **line;

    if (irq < 0 || irq >= NR_IRQ_VECTORS)
        panic("invalid call to put_irq_handler", irq);

    line = &irq_handlers[irq];
    id = 1;
    while (*line != NULL) {
        if (hook == *line) return;          /* extra initialization */
        line = &(*line)->next;
        id <= 1;
    }
    if (id == 0) panic("Too many handlers for irq", irq);

    hook->next = NULL;
    hook->handler = handler;
    hook->irq = irq;
    hook->id = id;
    *line = hook;

    irq_use |= 1 << irq;
}

/*=====
 *                               rm_irq_handler                               *
 *=====*/
PUBLIC void rm_irq_handler(hook)
irq_hook_t *hook;
{
    /* Unregister an interrupt handler. */
    int irq = hook->irq;
    int id = hook->id;
    irq_hook_t **line;

    if (irq < 0 || irq >= NR_IRQ_VECTORS)
        panic("invalid call to rm_irq_handler", irq);

    line = &irq_handlers[irq];
    while (*line != NULL) {
        if ((*line)->id == id) {
            (*line) = (*line)->next;
            if (!irq_handlers[irq]) irq_use &= ~(1 << irq);
            return;
        }
        line = &(*line)->next;
    }
    /* When the handler is not found, normally return here. */
}

/*=====

```

```

*                                     intr_handle                                     *
*=====*/
PUBLIC void intr_handle(hook)
irq_hook_t *hook;
{
/* Call the interrupt handlers for an interrupt with the given hook list.
 * The assembly part of the handler has already masked the IRQ, reenabled the
 * controller(s) and enabled interrupts.
 */

/* Call list of handlers for an IRQ. */
while (hook != NULL) {
/* For each handler in the list, mark it active by setting its ID bit,
 * call the function, and unmark it if the function returns true.
 */
irq_actids[hook->irq] |= hook->id;
if ((*hook->handler)(hook)) irq_actids[hook->irq] &= ~hook->id;
hook = hook->next;
}

/* The assembly code will now disable interrupts, unmask the IRQ if and only
 * if all active ID bits are cleared, and restart a process.
 */
}

#if _WORD_SIZE == 2
/*=====*/
*                                     set_vec                                     *
*=====*/
PRIVATE void set_vec(vec_nr, addr)
int vec_nr;                /* which vector */
vecaddr_t addr;            /* where to start */
{
/* Set up a real mode interrupt vector. */

ul6_t vec[2];

/* Build the vector in the array 'vec'. */
vec[0] = (ul6_t) addr;
vec[1] = (ul6_t) physb_to_hclick(code_base);

/* Copy the vector into place. */
phys_copy(vir2phys(vec), vec_nr * 4L, 4L);
}
#endif /* _WORD_SIZE == 2 */

```

```
#ifndef IPC_H
#define IPC_H

/* This header file defines constants for MINIX inter-process communication.
 * These definitions are used in the file proc.c.
 */
#include <minix/com.h>

/* Masks and flags for system calls. */
#define SYSCALL_FUNC      0x000F /* mask for system call function */
#define SYSCALL_FLAGS     0x00F0 /* mask for system call flags */
#define NON_BLOCKING      0x0010 /* do not block if target not ready */

/* System call numbers that are passed when trapping to the kernel. The
 * numbers are carefully defined so that it can easily be seen (based on
 * the bits that are on) which checks should be done in sys_call().
 */
#define SEND              1      /* 0001 : blocking send */
#define RECEIVE           2      /* 0010 : blocking receive */
#define SENDREC           3      /* 0011 : SEND + RECEIVE */
#define NOTIFY            4      /* 0100 : nonblocking notify */
#define ECHO              8      /* 1000 : echo a message */

#define IPC_REQUEST       5      /* 0101 : blocking request */
#define IPC_REPLY         6      /* 0110 : nonblocking reply */
#define IPC_NOTIFY        7      /* 0111 : nonblocking notification */
#define IPC_RECEIVE       9      /* 1001 : blocking receive */

/* The following bit masks determine what checks that should be done. */
#define CHECK_PTR         0xBB   /* 1011 1011 : validate message buffer */
#define CHECK_DST         0x55   /* 0101 0101 : validate message destination */
#define CHECK_DEADLOCK    0x93   /* 1001 0011 : check for deadlock */

#endif /* IPC_H */
```

```
#ifndef KERNEL_H
#define KERNEL_H

/* This is the master header for the kernel. It includes some other files
 * and defines the principal constants.
 */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */
#define _SYSTEM           1    /* tell headers that this is the kernel */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>    /* global configuration, MUST be first */
#include <ansi.h>            /* C style: ANSI or K&R, MUST be second */
#include <sys/types.h>       /* general system types */
#include <minix/const.h>     /* MINIX specific constants */
#include <minix/type.h>      /* MINIX specific types, e.g. message */
#include <minix/ipc.h>       /* MINIX run-time system */
#include <timers.h>          /* watchdog timer management */
#include <errno.h>           /* return codes and error numbers */

#if (CHIP == INTEL)
#include <ibm/portio.h>      /* device I/O and toggle interrupts */
#endif

/* Important kernel header files. */
#include "config.h"          /* configuration, MUST be first */
#include "const.h"           /* constants, MUST be second */
#include "type.h"            /* type definitions, MUST be third */
#include "proto.h"           /* function prototypes */
#include "glo.h"             /* global variables */
#include "ipc.h"             /* IPC constants */
#include "debug.h"           /* debugging, MUST be last kernel header */

#endif /* KERNEL_H */
```

```
#  
! Chooses between the 8086 and 386 versions of the low level kernel code.  
  
#include <minix/config.h>  
#if _WORD_SIZE == 2  
#include "klib88.s"  
#else  
#include "klib386.s"  
#endif
```

```

#
! sections

.sect .text; .sect .rom; .sect .data; .sect .bss

#include <minix/config.h>
#include <minix/const.h>
#include "const.h"
#include "sconst.h"
#include "protect.h"

! This file contains a number of assembly code utility routines needed by the
! kernel.  They are:

.define _monitor      ! exit Minix and return to the monitor
.define _int86        ! let the monitor make an 8086 interrupt call
.define _cp_mess      ! copies messages from source to destination
.define _exit         ! dummy for library routines
.define __exit        ! dummy for library routines
.define ___exit       ! dummy for library routines
.define ___main       ! dummy for GCC
.define _phys_insw    ! transfer data from (disk controller) port to memory
.define _phys_insb    ! likewise byte by byte
.define _phys_outsw   ! transfer data from memory to (disk controller) port
.define _phys_outsb   ! likewise byte by byte
.define _enable_irq   ! enable an irq at the 8259 controller
.define _disable_irq  ! disable an irq
.define _phys_copy    ! copy data from anywhere to anywhere in memory
.define _phys_memset  ! write pattern anywhere in memory
.define _mem_rdw      ! copy one word from [segment:offset]
.define _reset        ! reset the system
.define _idle_task    ! task executed when there is no work
.define _level0       ! call a function at level 0
.define _read_tsc     ! read the cycle counter (Pentium and up)
.define _read_cpu_flags ! read the cpu flags
.define _read_cr0     ! read cr0
.define _write_cr0     ! write a value in cr0
.define _write_cr3     ! write a value in cr3 (root of the page table)

! The routines only guarantee to preserve the registers the C compiler
! expects to be preserved (ebx, esi, edi, ebp, esp, segment registers, and
! direction bit in the flags).

.sect .text
!*=====*
!*                               monitor                               *
!*=====*
! PUBLIC void monitor();
! Return to the monitor.

_monitor:
    mov     esp, (_mon_sp)          ! restore monitor stack pointer
    o16 mov  dx, SS_SELECTOR        ! monitor data segment
    mov     ds, dx
    mov     es, dx
    mov     fs, dx
    mov     gs, dx
    mov     ss, dx
    pop     edi
    pop     esi
    pop     ebp
    o16 retf                        ! return to the monitor

!*=====*
!*                               int86                               *
!*=====*
! PUBLIC void int86();
_int86:
    cmpb    (_mon_return), 0        ! is the monitor there?
    jnz     0f
    movb     ah, 0x01               ! an int 13 error seems appropriate
    movb     (_reg86+ 0), ah        ! reg86.w.f = 1 (set carry flag)
    movb     (_reg86+13), ah        ! reg86.b.ah = 0x01 = "invalid command"

```



```

0:    ret
    push    ebp                ! save C registers
    push    esi
    push    edi
    push    ebx
    pushf
    cli                        ! save flags
                                ! no interruptions

    inb     INT2_CTLMASK
    movb    ah, al
    inb     INT_CTLMASK
    push    eax                ! save interrupt masks
    mov     eax, (_irq_use)    ! map of in-use IRQ's
    and     eax, ~[1<<CLOCK_IRQ] ! keep the clock ticking
    outb    INT_CTLMASK      ! enable all unused IRQ's and vv.
    movb    al, ah
    outb    INT2_CTLMASK

    mov     eax, SS_SELECTOR   ! monitor data segment
    mov     ss, ax
    xchg    esp, (_mon_sp)    ! switch stacks
    push    (_reg86+36)        ! parameters used in INT call
    push    (_reg86+32)
    push    (_reg86+28)
    push    (_reg86+24)
    push    (_reg86+20)
    push    (_reg86+16)
    push    (_reg86+12)
    push    (_reg86+ 8)
    push    (_reg86+ 4)
    push    (_reg86+ 0)
    mov     ds, ax            ! remaining data selectors
    mov     es, ax
    mov     fs, ax
    mov     gs, ax
    push    cs
    push    return            ! kernel return address and selector
o16 jmpf    20+2*4+10*4+2*4(esp) ! make the call
return:
    pop     (_reg86+ 0)
    pop     (_reg86+ 4)
    pop     (_reg86+ 8)
    pop     (_reg86+12)
    pop     (_reg86+16)
    pop     (_reg86+20)
    pop     (_reg86+24)
    pop     (_reg86+28)
    pop     (_reg86+32)
    pop     (_reg86+36)
    lgdt    (_gdt+GDT_SELECTOR) ! reload global descriptor table
    jmpf    CS_SELECTOR:csinit ! restore everything
csinit: mov     eax, DS_SELECTOR
    mov     ds, ax
    mov     es, ax
    mov     fs, ax
    mov     gs, ax
    mov     ss, ax
    xchg    esp, (_mon_sp)      ! unswitch stacks
    lidt    (_gdt+IDT_SELECTOR) ! reload interrupt descriptor table
    andb    (_gdt+TSS_SELECTOR+DESC_ACCESS), ~0x02 ! clear TSS busy bit
    mov     eax, TSS_SELECTOR
    ltr     ax                  ! set TSS register

    pop     eax
    outb    INT_CTLMASK        ! restore interrupt masks
    movb    al, ah
    outb    INT2_CTLMASK

    add     (_lost_ticks), ecx  ! record lost clock ticks

    popf
    pop     ebx                ! restore flags
    pop     edi                ! restore C registers
    pop     esi

```

```

    pop    ebp
    ret

```

```

! *=====*
! *                                     cp_mess                                     *
! *=====*
! PUBLIC void cp_mess(int src, phys_clicks src_clicks, vir_bytes src_offset,
!                     phys_clicks dst_clicks, vir_bytes dst_offset);
! This routine makes a fast copy of a message from anywhere in the address
! space to anywhere else. It also copies the source address provided as a
! parameter to the call into the first word of the destination message.
!
! Note that the message size, "Msize" is in DWORDS (not bytes) and must be set
! correctly. Changing the definition of message in the type file and not
! changing it here will lead to total disaster.

CM_ARGS =      4 + 4 + 4 + 4 + 4      ! 4 + 4 + 4 + 4 + 4
!             es  ds edi esi eip      proc scl sof dcl dof

    .align 16
_cp_mess:
    cld
    push    esi
    push    edi
    push    ds
    push    es

    mov     eax, FLAT_DS_SELECTOR
    mov     ds, ax
    mov     es, ax

    mov     esi, CM_ARGS+4(esp)        ! src clicks
    shl     esi, CLICK_SHIFT
    add     esi, CM_ARGS+4+4(esp)      ! src offset
    mov     edi, CM_ARGS+4+4+4(esp)    ! dst clicks
    shl     edi, CLICK_SHIFT
    add     edi, CM_ARGS+4+4+4+4(esp)  ! dst offset

    mov     eax, CM_ARGS(esp)          ! process number of sender
    stos    ! copy number of sender to dest message
    add     esi, 4                    ! do not copy first word
    mov     ecx, Msize - 1            ! remember, first word does not count
    rep     movs                      ! copy the message

    pop     es
    pop     ds
    pop     edi
    pop     esi
    ret                                ! that is all folks!

```

```

! *=====*
! *                                     exit                                     *
! *=====*
! PUBLIC void exit();
! Some library routines use exit, so provide a dummy version.
! Actual calls to exit cannot occur in the kernel.
! GNU CC likes to call __main from main() for nonobvious reasons.

__exit:
__exit:
__exit:
    sti
    jmp     __exit

__main:
    ret

```

```

! *=====*
! *                                     phys_insw                                     *
! *=====*

```

```
! PUBLIC void phys_insw(Port_t port, phys_bytes buf, size_t count);
! Input an array from an I/O port. Absolute address version of insw().
```

```
_phys_insw:
    push    ebp
    mov     ebp, esp
    cld
    push    edi
    push    es
    mov     ecx, FLAT_DS_SELECTOR
    mov     es, cx
    mov     edx, 8(ebp)           ! port to read from
    mov     edi, 12(ebp)         ! destination addr
    mov     ecx, 16(ebp)         ! byte count
    shr     ecx, 1               ! word count
rep o16 insw                     ! input many words
    pop     es
    pop     edi
    pop     ebp
    ret
```

```
! *=====*
! *                               phys_insb                               *
! *=====*
```

```
! PUBLIC void phys_insb(Port_t port, phys_bytes buf, size_t count);
! Input an array from an I/O port. Absolute address version of insb().
```

```
_phys_insb:
    push    ebp
    mov     ebp, esp
    cld
    push    edi
    push    es
    mov     ecx, FLAT_DS_SELECTOR
    mov     es, cx
    mov     edx, 8(ebp)           ! port to read from
    mov     edi, 12(ebp)         ! destination addr
    mov     ecx, 16(ebp)         ! byte count
    shr     ecx, 1               ! word count
! rep insb                       ! input many bytes
    pop     es
    pop     edi
    pop     ebp
    ret
```

```
! *=====*
! *                               phys_outsw                               *
! *=====*
```

```
! PUBLIC void phys_outsw(Port_t port, phys_bytes buf, size_t count);
! Output an array to an I/O port. Absolute address version of outsw().
```

```
    .align 16
_phys_outsw:
    push    ebp
    mov     ebp, esp
    cld
    push    esi
    push    ds
    mov     ecx, FLAT_DS_SELECTOR
    mov     ds, cx
    mov     edx, 8(ebp)           ! port to write to
    mov     esi, 12(ebp)         ! source addr
    mov     ecx, 16(ebp)         ! byte count
    shr     ecx, 1               ! word count
rep o16 outsw                     ! output many words
    pop     ds
    pop     esi
    pop     ebp
    ret
```

```
! *=====*
```

```

!*=====phys_outsb*
!*=====
! PUBLIC void phys_outsb(Port_t port, phys_bytes buf, size_t count);
! Output an array to an I/O port. Absolute address version of outsb().

    .align 16
_phys_outsb:
    push    ebp
    mov     ebp, esp
    cld
    push    esi
    push    ds
    mov     ecx, FLAT_DS_SELECTOR
    mov     ds, cx
    mov     edx, 8(ebp)      ! port to write to
    mov     esi, 12(ebp)     ! source addr
    mov     ecx, 16(ebp)     ! byte count
    rep     outsb            ! output many bytes
    pop     ds
    pop     esi
    pop     ebp
    ret

!*=====
!*=====enable_irq*
!*=====*/
! PUBLIC void enable_irq(irq_hook_t *hook)
! Enable an interrupt request line by clearing an 8259 bit.
! Equivalent C code for hook->irq < 8:
!   if ((irq_actids[hook->irq] &= ~hook->id) == 0)
!       outb(INT_CTLMASK, inb(INT_CTLMASK) & ~(1 << irq));

    .align 16
_enable_irq:
    push    ebp
    mov     ebp, esp
    pushf
    cli
    mov     eax, 8(ebp)      ! hook
    mov     ecx, 8(eax)      ! irq
    mov     eax, 12(eax)     ! id bit
    not     eax
    and     _irq_actids(ecx*4), eax ! clear this id bit
    jnz     en_done          ! still masked by other handlers?
    movb    ah, ~1
    rolb    ah, cl           ! ah = ~(1 << (irq % 8))
    mov     edx, INT_CTLMASK ! enable irq < 8 at the master 8259
    cmpb    cl, 8
    jnb     0f
    mov     edx, INT2_CTLMASK ! enable irq ≥ 8 at the slave 8259
0:    inb     dx
    andb    al, ah
    outb    dx              ! clear bit at the 8259
en_done: popf
    leave
    ret

!*=====
!*=====disable_irq*
!*=====*/
! PUBLIC int disable_irq(irq_hook_t *hook)
! Disable an interrupt request line by setting an 8259 bit.
! Equivalent C code for irq < 8:
!   irq_actids[hook->irq] |= hook->id;
!   outb(INT_CTLMASK, inb(INT_CTLMASK) | (1 << irq));
! Returns true iff the interrupt was not already disabled.

    .align 16
_disable_irq:
    push    ebp
    mov     ebp, esp
    pushf

```

```

    cli
    mov     eax, 8(ebp)          ! hook
    mov     ecx, 8(eax)          ! irq
    mov     eax, 12(eax)         ! id bit
    or      _irq_actids(ecx*4), eax ! set this id bit
    movb    ah, 1
    rolb    ah, cl               ! ah = (1 << (irq % 8))
    mov     edx, INT_CTLMASK     ! disable irq < 8 at the master 8259
    cmpb    cl, 8
    jnb     0f
    mov     edx, INT2_CTLMASK     ! disable irq ≥ 8 at the slave 8259
0:    inb     dx
    testb   al, ah
    jnz     dis_already          ! already disabled?
    orb     al, ah
    outb    dx                  ! set bit at the 8259
    mov     eax, 1              ! disabled by this function
    popf
    leave
    ret
dis_already:
    xor     eax, eax            ! already disabled
    popf
    leave
    ret

! *=====*
! *                               phys_copy                               *
! *=====*
! PUBLIC void phys_copy(phys_bytes source, phys_bytes destination,
!                       phys_bytes bytecount);
! Copy a block of physical memory.

PC_ARGS =      4 + 4 + 4 + 4    ! 4 + 4 + 4
!                es edi esi eip  src dst len

    .align 16
_phys_copy:
    cld
    push    esi
    push    edi
    push    es

    mov     eax, FLAT_DS_SELECTOR
    mov     es, ax

    mov     esi, PC_ARGS(esp)
    mov     edi, PC_ARGS+4(esp)
    mov     eax, PC_ARGS+4+4(esp)

    cmp     eax, 10              ! avoid align overhead for small counts
    jnb     pc_small
    mov     ecx, esi             ! align source, hope target is too
    neg     ecx
    and     ecx, 3               ! count for alignment
    sub     eax, ecx
    rep
    eseg movsb
    mov     ecx, eax
    shr     ecx, 2               ! count of dwords
    rep
    eseg movs
    and     eax, 3
pc_small:
    xchg    ecx, eax             ! remainder
    rep
    eseg movsb

    pop     es
    pop     edi
    pop     esi
    ret

```

```

!*=====*
!*                                     phys_memset                      *
!*=====*
! PUBLIC void phys_memset(phys_bytes source, unsigned long pattern,
!     phys_bytes bytecount);
! Fill a block of physical memory with pattern.

        .align 16
_phys_memset:
        push    ebp
        mov     ebp, esp
        push    esi
        push    ebx
        push    ds
        mov     esi, 8(ebp)
        mov     eax, 16(ebp)
        mov     ebx, FLAT_DS_SELECTOR
        mov     ds, bx
        mov     ebx, 12(ebp)
        shr     eax, 2
fill_start:
        mov     (esi), ebx
        add     esi, 4
        dec     eax
        jnz     fill_start
        ! Any remaining bytes?
        mov     eax, 16(ebp)
        and     eax, 3
remain_fill:
        cmp     eax, 0
        jz      fill_done
        movb    bl, 12(ebp)
        movb    (esi), bl
        add     esi, 1
        inc     ebp
        dec     eax
        jmp     remain_fill
fill_done:
        pop     ds
        pop     ebx
        pop     esi
        pop     ebp
        ret

!*=====*
!*                                     mem_rdw                        *
!*=====*
! PUBLIC u16_t mem_rdw(U16_t segment, u16_t *offset);
! Load and return word at far pointer segment:offset.

        .align 16
_mem_rdw:
        mov     cx, ds
        mov     ds, 4(esp)           ! segment
        mov     eax, 4+4(esp)       ! offset
        movzx   eax, (eax)          ! word to return
        mov     ds, cx
        ret

!*=====*
!*                                     reset                          *
!*=====*
! PUBLIC void reset();
! Reset the system by loading IDT with offset 0 and interrupting.

_reset:
        lidt    (idt_zero)
        int     3                   ! anything goes, the 386 will not like it
.sect .data
idt_zero:    .data4 0, 0
.sect .text

```

```

!*=====*
!*                                     idle_task                      *
!*=====*
_idle_task:
! This task is called when the system has nothing else to do.  The HLT
! instruction puts the processor in a state where it draws minimum power.
    push    halt
    call    _level0          ! level0(halt)
    pop     eax
    jmp     _idle_task
halt:
    sti
    hlt
    cli
    ret

!*=====*
!*                                     level0                        *
!*=====*
! PUBLIC void level0(void (*func)(void))
! Call a function at permission level 0.  This allows kernel tasks to do
! things that are only possible at the most privileged CPU level.
!
_level0:
    mov     eax, 4(esp)
    mov     (_level0_func), eax
    int     LEVEL0_VECTOR
    ret

!*=====*
!*                                     read_tsc                      *
!*=====*
! PUBLIC void read_tsc(unsigned long *high, unsigned long *low);
! Read the cycle counter of the CPU. Pentium and up.
.align 16
_read_tsc:
.data1 0x0f          ! this is the RDTSC instruction
.data1 0x31          ! it places the TSC in EDX:EAX
    push    ebp
    mov     ebp, 8(esp)
    mov     (ebp), edx
    mov     ebp, 12(esp)
    mov     (ebp), eax
    pop     ebp
    ret

!*=====*
!*                                     read_flags                    *
!*=====*
! PUBLIC unsigned long read_cpu_flags(void);
! Read CPU status flags from C.
.align 16
_read_cpu_flags:
    pushf
    mov     eax, (esp)
    popf
    ret

!*=====*
!*                                     read_cr0                      *
!*=====*
! PUBLIC unsigned long read_cr0(void);
_read_cr0:
    push    ebp
    mov     ebp, esp
    mov     eax, cr0
    pop     ebp
    ret

!*=====*
!*                                     write_cr0                     *
!*=====*

```

```
! PUBLIC void write_cr0(unsigned long value);
```

```
_write_cr0:
```

```
    push    ebp
```

```
    mov     ebp, esp
```

```
    mov     eax, 8(ebp)
```

```
    mov     cr0, eax
```

```
    jmp     0f           ! A jump is required for some flags
```

```
0:
```

```
    pop     ebp
```

```
    ret
```

```
!*=====*
```

```
!*                               write_cr3
```

```
*
```

```
!*=====*
```

```
! PUBLIC void write_cr3(unsigned long value);
```

```
_write_cr3:
```

```
    push    ebp
```

```
    mov     ebp, esp
```

```
    mov     eax, 8(ebp)
```

```
    mov     cr3, eax
```

```
    pop     ebp
```

```
    ret
```



```

/*
 * printf for the kernel
 *
 * Changes:
 *   Dec 10, 2004   kernel printing to circular buffer (Jorrit N. Herder)
 *
 * This file contains the routines that take care of kernel messages, i.e.,
 * diagnostic output within the kernel. Kernel messages are not directly
 * displayed on the console, because this must be done by the output driver.
 * Instead, the kernel accumulates characters in a buffer and notifies the
 * output driver when a new message is ready.
 */

#include "kernel.h"
#include "proc.h"
#include <signal.h>

#define printf kprintf

#include "../lib/sysutil/kprintf.c"

#define END_OF_KMESS      0
FORWARD _PROTOTYPE( void ser_putc, (char c));

/*=====
 *
 *                               kputc
 *=====*/
PUBLIC void kputc(c)
int c;
/* character to append */
{
/* Accumulate a single character for a kernel message. Send a notification
 * to the output driver if an END_OF_KMESS is encountered.
 */
    if (c != END_OF_KMESS) {
        if (do_serial_debug)
            ser_putc(c);
        kmess.km_buf[kmess.km_next] = c; /* put normal char in buffer */
        if (kmess.km_size < KMESS_BUF_SIZE)
            kmess.km_size += 1;
        kmess.km_next = (kmess.km_next + 1) % KMESS_BUF_SIZE;
    } else {
        int p, outprocs[] = OUTPUT_PROCS_ARRAY;
        for(p = 0; outprocs[p] != NONE; p++) {
            if(isokprocn(outprocs[p]) && !isemptyn(outprocs[p])) {
                send_sig(outprocs[p], SIGKMESS);
            }
        }
    }
}

#define COM1_BASE          0x3F8
#define COM1_THR            (COM1_BASE + 0)
#define LSR_THRE           0x20
#define COM1_LSR            (COM1_BASE + 5)

PRIVATE void ser_putc(char c)
{
    int i;
    int lsr, thr;

    return;

    lsr= COM1_LSR;
    thr= COM1_THR;
    for (i= 0; i<100000; i++)
    {
        if (inb(lsr) & LSR_THRE)
            break;
    }
    outb(thr, c);
}

```

```

/* 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, st_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 */
    }

/* 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;
    data_clicks = (e_hdr.a_data + e_hdr.a_bss + CLICK_SIZE-1) >> CLICK_SHIFT;
    st_clicks = (e_hdr.a_total + CLICK_SIZE-1) >> CLICK_SHIFT;
    if (!(e_hdr.a_flags & A_SEP))
    {
        data_clicks = (e_hdr.a_text + e_hdr.a_data + e_hdr.a_bss +
                       CLICK_SIZE-1) >> CLICK_SHIFT;
        text_clicks = 0; /* common I&D */
    }
    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 + st_clicks;
    rp->p_memmap[S].mem_vir = st_clicks;
    rp->p_memmap[S].mem_len = 0;

    /* 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_PRIORITY; /* 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);
#if (CHIP == INTEL)
    /* Real mode, or 16/32-bit protected mode? */
    kprintf("Executing in %s mode.\n\n",
        machine.protected ? "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.
     */
#if 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);
}
```

```
#  
! Chooses between the 8086 and 386 versions of the Minix startup code.  
  
#include <minix/config.h>  
#if _WORD_SIZE == 2  
#include "mpx88.s"  
#else  
#include "mpx386.s"  
#endif
```

```
#
! This file, mpx386.s, is included by mpx.s when Minix is compiled for
! 32-bit Intel CPUs. The alternative mpx88.s is compiled for 16-bit CPUs.

! This file is part of the lowest layer of the MINIX kernel. (The other part
! is "proc.c".) The lowest layer does process switching and message handling.
! Furthermore it contains the assembler startup code for Minix and the 32-bit
! interrupt handlers. It cooperates with the code in "start.c" to set up a
! good environment for main().

! Every transition to the kernel goes through this file. Transitions to the
! kernel may be nested. The initial entry may be with a system call (i.e.,
! send or receive a message), an exception or a hardware interrupt; kernel
! reentries may only be made by hardware interrupts. The count of reentries
! is kept in "k_reenter". It is important for deciding whether to switch to
! the kernel stack and for protecting the message passing code in "proc.c".

! For the message passing trap, most of the machine state is saved in the
! proc table. (Some of the registers need not be saved.) Then the stack is
! switched to "k_stack", and interrupts are reenabled. Finally, the system
! call handler (in C) is called. When it returns, interrupts are disabled
! again and the code falls into the restart routine, to finish off held-up
! interrupts and run the process or task whose pointer is in "proc_ptr".

! Hardware interrupt handlers do the same, except (1) The entire state must
! be saved. (2) There are too many handlers to do this inline, so the save
! routine is called. A few cycles are saved by pushing the address of the
! appropriate restart routine for a return later. (3) A stack switch is
! avoided when the stack is already switched. (4) The (master) 8259 interrupt
! controller is reenabled centrally in save(). (5) Each interrupt handler
! masks its interrupt line using the 8259 before enabling (other unmasked)
! interrupts, and unmask it after servicing the interrupt. This limits the
! nest level to the number of lines and protects the handler from itself.

! For communication with the boot monitor at startup time some constant
! data are compiled into the beginning of the text segment. This facilitates
! reading the data at the start of the boot process, since only the first
! sector of the file needs to be read.

! Some data storage is also allocated at the end of this file. This data
! will be at the start of the data segment of the kernel and will be read
! and modified by the boot monitor before the kernel starts.

! sections

.sect .text
begtext:
.sect .rom
begrom:
.sect .data
begdata:
.sect .bss
begbss:

#include <minix/config.h>
#include <minix/const.h>
#include <minix/com.h>
#include <ibm/interrupt.h>
#include "const.h"
#include "protect.h"
#include "sconst.h"

/* Selected 386 tss offsets. */
#define TSS3_S_SP0      4

! Exported functions
! Note: in assembly language the .define statement applied to a function name
! is loosely equivalent to a prototype in C code -- it makes it possible to
! link to an entity declared in the assembly code but does not create
! the entity.

.define _restart
.define save
```

```

.define _divide_error
.define _single_step_exception
.define _nmi
.define _breakpoint_exception
.define _overflow
.define _bounds_check
.define _inval_opcode
.define _copr_not_available
.define _double_fault
.define _copr_seg_overrun
.define _inval_tss
.define _segment_not_present
.define _stack_exception
.define _general_protection
.define _page_fault
.define _copr_error

.define _hwint00      ! handlers for hardware interrupts
.define _hwint01
.define _hwint02
.define _hwint03
.define _hwint04
.define _hwint05
.define _hwint06
.define _hwint07
.define _hwint08
.define _hwint09
.define _hwint10
.define _hwint11
.define _hwint12
.define _hwint13
.define _hwint14
.define _hwint15

.define _s_call
.define _p_s_call
.define _level0_call

! Exported variables.
.define begbss
.define begdata

.sect .text
!*=====*
!*                               MINIX                               *
!*=====*
MINIX:      ! this is the entry point for the MINIX kernel
            jmp      over_flags      ! skip over the next few bytes
            .data2    CLICK_SHIFT    ! for the monitor: memory granularity

flags:      .data2    0x01FD          ! boot monitor flags:
            !          call in 386 mode, make bss, make stack,
            !          load high, don't patch, will return,
            !          uses generic INT, memory vector,
            !          new boot code return
            nop          ! extra byte to sync up disassembler
over_flags:

! Set up a C stack frame on the monitor stack.  (The monitor sets cs and ds
! right.  The ss descriptor still references the monitor data segment.)
            movzx    esp, sp          ! monitor stack is a 16 bit stack
            push     ebp
            mov      ebp, esp
            push     esi
            push     edi
            cmp      4(ebp), 0        ! monitor return vector is
            jz       noret            ! nonzero if return possible
            inc      (_mon_return)
noret:      mov      (_mon_sp), esp    ! save stack pointer for later return

! Copy the monitor global descriptor table to the address space of kernel and
! switch over to it.  Prot_init() can then update it with immediate effect.

            sgdt     (_gdt+GDT_SELECTOR)    ! get the monitor gdttr

```



```

    mov     esi, (_gdt+GDT_SELECTOR+2)    ! absolute address of GDT
    mov     ebx, _gdt                    ! address of kernel GDT
    mov     ecx, 8*8                     ! copying eight descriptors
copygdt:
    eseg   movb    al, (esi)
    movb   (ebx), al
    inc    esi
    inc    ebx
    loop   copygdt
    mov    eax, (_gdt+DS_SELECTOR+2)      ! base of kernel data
    and    eax, 0x00FFFFFF                ! only 24 bits
    add    eax, _gdt                      ! eax = vir2phys(gdt)
    mov    (_gdt+GDT_SELECTOR+2), eax     ! set base of GDT
    lgdt   (_gdt+GDT_SELECTOR)           ! switch over to kernel GDT

! Locate boot parameters, set up kernel segment registers and stack.
    mov    ebx, 8(ebp)                    ! boot parameters offset
    mov    edx, 12(ebp)                   ! boot parameters length
    mov    eax, 16(ebp)                   ! address of a.out headers
    mov    (_aout), eax
    mov    ax, ds                         ! kernel data
    mov    es, ax
    mov    fs, ax
    mov    gs, ax
    mov    ss, ax
    mov    esp, k_stktop                  ! set sp to point to the top of kernel stack

! Call C startup code to set up a proper environment to run main().
    push   edx
    push   ebx
    push   SS_SELECTOR
    push   DS_SELECTOR
    push   CS_SELECTOR
    call   _cstart                        ! cstart(cs, ds, mds, parmoff, parmlen)
    add    esp, 5*4

! Reload gdtr, idtr and the segment registers to global descriptor table set
! up by prot_init().

    lgdt   (_gdt+GDT_SELECTOR)
    lidt   (_gdt+IDT_SELECTOR)

    jmpf   CS_SELECTOR:csinit
csinit:
    o16    mov    ax, DS_SELECTOR
    mov    ds, ax
    mov    es, ax
    mov    fs, ax
    mov    gs, ax
    mov    ss, ax
    o16    mov    ax, TSS_SELECTOR         ! no other TSS is used
    ltr    ax
    push   0                             ! set flags to known good state
    popf                                       ! esp, clear nested task and int enable

    jmp    _main                          ! main()

!*=====*
!*                               interrupt handlers                               *
!*                               interrupt handlers for 386 32-bit protected mode    *
!*=====*

!*=====*
!*                               hwint00 - 07                                     *
!*=====*
! Note this is a macro, it just looks like a subroutine.
#define hwint_master(irq) \
    call    save                        /* save interrupted process state */;\
    push    (_irq_handlers+4*irq)      /* irq_handlers[irq] */;\
    call    _intr_handle               /* intr_handle(irq_handlers[irq]) */;\
    pop     ecx                        ;\
    cmp     (_irq_actids+4*irq), 0     /* interrupt still active? */;\
    jz      0f                        ;\

```

```

    inb     INT_CTLMASK          /* get current mask */           ;\
    orb     al, [1<<irq]        /* mask irq */                     ;\
    outb    INT_CTLMASK         /* disable the irq */                */;\
0:  movb    al, END_OF_INT       ;\
    outb    INT_CTL             /* reenable master 8259 */          */;\
    ret                                /* restart (another) process */

```

! Each of these entry points is an expansion of the hwint\_master macro

```

    .align 16
_hwint00:      ! Interrupt routine for irq 0 (the clock).
    hwint_master(0)

    .align 16
_hwint01:      ! Interrupt routine for irq 1 (keyboard)
    hwint_master(1)

    .align 16
_hwint02:      ! Interrupt routine for irq 2 (cascade!)
    hwint_master(2)

    .align 16
_hwint03:      ! Interrupt routine for irq 3 (second serial)
    hwint_master(3)

    .align 16
_hwint04:      ! Interrupt routine for irq 4 (first serial)
    hwint_master(4)

    .align 16
_hwint05:      ! Interrupt routine for irq 5 (XT winchester)
    hwint_master(5)

    .align 16
_hwint06:      ! Interrupt routine for irq 6 (floppy)
    hwint_master(6)

    .align 16
_hwint07:      ! Interrupt routine for irq 7 (printer)
    hwint_master(7)

```

! \*=====\*

! \* hwint08 - 15

! \*=====\*

! Note this is a macro, it just looks like a subroutine.

```

#define hwint_slave(irq) \
    call    save                /* save interrupted process state */;\
    push    (_irq_handlers+4*irq) /* irq_handlers[irq] */;\
    call    _intr_handle        /* intr_handle(irq_handlers[irq]) */;\
    pop     ecx                 ;\
    cmp     (_irq_actids+4*irq), 0 /* interrupt still active? */;\
    jz      0f                 ;\
    inb     INT2_CTLMASK        ;\
    orb     al, [1<<[irq-8]]    ;\
    outb    INT2_CTLMASK         /* disable the irq */;\
0:  movb    al, END_OF_INT       ;\
    outb    INT_CTL             /* reenable master 8259 */;\
    outb    INT2_CTL            /* reenable slave 8259 */;\
    ret                                /* restart (another) process */

```

! Each of these entry points is an expansion of the hwint\_slave macro

```

    .align 16
_hwint08:      ! Interrupt routine for irq 8 (realtime clock)
    hwint_slave(8)

    .align 16
_hwint09:      ! Interrupt routine for irq 9 (irq 2 redirected)
    hwint_slave(9)

    .align 16
_hwint10:      ! Interrupt routine for irq 10
    hwint_slave(10)

    .align 16
_hwint11:      ! Interrupt routine for irq 11

```

```

    hwint_slave(11)

    .align 16
_hwint12:    ! Interrupt routine for irq 12
    hwint_slave(12)

    .align 16
_hwint13:    ! Interrupt routine for irq 13 (FPU exception)
    hwint_slave(13)

    .align 16
_hwint14:    ! Interrupt routine for irq 14 (AT winchester)
    hwint_slave(14)

    .align 16
_hwint15:    ! Interrupt routine for irq 15
    hwint_slave(15)

!=====
!*                                     save                                     *
!=====
! Save for protected mode.
! This is much simpler than for 8086 mode, because the stack already points
! into the process table, or has already been switched to the kernel stack.

    .align 16
save:
    cld                                ! set direction flag to a known value
    pushad                            ! save "general" registers
    o16 push    ds                    ! save ds
    o16 push    es                    ! save es
    o16 push    fs                    ! save fs
    o16 push    gs                    ! save gs
    mov     dx, ss                    ! ss is kernel data segment
    mov     ds, dx                    ! load rest of kernel segments
    mov     es, dx                    ! kernel does not use fs, gs
    mov     eax, esp                  ! prepare to return
    incb    (_k_reenter)              ! from -1 if not reentering
    jnz     set_restart1              ! stack is already kernel stack
    mov     esp, k_stktop
    push    _restart                  ! build return address for int handler
    xor     ebp, ebp                  ! for stacktrace
    jmp     RETADR-P_STACKBASE(eax)

    .align 4
set_restart1:
    push    restart1
    jmp     RETADR-P_STACKBASE(eax)

!=====
!*                                     _s_call                               *
!=====
    .align 16
_s_call:
_p_s_call:
    cld                                ! set direction flag to a known value
    sub     esp, 6*4                  ! skip RETADR, eax, ecx, edx, ebx, esi
    push    ebp                      ! stack already points into proc table
    push    esi
    push    edi
    o16 push    ds
    o16 push    es
    o16 push    fs
    o16 push    gs
    mov     si, ss                    ! ss is kernel data segment
    mov     ds, si                    ! load rest of kernel segments
    mov     es, si                    ! kernel does not use fs, gs
    incb    (_k_reenter)              ! increment kernel entry count
    mov     esi, esp                  ! assumes P_STACKBASE == 0
    mov     esp, k_stktop
    xor     ebp, ebp                  ! for stacktrace
    ! end of inline save
    ! now set up parameters for sys_call()
    push    edx                      ! event set or flags bit map

```

```

    push    ebx                ! pointer to user message
    push    eax                ! source / destination
    push    ecx                ! call number (ipc primitive to use)
    call    _sys_call          ! sys_call(call_nr, src_dst, m_ptr, bit_map)
                                ! caller is now explicitly in proc_ptr
    mov     AXREG(esi), eax    ! sys_call MUST PRESERVE si

! Fall into code to restart proc/task running.

!*=====*
!*                                restart                                *
!*=====*
_restart:

! Restart the current process or the next process if it is set.

    cmp     (_next_ptr), 0      ! see if another process is scheduled
    jz      0f
    mov     eax, (_next_ptr)
    mov     (_proc_ptr), eax    ! schedule new process
    mov     (_next_ptr), 0
0:    mov     esp, (_proc_ptr)    ! will assume P_STACKBASE = 0
    lldt    P_LDT_SEL(esp)      ! enable process' segment descriptors
    lea     eax, P_STACKTOP(esp) ! arrange for next interrupt
    mov     (_tss+TSS3_S_SP0), eax ! to save state in process table
restart1:
    decb    (_k_reenter)
    o16 pop    gs
    o16 pop    fs
    o16 pop    es
    o16 pop    ds
    popad
    add     esp, 4              ! skip return adr
    iretd                      ! continue process

!*=====*
!*                                exception handlers                        *
!*=====*
_divide_error:
    push    DIVIDE_VECTOR
    jmp     exception

_single_step_exception:
    push    DEBUG_VECTOR
    jmp     exception

_nmi:
    push    NMI_VECTOR
    jmp     exception

_breakpoint_exception:
    push    BREAKPOINT_VECTOR
    jmp     exception

_overflow:
    push    OVERFLOW_VECTOR
    jmp     exception

_bounds_check:
    push    BOUNDS_VECTOR
    jmp     exception

_inval_opcode:
    push    INVALID_OP_VECTOR
    jmp     exception

_copr_not_available:
    push    COPROC_NOT_VECTOR
    jmp     exception

_double_fault:
    push    DOUBLE_FAULT_VECTOR
    jmp     errexception

```

```

_copr_seg_overrun:
    push    COPROC_SEG_VECTOR
    jmp     exception

_inval_tss:
    push    INVALID_TSS_VECTOR
    jmp     errexception

_segment_not_present:
    push    SEG_NOT_VECTOR
    jmp     errexception

_stack_exception:
    push    STACK_FAULT_VECTOR
    jmp     errexception

_general_protection:
    push    PROTECTION_VECTOR
    jmp     errexception

_page_fault:
    push    PAGE_FAULT_VECTOR
    jmp     errexception

_copr_error:
    push    COPROC_ERR_VECTOR
    jmp     exception

! *=====*
! *                               exception                               *
! *=====*
! This is called for all exceptions which do not push an error code.

        .align 16
exception:
    sseg    mov     (trap_errno), 0          ! clear trap_errno
    sseg    pop     (ex_number)
    jmp     exceptionl

! *=====*
! *                               errexception                               *
! *=====*
! This is called for all exceptions which push an error code.

        .align 16
errexception:
    sseg    pop     (ex_number)
    sseg    pop     (trap_errno)
exceptionl:
    push    eax                                ! Common for all exceptions.
    mov     eax, 0+4(esp)                      ! eax is scratch register
    sseg    mov     (old_eip), eax
    movzx   eax, 4+4(esp)                      ! old cs
    sseg    mov     (old_cs), eax
    mov     eax, 8+4(esp)                      ! old eflags
    sseg    mov     (old_eflags), eax
    pop     eax
    call    save
    push    (old_eflags)
    push    (old_cs)
    push    (old_eip)
    push    (trap_errno)
    push    (ex_number)
    call    _exception                        ! (ex_number, trap_errno, old_eip,
    !                                             old_cs, old_eflags)
    add     esp, 5*4
    ret

! *=====*
! *                               level0_call                               *
! *=====*
_level0_call:
    call    save
    jmp     (_level0_func)

```

```
!*=====*
!*                               data                               *
!*=====*

.sect .rom      ! Before the string table please
               .data2 0x526F      ! this must be the first data entry (magic #)

.sect .bss
k_stack:
               .space  K_STACK_BYTES ! kernel stack
k_stktop:      ! top of kernel stack
               .comm   ex_number, 4
               .comm   trap_errno, 4
               .comm   old_eip, 4
               .comm   old_cs, 4
               .comm   old_eflags, 4
```

```

#ifndef PRIV_H
#define PRIV_H

/* Declaration of the system privileges structure. It defines flags, system
 * call masks, an synchronous alarm timer, I/O privileges, pending hardware
 * interrupts and notifications, and so on.
 * System processes each get their own structure with properties, whereas all
 * user processes share one structure. This setup provides a clear separation
 * between common and privileged process fields and is very space efficient.
 *
 * Changes:
 *   Jul 01, 2005      Created. (Jorrit N. Herder)
 */
#include <minix/com.h>
#include "protect.h"
#include "const.h"
#include "type.h"

/* Max. number of I/O ranges that can be assigned to a process */
#define NR_IO_RANGE      10

/* Max. number of device memory ranges that can be assigned to a process */
#define NR_MEM_RANGE     10

/* Max. number of IRQs that can be assigned to a process */
#define NR_IRQ           4

struct priv {
    proc_nr_t s_proc_nr;           /* number of associated process */
    sys_id_t s_id;                 /* index of this system structure */
    short s_flags;                 /* PREEMTIBLE, BILLABLE, etc. */

    short s_trap_mask;             /* allowed system call traps */
    sys_map_t s_ipc_from;          /* allowed callers to receive from */
    sys_map_t s_ipc_to;            /* allowed destination processes */
    long s_call_mask;              /* allowed kernel calls */

    sys_map_t s_notify_pending;    /* bit map with pending notifications */
    irq_id_t s_int_pending;        /* pending hardware interrupts */
    sigset_t s_sig_pending;        /* pending signals */

    timer_t s_alarm_timer;         /* synchronous alarm timer */
    struct far_mem s_farmem[NR_REMOTE_SEGS]; /* remote memory map */
    reg_t *s_stack_guard;          /* stack guard word for kernel tasks */

    int s_nr_io_range;             /* allowed I/O ports */
    struct io_range s_io_tab[NR_IO_RANGE];

    int s_nr_mem_range;            /* allowed memory ranges */
    struct mem_range s_mem_tab[NR_MEM_RANGE];

    int s_nr_irq;                  /* allowed IRQ lines */
    int s_irq_tab[NR_IRQ];
};

/* Guard word for task stacks. */
#define STACK_GUARD      ((reg_t) (sizeof(reg_t) == 2 ? 0xBEEF : 0xDEADBEEF))

/* Bits for the system property flags. */
#define PREEMPTIBLE      0x02    /* kernel tasks are not preemptible */
#define BILLABLE        0x04    /* some processes are not billable */

#define SYS_PROC         0x10    /* system processes have own priv structure */
#define CHECK_IO_PORT    0x20    /* check if I/O request is allowed */
#define CHECK_IRQ        0x40    /* check if IRQ can be used */
#define CHECK_MEM        0x80    /* check if (VM) mem map request is allowed */

/* Magic system structure table addresses. */
#define BEG_PRIV_ADDR    (&priv[0])
#define END_PRIV_ADDR    (&priv[NR_SYS_PROCS])

#define priv_addr(i)      (ppriv_addr)[(i)]
#define priv_id(rp)       ((rp)->p_priv->s_id)
#define priv(rp)          ((rp)->p_priv)

```

```
#define id_to_nr(id)      priv_addr(id)->s_proc_nr
#define nr_to_id(nr)      priv(proc_addr(nr))->s_id

/* The system structures table and pointers to individual table slots. The
 * pointers allow faster access because now a process entry can be found by
 * indexing the psys_addr array, while accessing an element i requires a
 * multiplication with sizeof(struct sys) to determine the address.
 */
EXTERN struct priv priv[NR_SYS_PROCS];          /* system properties table */
EXTERN struct priv *ppriv_addr[NR_SYS_PROCS];    /* direct slot pointers */

/* Unprivileged user processes all share the same privilege structure.
 * This id must be fixed because it is used to check send mask entries.
 */
#define USER_PRIV_ID      0

/* Make sure the system can boot. The following sanity check verifies that
 * the system privileges table is large enough for the number of processes
 * in the boot image.
 */
#if (NR_BOOT_PROCS > NR_SYS_PROCS)
#error NR_SYS_PROCS must be larger than NR_BOOT_PROCS
#endif

#endif /* PRIV_H */
```



```

/* This file contains essentially all of the process and message handling.
 * Together with "mpx.s" it forms the lowest layer of the MINIX kernel.
 * There is one entry point from the outside:
 *
 * sys_call:      a system call, i.e., the kernel is trapped with an INT
 *
 * As well as several entry points used from the interrupt and task level:
 *
 * lock_notify:   notify a process of a system event
 * lock_send:     send a message to a process
 * lock_enqueue:  put a process on one of the scheduling queues
 * lock_dequeue:  remove a process from the scheduling queues
 *
 * Changes:
 * Aug 19, 2005    rewrote scheduling code (Jorrit N. Herder)
 * Jul 25, 2005    rewrote system call handling (Jorrit N. Herder)
 * May 26, 2005    rewrote message passing functions (Jorrit N. Herder)
 * May 24, 2005    new notification system call (Jorrit N. Herder)
 * Oct 28, 2004    nonblocking send and receive calls (Jorrit N. Herder)
 *
 * The code here is critical to make everything work and is important for the
 * overall performance of the system. A large fraction of the code deals with
 * list manipulation. To make this both easy to understand and fast to execute
 * pointer pointers are used throughout the code. Pointer pointers prevent
 * exceptions for the head or tail of a linked list.
 *
 * node_t *queue, *new_node;    // assume these as global variables
 * node_t **xpp = &queue;      // get pointer pointer to head of queue
 * while (*xpp != NULL)         // find last pointer of the linked list
 *     xpp = &(*xpp)->next;     // get pointer to next pointer
 * *xpp = new_node;             // now replace the end (the NULL pointer)
 * new_node->next = NULL;        // and mark the new end of the list
 *
 * For example, when adding a new node to the end of the list, one normally
 * makes an exception for an empty list and looks up the end of the list for
 * nonempty lists. As shown above, this is not required with pointer pointers.
 */

```

```

#include <minix/com.h>
#include <minix/callnr.h>
#include <minix/endpoint.h>
#include "debug.h"
#include "kernel.h"
#include "proc.h"
#include <signal.h>

```

```

/* Scheduling and message passing functions. The functions are available to
 * other parts of the kernel through lock_...(). The lock temporarily disables
 * interrupts to prevent race conditions.
 */

```

```

FORWARD _PROTOTYPE( int mini_send, (struct proc *caller_ptr, int dst_e,
                                message *m_ptr, unsigned flags));
FORWARD _PROTOTYPE( int mini_receive, (struct proc *caller_ptr, int src,
                                message *m_ptr, unsigned flags));
FORWARD _PROTOTYPE( int mini_notify, (struct proc *caller_ptr, int dst));
FORWARD _PROTOTYPE( int deadlock, (int function,
                                register struct proc *caller, int src_dst));
FORWARD _PROTOTYPE( void enqueue, (struct proc *rp));
FORWARD _PROTOTYPE( void dequeue, (struct proc *rp));
FORWARD _PROTOTYPE( void sched, (struct proc *rp, int *queue, int *front));
FORWARD _PROTOTYPE( void pick_proc, (void));

```

```

#define BuildMess(m_ptr, src, dst_ptr) \
    (m_ptr)->m_source = proc_addr(src)->p_endpoint; \
    (m_ptr)->m_type = NOTIFY_FROM(src); \
    (m_ptr)->NOTIFY_TIMESTAMP = get_uptime(); \
    switch (src) { \
    case HARDWARE: \
        (m_ptr)->NOTIFY_ARG = priv(dst_ptr)->s_int_pending; \
        priv(dst_ptr)->s_int_pending = 0; \
        break; \
    case SYSTEM: \
        (m_ptr)->NOTIFY_ARG = priv(dst_ptr)->s_sig_pending; \
        priv(dst_ptr)->s_sig_pending = 0; \
    }

```

```

        break;
    }

#if (CHIP == INTEL)
#define CopyMess(s, sp, sm, dp, dm) \
    cp_mess(proc_addr(s)->p_endpoint, \
            (sp)->p_memmap[D].mem_phys, \
            (vir_bytes)sm, (dp)->p_memmap[D].mem_phys, (vir_bytes)dm)
#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)
/* M68000 does not have cp_mess() in assembly like INTEL. Declare prototype
 * for cp_mess() here and define the function below. Also define CopyMess.
 */
#endif /* (CHIP == M68000) */

/*=====
 *                               sys_call                               *
 *=====*/
PUBLIC int sys_call(call_nr, src_dst_e, m_ptr, bit_map)
int call_nr;                /* system call number and flags */
int src_dst_e;              /* src to receive from or dst to send to */
message *m_ptr;             /* pointer to message in the caller's space */
long bit_map;              /* notification event set or flags */
{
/* System calls are done by trapping to the kernel with an INT instruction.
 * The trap is caught and sys_call() is called to send or receive a message
 * (or both). The caller is always given by 'proc_ptr'.
 */
    register struct proc *caller_ptr = proc_ptr; /* get pointer to caller */
    int function = call_nr & SYSCALL_FUNC;      /* get system call function */
    unsigned flags = call_nr & SYSCALL_FLAGS;    /* get flags */
    int mask_entry;                             /* bit to check in send mask */
    int group_size;                             /* used for deadlock check */
    int result;                                  /* the system call's result */
    int src_dst;
    vir_clicks vlo, vhi;                       /* virtual clicks containing message to send */

#if 0
    if (caller_ptr->p_rts_flags & SLOT_FREE)
    {
        kprintf("called by the dead?!?\n");
        return EINVAL;
    }
#endif

    /* Require a valid source and/ or destination process, unless echoing. */
    if (src_dst_e != ANY && function != ECHO) {
        if (!isokendpt(src_dst_e, &src_dst)) {
#if DEBUG_ENABLE_IPC_WARNINGS
            kprintf("sys_call: trap %d by %d with bad endpoint %d\n",
                    function, proc_nr(caller_ptr), src_dst_e);
#endif

            return EDEADSRCDST;
        }
    } else src_dst = src_dst_e;

    /* Check if the process has privileges for the requested call. Calls to the
     * kernel may only be SENDREC, because tasks always reply and may not block
     * if the caller doesn't do receive().
     */
    if (! (priv(caller_ptr)->s_trap_mask & (1 << function)) ||
        (iskerneln(src_dst) && function != SENDREC
         && function != RECEIVE)) {
#if DEBUG_ENABLE_IPC_WARNINGS
        kprintf("sys_call: trap %d not allowed, caller %d, src_dst %d\n",
                function, proc_nr(caller_ptr), src_dst);
#endif

        return(ETRAPDENIED); /* trap denied by mask or kernel */
    }

    /* If the call involves a message buffer, i.e., for SEND, RECEIVE, SENDREC,
     * or ECHO, check the message pointer. This check allows a message to be
     * anywhere in data or stack or gap. It will have to be made more elaborate

```

```

    * for machines which don't have the gap mapped.
    */
    if (function & CHECK_PTR) {
        vlo = (vir_bytes) m_ptr >> CLICK_SHIFT;
        vhi = ((vir_bytes) m_ptr + MESS_SIZE - 1) >> CLICK_SHIFT;
        if (vlo < caller_ptr->p_memmap[D].mem_vir || vlo > vhi ||
            vhi >= caller_ptr->p_memmap[S].mem_vir +
            caller_ptr->p_memmap[S].mem_len) {
#if DEBUG_ENABLE_IPC_WARNINGS
            kprintf("sys_call: invalid message pointer, trap %d, caller %d\n",
                    function, proc_nr(caller_ptr));
#endif
            return(EFAULT);          /* invalid message pointer */
        }

        /* If the call is to send to a process, i.e., for SEND, SENDREC or NOTIFY,
        * verify that the caller is allowed to send to the given destination.
        */
        if (function & CHECK_DST) {
            if (!get_sys_bit(priv(caller_ptr)->s_ipc_to, nr_to_id(src_dst))) {
#if DEBUG_ENABLE_IPC_WARNINGS
                kprintf("sys_call: ipc mask denied trap %d from %d to %d\n",
                        function, proc_nr(caller_ptr), src_dst);
#endif
                return(ECALLDENIED); /* call denied by ipc mask */
            }
        }

        /* Check for a possible deadlock for blocking SEND(REC) and RECEIVE. */
        if (function & CHECK_DEADLOCK) {
            if (group_size = deadlock(function, caller_ptr, src_dst)) {
#if DEBUG_ENABLE_IPC_WARNINGS
                kprintf("sys_call: trap %d from %d to %d deadlocked, group size %d\n",
                        function, proc_nr(caller_ptr), src_dst, group_size);
#endif
                return(ELOCKED);
            }
        }

        /* Now check if the call is known and try to perform the request. The only
        * system calls that exist in MINIX are sending and receiving messages.
        * - SENDREC: combines SEND and RECEIVE in a single system call
        * - SEND: sender blocks until its message has been delivered
        * - RECEIVE: receiver blocks until an acceptable message has arrived
        * - NOTIFY: nonblocking call; deliver notification or mark pending
        * - ECHO: nonblocking call; directly echo back the message
        */
        switch(function) {
        case SENDREC:
            /* A flag is set so that notifications cannot interrupt SENDREC. */
            caller_ptr->p_misc_flags |= REPLY_PENDING;
            /* fall through */
        case SEND:
            result = mini_send(caller_ptr, src_dst_e, m_ptr, flags);
            if (function == SEND || result != OK) {
                break; /* done, or SEND failed */
            } /* fall through for SENDREC */
        case RECEIVE:
            if (function == RECEIVE)
                caller_ptr->p_misc_flags &= ~REPLY_PENDING;
            result = mini_receive(caller_ptr, src_dst_e, m_ptr, flags);
            break;
        case NOTIFY:
            result = mini_notify(caller_ptr, src_dst);
            break;
        case ECHO:
            CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, caller_ptr, m_ptr);
            result = OK;
            break;
        default:
            result = EBADCALL; /* illegal system call */
        }
    }

```

```

/* Now, return the result of the system call to the caller. */
return(result);
}

/*=====
*
*                                deadlock
*=====*/
PRIVATE int deadlock(function, cp, src_dst)
int function;                                /* trap number */
register struct proc *cp;                    /* pointer to caller */
int src_dst;                                /* src or dst process */
{
/* Check for deadlock. This can happen if 'caller_ptr' and 'src_dst' have
* a cyclic dependency of blocking send and receive calls. The only cyclic
* dependency that is not fatal is if the caller and target directly SEND(REC)
* and RECEIVE to each other. If a deadlock is found, the group size is
* returned. Otherwise zero is returned.
*/
register struct proc *xp;                    /* process pointer */
int group_size = 1;                          /* start with only caller */
int trap_flags;

while (src_dst != ANY) {                    /* check while process nr */
    int src_dst_e;
    xp = proc_addr(src_dst);                /* follow chain of processes */
    group_size++;                           /* extra process in group */

/* Check whether the last process in the chain has a dependency. If it
* has not, the cycle cannot be closed and we are done.
*/
if (xp->p_rts_flags & RECEIVING) {          /* xp has dependency */
    if(xp->p_getfrom_e == ANY) src_dst = ANY;
    else okendpt(xp->p_getfrom_e, &src_dst);
} else if (xp->p_rts_flags & SENDING) {      /* xp has dependency */
    okendpt(xp->p_sendto_e, &src_dst);
} else {
    return(0);                             /* not a deadlock */
}

/* Now check if there is a cyclic dependency. For group sizes of two,
* a combination of SEND(REC) and RECEIVE is not fatal. Larger groups
* or other combinations indicate a deadlock.
*/
if (src_dst == proc_nr(cp)) {              /* possible deadlock */
    if (group_size == 2) {                  /* caller and src_dst */
        /* The function number is magically converted to flags. */
        if ((xp->p_rts_flags ^ (function < 2)) & SENDING) {
            return(0);                     /* not a deadlock */
        }
    }
    return(group_size);                    /* deadlock found */
}
}
return(0);                                /* not a deadlock */
}

/*=====
*
*                                mini_send
*=====*/
PRIVATE int mini_send(caller_ptr, dst_e, m_ptr, flags)
register struct proc *caller_ptr;            /* who is trying to send a message? */
int dst_e;                                  /* to whom is message being sent? */
message *m_ptr;                             /* pointer to message buffer */
unsigned flags;                             /* system call flags */
{
/* Send a message from 'caller_ptr' to 'dst'. If 'dst' is blocked waiting
* for this message, copy the message to it and unblock 'dst'. If 'dst' is
* not waiting at all, or is waiting for another source, queue 'caller_ptr'.
*/
register struct proc *dst_ptr;
register struct proc **xpp;
int dst_p;

dst_p = _ENDPOINT_P(dst_e);

```

```

dst_ptr = proc_addr(dst_p);

if (dst_ptr->p_rts_flags & NO_ENDPOINT) return EDSTDIED;

/* Check if 'dst' is blocked waiting for this message. The destination's
 * SENDING flag may be set when its SENDREC call blocked while sending.
 */
if ( (dst_ptr->p_rts_flags & (RECEIVING | SENDING)) == RECEIVING &&
     (dst_ptr->p_getfrom_e == ANY
      || dst_ptr->p_getfrom_e == caller_ptr->p_endpoint)) {
    /* Destination is indeed waiting for this message. */
    CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, dst_ptr,
             dst_ptr->p_messbuf);
    if ((dst_ptr->p_rts_flags & ~RECEIVING) == 0) enqueue(dst_ptr);
} else if ( ! (flags & NON_BLOCKING)) {
    /* Destination is not waiting. Block and dequeue caller. */
    caller_ptr->p_messbuf = m_ptr;
    if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
    caller_ptr->p_rts_flags |= SENDING;
    caller_ptr->p_sendto_e = dst_e;

    /* Process is now blocked. Put in on the destination's queue. */
    xpp = &dst_ptr->p_caller_q; /* find end of list */
    while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;
    *xpp = caller_ptr; /* add caller to end */
    caller_ptr->p_q_link = NIL_PROC; /* mark new end of list */
} else {
    return(ENOTREADY);
}
return(OK);
}

/*=====
 * mini_receive
 *=====*/
PRIVATE int mini_receive(caller_ptr, src_e, m_ptr, flags)
register struct proc *caller_ptr; /* process trying to get message */
int src_e; /* which message source is wanted */
message *m_ptr; /* pointer to message buffer */
unsigned flags; /* system call flags */
{
    /* A process or task wants to get a message. If a message is already queued,
     * acquire it and deblock the sender. If no message from the desired source
     * is available block the caller, unless the flags don't allow blocking.
     */
    register struct proc **xpp;
    register struct notification **ntf_q_pp;
    message m;
    int bit_nr;
    sys_map_t *map;
    bitchunk_t *chunk;
    int i, src_id, src_proc_nr, src_p;

    if(src_e == ANY) src_p = ANY;
    else
    {
        okendpt(src_e, &src_p);
        if (proc_addr(src_p)->p_rts_flags & NO_ENDPOINT) return ESRCDIED;
    }

    /* Check to see if a message from desired source is already available.
     * The caller's SENDING flag may be set if SENDREC couldn't send. If it is
     * set, the process should be blocked.
     */
    if (!(caller_ptr->p_rts_flags & SENDING)) {

        /* Check if there are pending notifications, except for SENDREC. */
        if (!(caller_ptr->p_misc_flags & REPLY_PENDING)) {

            map = &priv(caller_ptr)->s_notify_pending;
            for (chunk=&map->chunk[0]; chunk<&map->chunk[NR_SYS_CHUNKS]; chunk++) {

                /* Find a pending notification from the requested source. */

```

```

        if (! *chunk) continue; /* no bits in chunk */
        for (i=0; ! (*chunk & (1<<i)); ++i) {} /* look up the bit */
        src_id = (chunk - &map->chunk[0]) * BITCHUNK_BITS + i;
        if (src_id >= NR_SYS_PROCS) break; /* out of range */
        src_proc_nr = id_to_nr(src_id); /* get source proc */
#if DEBUG_ENABLE_IPC_WARNINGS
        if (src_proc_nr == NONE) {
            kprintf("mini_receive: sending notify from NONE\n");
        }
#endif

        if (src_e!=ANY && src_p != src_proc_nr) continue; /* source not ok */
        *chunk &= ~(1 << i); /* no longer pending */

        /* Found a suitable source, deliver the notification message. */
        BuildMess(&m, src_proc_nr, caller_ptr); /* assemble message */
        CopyMess(src_proc_nr, proc_addr(HARDWARE), &m, caller_ptr, m_ptr);
        return(OK); /* report success */
    }
}

/* Check caller queue. Use pointer pointers to keep code simple. */
xpp = &caller_ptr->p_caller_q;
while (*xpp != NIL_PROC) {
    if (src_e == ANY || src_p == proc_nr(*xpp)) {
#if 0
        if ((*xpp)->p_rts_flags & SLOT_FREE)
        {
            kprintf("listening to the dead?!?\n");
            return EINVAL;
        }
#endif

        /* Found acceptable message. Copy it and update status. */
        CopyMess((*xpp)->p_nr, *xpp, (*xpp)->p_messbuf, caller_ptr, m_ptr);
        if ((*xpp)->p_rts_flags & ~SENDING) == 0) enqueue(*xpp);
        *xpp = (*xpp)->p_q_link; /* remove from queue */
        return(OK); /* report success */
    }
    xpp = &(*xpp)->p_q_link; /* proceed to next */
}

/* No suitable message is available or the caller couldn't send in SENDREC.
 * Block the process trying to receive, unless the flags tell otherwise.
 */
if ( ! (flags & NON_BLOCKING)) {
    caller_ptr->p_getfrom_e = src_e;
    caller_ptr->p_messbuf = m_ptr;
    if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
    caller_ptr->p_rts_flags |= RECEIVING;
    return(OK);
} else {
    return(ENOTREADY);
}
}

/*=====
 * mini_notify
 *=====*/
PRIVATE int mini_notify(caller_ptr, dst)
register struct proc *caller_ptr; /* sender of the notification */
int dst; /* which process to notify */
{
    register struct proc *dst_ptr = proc_addr(dst);
    int src_id; /* source id for late delivery */
    message m; /* the notification message */

    /* Check to see if target is blocked waiting for this message. A process
     * can be both sending and receiving during a SENDREC system call.
     */
    if ((dst_ptr->p_rts_flags & (RECEIVING|SENDING)) == RECEIVING &&
        ! (dst_ptr->p_misc_flags & REPLY_PENDING) &&
        (dst_ptr->p_getfrom_e == ANY ||
         dst_ptr->p_getfrom_e == caller_ptr->p_endpoint)) {

```

```

    /* Destination is indeed waiting for a message. Assemble a notification
    * message and deliver it. Copy from pseudo-source HARDWARE, since the
    * message is in the kernel's address space.
    */
    BuildMess(&m, proc_nr(caller_ptr), dst_ptr);
    CopyMess(proc_nr(caller_ptr), proc_addr(HARDWARE), &m,
        dst_ptr, dst_ptr->p_messbuf);
    dst_ptr->p_rts_flags &= ~RECEIVING; /* deblock destination */
    if (dst_ptr->p_rts_flags == 0) enqueue(dst_ptr);
    return(OK);
}

/* Destination is not ready to receive the notification. Add it to the
* bit map with pending notifications. Note the indirectness: the system id
* instead of the process number is used in the pending bit map.
*/
src_id = priv(caller_ptr)->s_id;
set_sys_bit(priv(dst_ptr)->s_notify_pending, src_id);
return(OK);
}

/*=====
*                               lock_notify                               *
*=====*/
PUBLIC int lock_notify(src_e, dst_e)
int src_e; /* (endpoint) sender of the notification */
int dst_e; /* (endpoint) who is to be notified */
{
    /* Safe gateway to mini_notify() for tasks and interrupt handlers. The sender
    * is explicitly given to prevent confusion where the call comes from. MINIX
    * kernel is not reentrant, which means interrupts are disabled after
    * the first kernel entry (hardware interrupt, trap, or exception). Locking
    * is done by temporarily disabling interrupts.
    */
    int result, src, dst;

    if(!isokendpt(src_e, &src) || !isokendpt(dst_e, &dst))
        return EDEADSRCDST;

    /* Exception or interrupt occurred, thus already locked. */
    if (k_reenter >= 0) {
        result = mini_notify(proc_addr(src), dst);
    }

    /* Call from task level, locking is required. */
    else {
        lock(0, "notify");
        result = mini_notify(proc_addr(src), dst);
        unlock(0);
    }
    return(result);
}

/*=====
*                               enqueue                               *
*=====*/
PRIVATE void enqueue(rp)
register struct proc *rp; /* this process is now runnable */
{
    /* Add 'rp' to one of the queues of runnable processes. This function is
    * responsible for inserting a process into one of the scheduling queues.
    * The mechanism is implemented here. The actual scheduling policy is
    * defined in sched() and pick_proc().
    */
    int q; /* scheduling queue to use */
    int front; /* add to front or back */

#ifdef DEBUG_SCHED_CHECK
    check_runqueues("enqueue");
    if (rp->p_ready) kprintf("enqueue() already ready process\n");
#endif

    /* Determine where to insert to process. */

```

```

sched(rp, &q, &front);

/* Now add the process to the queue. */
if (rdy_head[q] == NIL_PROC) {
    rdy_head[q] = rdy_tail[q] = rp;
    rp->p_nextready = NIL_PROC;
}
else if (front) {
    rp->p_nextready = rdy_head[q];
    rdy_head[q] = rp;
}
else {
    rdy_tail[q]->p_nextready = rp;
    rdy_tail[q] = rp;
    rp->p_nextready = NIL_PROC;
}

/* add to empty queue */
/* create a new queue */
/* mark new end */

/* add to head of queue */
/* chain head of queue */
/* set new queue head */

/* add to tail of queue */
/* chain tail of queue */
/* set new queue tail */
/* mark new end */

/* Now select the next process to run. */
pick_proc();

#if DEBUG_SCHED_CHECK
    rp->p_ready = 1;
    check_runqueues("enqueue");
#endif
}

/*=====
*                               dequeue                               *
*=====*/
PRIVATE void dequeue(rp)
register struct proc *rp;          /* this process is no longer runnable */
{
    /* A process must be removed from the scheduling queues, for example, because
    * it has blocked. If the currently active process is removed, a new process
    * is picked to run by calling pick_proc().
    */
    register int q = rp->p_priority;          /* queue to use */
    register struct proc **xpp;              /* iterate over queue */
    register struct proc *prev_xp;

    /* Side-effect for kernel: check if the task's stack still is ok? */
    if (iskernelp(rp)) {
        if (*priv(rp)->s_stack_guard != STACK_GUARD)
            panic("stack overrun by task", proc_nr(rp));
    }

#if DEBUG_SCHED_CHECK
    check_runqueues("dequeue");
    if (! rp->p_ready) kprintf("dequeue() already unready process\n");
#endif

    /* Now make sure that the process is not in its ready queue. Remove the
    * process if it is found. A process can be made unready even if it is not
    * running by being sent a signal that kills it.
    */
    prev_xp = NIL_PROC;
    for (xpp = &rdy_head[q]; *xpp != NIL_PROC; xpp = &(*xpp)->p_nextready) {
        if (*xpp == rp) {
            /* found process to remove */
            *xpp = (*xpp)->p_nextready; /* replace with next chain */
            if (rp == rdy_tail[q])
                rdy_tail[q] = prev_xp; /* queue tail removed */
            /* set new tail */
            if (rp == proc_ptr || rp == next_ptr) /* active process removed */
                pick_proc(); /* pick new process to run */
            break;
        }
        prev_xp = *xpp; /* save previous in chain */
    }

#if DEBUG_SCHED_CHECK
    rp->p_ready = 0;
    check_runqueues("dequeue");
#endif
}

```



```

/*=====
 *                                     sched                                     *
 *=====*/
PRIVATE void sched(rp, queue, front)
register struct proc *rp;          /* process to be scheduled */
int *queue;                       /* return: queue to use */
int *front;                       /* return: front or back */
{
/* This function determines the scheduling policy. It is called whenever a
 * process must be added to one of the scheduling queues to decide where to
 * insert it. As a side-effect the process' priority may be updated.
 */
    int time_left = (rp->p_ticks_left > 0);    /* quantum fully consumed */

/* Check whether the process has time left. Otherwise give a new quantum
 * and lower the process' priority, unless the process already is in the
 * lowest queue.
 */
    if (!time_left) {                /* quantum consumed ? */
        rp->p_ticks_left = rp->p_quantum_size; /* give new quantum */
        if (rp->p_priority < (IDLE_Q-1)) {
            rp->p_priority += 1;        /* lower priority */
        }
    }

/* If there is time left, the process is added to the front of its queue,
 * so that it can immediately run. The queue to use simply is always the
 * process' current priority.
 */
    *queue = rp->p_priority;
    *front = time_left;
}

/*=====
 *                                     pick_proc                               *
 *=====*/
PRIVATE void pick_proc()
{
/* Decide who to run now. A new process is selected by setting 'next_ptr'.
 * When a billable process is selected, record it in 'bill_ptr', so that the
 * clock task can tell who to bill for system time.
 */
    register struct proc *rp;        /* process to run */
    int q;                          /* iterate over queues */

/* Check each of the scheduling queues for ready processes. The number of
 * queues is defined in proc.h, and priorities are set in the task table.
 * The lowest queue contains IDLE, which is always ready.
 */
    for (q=0; q < NR_SCHED_QUEUES; q++) {
        if ( (rp = rdy_head[q]) != NIL_PROC) {
            next_ptr = rp;            /* run process 'rp' next */
            if (priv(rp)->s_flags & BILLABLE)
                bill_ptr = rp;        /* bill for system time */
            return;
        }
    }
}

/*=====
 *                                     balance_queues                           *
 *=====*/
#define Q_BALANCE_TICKS 100
PUBLIC void balance_queues(tp)
timer_t *tp;                      /* watchdog timer pointer */
{
/* Check entire process table and give all process a higher priority. This
 * effectively means giving a new quantum. If a process already is at its
 * maximum priority, its quantum will be renewed.
 */
    static timer_t queue_timer;      /* timer structure to use */
    register struct proc* rp;        /* process table pointer */
    clock_t next_period;              /* time of next period */

```

```

int ticks_added = 0;                                /* total time added */

for (rp=BEG_PROC_ADDR; rp<END_PROC_ADDR; rp++) {
    if (! isemptyp(rp)) {                            /* check slot use */
        lock(5, "balance_queues");
        if (rp->p_priority > rp->p_max_priority) {    /* update priority? */
            if (rp->p_rts_flags == 0) dequeue(rp);    /* take off queue */
            ticks_added += rp->p_quantum_size;        /* do accounting */
            rp->p_priority -= 1;                      /* raise priority */
            if (rp->p_rts_flags == 0) enqueue(rp);    /* put on queue */
        }
        else {
            ticks_added += rp->p_quantum_size - rp->p_ticks_left;
            rp->p_ticks_left = rp->p_quantum_size;    /* give new quantum */
        }
        unlock(5);
    }
}
#endif
kprintf("ticks_added: %d\n", ticks_added);
#endif

/* Now schedule a new watchdog timer to balance the queues again. The
 * period depends on the total amount of quantum ticks added.
 */
next_period = MAX(Q_BALANCE_TICKS, ticks_added);    /* calculate next */
set_timer(&queue_timer, get_uptime() + next_period, balance_queues);
}

/*=====
 *                               lock_send                               *
 *=====*/
PUBLIC int lock_send(dst_e, m_ptr)
int dst_e;                                /* to whom is message being sent? */
message *m_ptr;                          /* pointer to message buffer */
{
    /* Safe gateway to mini_send() for tasks. */
    int result;
    lock(2, "send");
    result = mini_send(proc_ptr, dst_e, m_ptr, NON_BLOCKING);
    unlock(2);
    return(result);
}

/*=====
 *                               lock_enqueue                           *
 *=====*/
PUBLIC void lock_enqueue(rp)
struct proc *rp;                          /* this process is now runnable */
{
    /* Safe gateway to enqueue() for tasks. */
    lock(3, "enqueue");
    enqueue(rp);
    unlock(3);
}

/*=====
 *                               lock_dequeue                           *
 *=====*/
PUBLIC void lock_dequeue(rp)
struct proc *rp;                          /* this process is no longer runnable */
{
    /* Safe gateway to dequeue() for tasks. */
    if (k_reenter >= 0) {
        /* We're in an exception or interrupt, so don't lock (and ...
         * don't unlock).
         */
        dequeue(rp);
    } else {
        lock(4, "dequeue");
        dequeue(rp);
        unlock(4);
    }
}

```

```

/*=====
 *
 *                               isokendpt_f
 *=====*/
#if DEBUG_ENABLE_IPC_WARNINGS
PUBLIC int isokendpt_f(file, line, e, p, fatalflag)
char *file;
int line;
#else
PUBLIC int isokendpt_f(e, p, fatalflag)
#endif
int e, *p, fatalflag;
{
    int ok = 0;
    /* Convert an endpoint number into a process number.
     * Return nonzero if the process is alive with the corresponding
     * generation number, zero otherwise.
     *
     * This function is called with file and line number by the
     * isokendpt_d macro if DEBUG_ENABLE_IPC_WARNINGS is defined,
     * otherwise without. This allows us to print the where the
     * conversion was attempted, making the errors verbose without
     * adding code for that at every call.
     *
     * If fatalflag is nonzero, we must panic if the conversion doesn't
     * succeed.
     */
    *p = _ENDPOINT_P(e);
    if (!isokprocn(*p)) {
#if DEBUG_ENABLE_IPC_WARNINGS
        kprintf("kernel:%s:%d: bad endpoint %d: proc %d out of range\n",
            file, line, e, *p);
#endif
    } else if (isempty(*p)) {
#if DEBUG_ENABLE_IPC_WARNINGS
        kprintf("kernel:%s:%d: bad endpoint %d: proc %d empty\n", file, line, e, *p);
#endif
    } else if (proc_addr(*p)->p_endpoint != e) {
#if DEBUG_ENABLE_IPC_WARNINGS
        kprintf("kernel:%s:%d: bad endpoint %d: proc %d has ept %d (generation %d vs. %d)\n", file, l
ine,
            e, *p, proc_addr(*p)->p_endpoint,
            _ENDPOINT_G(e), _ENDPOINT_G(proc_addr(*p)->p_endpoint));
#endif
    } else ok = 1;
    if (!ok && fatalflag) {
        panic("invalid endpoint ", e);
    }
    return ok;
}

```

```

#ifndef PROC_H
#define PROC_H

/* Here is the declaration of the process table. It contains all process
 * data, including registers, flags, scheduling priority, memory map,
 * accounting, message passing (IPC) information, and so on.
 *
 * Many assembly code routines reference fields in it. The offsets to these
 * fields are defined in the assembler include file sconst.h. When changing
 * struct proc, be sure to change sconst.h to match.
 */
#include <minix/com.h>
#include "protect.h"
#include "const.h"
#include "priv.h"

struct proc {
    struct stackframe_s p_reg;      /* process' registers saved in stack frame */

#if (CHIP == INTEL)
        reg_t p_ldt_sel;           /* selector in gdt with ldt base and limit */
        struct segdesc_s p_ldt[2+NR_REMOTE_SEGS]; /* CS, DS and remote segments */
#endif

#if (CHIP == M68000)
/* M68000 specific registers and FPU details go here. */
#endif

    proc_nr_t p_nr;                /* number of this process (for fast access) */
    struct priv *p_priv;           /* system privileges structure */
    short p_rts_flags;             /* process is runnable only if zero */
    short p_misc_flags;           /* flags that do suspend the process */

    char p_priority;               /* current scheduling priority */
    char p_max_priority;          /* maximum scheduling priority */
    char p_ticks_left;            /* number of scheduling ticks left */
    char p_quantum_size;          /* quantum size in ticks */

    struct mem_map p_memmap[NR_LOCAL_SEGS]; /* memory map (T, D, S) */

    clock_t p_user_time;          /* user time in ticks */
    clock_t p_sys_time;          /* sys time in ticks */

    struct proc *p_nextready;     /* pointer to next ready process */
    struct proc *p_caller_q;      /* head of list of procs wishing to send */
    struct proc *p_q_link;        /* link to next proc wishing to send */
    message *p_messbuf;           /* pointer to passed message buffer */
    int p_getfrom_e;              /* from whom does process want to receive? */
    int p_sendto_e;              /* to whom does process want to send? */

    sigset_t p_pending;           /* bit map for pending kernel signals */

    char p_name[P_NAME_LEN];      /* name of the process, including \0 */

    int p_endpoint;              /* endpoint number, generation-aware */

#if DEBUG_SCHED_CHECK
    int p_ready, p_found;
#endif
};

/* Bits for the runtime flags. A process is runnable iff p_rts_flags == 0. */
#define SLOT_FREE      0x01    /* process slot is free */
#define NO_PRIORITY    0x02    /* process has been stopped */
#define SENDING        0x04    /* process blocked trying to send */
#define RECEIVING      0x08    /* process blocked trying to receive */
#define SIGNED         0x10    /* set when new kernel signal arrives */
#define SIG_PENDING    0x20    /* unready while signal being processed */
#define P_STOP         0x40    /* set when process is being traced */
#define NO_PRIV        0x80    /* keep forked system process from running */
#define NO_ENDPOINT    0x100   /* process cannot send or receive messages */

/* Misc flags */
#define REPLY_PENDING  0x01    /* reply to IPC_REQUEST is pending */

```

```

#define MF_VM          0x08      /* process uses VM */

/* Scheduling priorities for p_priority. Values must start at zero (highest
 * priority) and increment. Priorities of the processes in the boot image
 * can be set in table.c. IDLE must have a queue for itself, to prevent low
 * priority user processes to run round-robin with IDLE.
 */
#define NR_SCHED_QUEUES 16      /* MUST equal minimum priority + 1 */
#define TASK_Q          0      /* highest, used for kernel tasks */
#define MAX_USER_Q      0      /* highest priority for user processes */
#define USER_Q          7      /* default (should correspond to nice 0) */
#define MIN_USER_Q      14     /* minimum priority for user processes */
#define IDLE_Q           15     /* lowest, only IDLE process goes here */

/* Magic process table addresses. */
#define BEG_PROC_ADDR (&proc[0])
#define BEG_USER_ADDR (&proc[NR_TASKS])
#define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])

#define NIL_PROC        ((struct proc *) 0)
#define NIL_SYS_PROC    ((struct proc *) 1)
#define cproc_addr(n)   (&(proc + NR_TASKS)[(n)])
#define pproc_addr(n)   (pproc_addr + NR_TASKS)[(n)]
#define proc_nr(p)      ((p)->p_nr)

#define isokprocn(n)     ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
#define isemptyn(n)      isemptyp(proc_addr(n))
#define isemptyp(p)      ((p)->p_rts_flags == SLOT_FREE)
#define iskernelp(p)     iskerneln((p)->p_nr)
#define iskerneln(n)     ((n) < 0)
#define isuserp(p)       isusern((p)->p_nr)
#define isusern(n)       ((n) >= 0)

/* The process table and pointers to process table slots. The pointers allow
 * faster access because now a process entry can be found by indexing the
 * pproc_addr array, while accessing an element i requires a multiplication
 * with sizeof(struct proc) to determine the address.
 */
EXTERN struct proc proc[NR_TASKS + NR_PROCS]; /* process table */
EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
EXTERN struct proc *rdy_head[NR_SCHED_QUEUES]; /* ptrs to ready list headers */
EXTERN struct proc *rdy_tail[NR_SCHED_QUEUES]; /* ptrs to ready list tails */

#endif /* PROC_H */

```

```

/* This file contains code for initialization of protected mode, to initialize
 * code and data segment descriptors, and to initialize global descriptors
 * for local descriptors in the process table.
 */

#include "kernel.h"
#include "proc.h"
#include "protect.h"

#if _WORD_SIZE == 4
#define INT_GATE_TYPE      (INT_286_GATE | DESC_386_BIT)
#define TSS_TYPE           (AVL_286_TSS | DESC_386_BIT)
#else
#define INT_GATE_TYPE      INT_286_GATE
#define TSS_TYPE           AVL_286_TSS
#endif

struct desctableptr_s {
    char limit[sizeof(u16_t)];
    char base[sizeof(u32_t)];           /* really u24_t + pad for 286 */
};

struct gatedesc_s {
    u16_t offset_low;
    u16_t selector;
    u8_t pad;                          /* |000|XXXXX| ig & trpg, |XXXXXXXX| task g */
    u8_t p_dpl_type;                  /* |P|DL|0|TYPE| */
    u16_t offset_high;
};

struct tss_s {
    reg_t backlink;
    reg_t sp0;                         /* stack pointer to use during interrupt */
    reg_t ss0;                         /* " segment " " " " " */
    reg_t spl;
    reg_t ss1;
    reg_t sp2;
    reg_t ss2;
};

#if _WORD_SIZE == 4
    reg_t cr3;
#endif

reg_t ip;
reg_t flags;
reg_t ax;
reg_t cx;
reg_t dx;
reg_t bx;
reg_t sp;
reg_t bp;
reg_t si;
reg_t di;
reg_t es;
reg_t cs;
reg_t ss;
reg_t ds;

#if _WORD_SIZE == 4
    reg_t fs;
    reg_t gs;
#endif

reg_t ldt;

#if _WORD_SIZE == 4
    u16_t trap;
    u16_t iobase;
    /* u8_t iomap[0]; */
#endif

PUBLIC struct segdesc_s gdt[GDT_SIZE]; /* used in klib.s and mpx.s */
PRIVATE struct gatedesc_s idt[IDT_SIZE]; /* zero-init so none present */
PUBLIC struct tss_s tss;                /* zero init */

FORWARD _PROTOTYPE( void int_gate, (unsigned vec_nr, vir_bytes offset,
                                   unsigned dpl_type) );
FORWARD _PROTOTYPE( void sdesc, (struct segdesc_s *segdp, phys_bytes base,

```

```

        vir_bytes size) );

/*=====
 *                               prot_init                               *
 *=====*/
PUBLIC void prot_init()
{
    /* Set up tables for protected mode.
     * All GDT slots are allocated at compile time.
     */
    struct gate_table_s *gtp;
    struct desctableptr_s *dtp;
    unsigned ldt_index;
    register struct proc *rp;

    static struct gate_table_s {
        _PROTOTYPE( void (*gate), (void) );
        unsigned char vec_nr;
        unsigned char privilege;
    }
    gate_table[] = {
        { divide_error, DIVIDE_VECTOR, INTR_PRIVILEGE },
        { single_step_exception, DEBUG_VECTOR, INTR_PRIVILEGE },
        { nmi, NMI_VECTOR, INTR_PRIVILEGE },
        { breakpoint_exception, BREAKPOINT_VECTOR, USER_PRIVILEGE },
        { overflow, OVERFLOW_VECTOR, USER_PRIVILEGE },
        { bounds_check, BOUNDS_VECTOR, INTR_PRIVILEGE },
        { inval_opcode, INVAL_OP_VECTOR, INTR_PRIVILEGE },
        { copr_not_available, COPROC_NOT_VECTOR, INTR_PRIVILEGE },
        { double_fault, DOUBLE_FAULT_VECTOR, INTR_PRIVILEGE },
        { copr_seg_overrun, COPROC_SEG_VECTOR, INTR_PRIVILEGE },
        { inval_tss, INVAL_TSS_VECTOR, INTR_PRIVILEGE },
        { segment_not_present, SEG_NOT_VECTOR, INTR_PRIVILEGE },
        { stack_exception, STACK_FAULT_VECTOR, INTR_PRIVILEGE },
        { general_protection, PROTECTION_VECTOR, INTR_PRIVILEGE },
#ifdef _WORD_SIZE == 4
        { page_fault, PAGE_FAULT_VECTOR, INTR_PRIVILEGE },
        { copr_error, COPROC_ERR_VECTOR, INTR_PRIVILEGE },
#endif
        { hwint00, VECTOR( 0), INTR_PRIVILEGE },
        { hwint01, VECTOR( 1), INTR_PRIVILEGE },
        { hwint02, VECTOR( 2), INTR_PRIVILEGE },
        { hwint03, VECTOR( 3), INTR_PRIVILEGE },
        { hwint04, VECTOR( 4), INTR_PRIVILEGE },
        { hwint05, VECTOR( 5), INTR_PRIVILEGE },
        { hwint06, VECTOR( 6), INTR_PRIVILEGE },
        { hwint07, VECTOR( 7), INTR_PRIVILEGE },
        { hwint08, VECTOR( 8), INTR_PRIVILEGE },
        { hwint09, VECTOR( 9), INTR_PRIVILEGE },
        { hwint10, VECTOR(10), INTR_PRIVILEGE },
        { hwint11, VECTOR(11), INTR_PRIVILEGE },
        { hwint12, VECTOR(12), INTR_PRIVILEGE },
        { hwint13, VECTOR(13), INTR_PRIVILEGE },
        { hwint14, VECTOR(14), INTR_PRIVILEGE },
        { hwint15, VECTOR(15), INTR_PRIVILEGE },
#ifdef _WORD_SIZE == 2
        { p_s_call, SYS_VECTOR, USER_PRIVILEGE },          /* 286 system call */
#else
        { s_call, SYS386_VECTOR, USER_PRIVILEGE },          /* 386 system call */
#endif
        { level0_call, LEVEL0_VECTOR, TASK_PRIVILEGE },
    };

    /* Build gdt and idt pointers in GDT where the BIOS expects them. */
    dtp= (struct desctableptr_s *) &gdt[GDT_INDEX];
    * (u16_t *) dtp->limit = (sizeof gdt) - 1;
    * (u32_t *) dtp->base = vir2phys(gdt);

    dtp= (struct desctableptr_s *) &gdt[IDT_INDEX];
    * (u16_t *) dtp->limit = (sizeof idt) - 1;
    * (u32_t *) dtp->base = vir2phys(idt);

    /* Build segment descriptors for tasks and interrupt handlers. */
    init_codeseg(&gdt[CS_INDEX],

```

```

    kinfo.code_base, kinfo.code_size, INTR_PRIVILEGE);
init_dataseg(&gdt[DS_INDEX],
    kinfo.data_base, kinfo.data_size, INTR_PRIVILEGE);
init_dataseg(&gdt[ES_INDEX], 0L, 0, TASK_PRIVILEGE);

/* Build scratch descriptors for functions in klib88. */
init_dataseg(&gdt[DS_286_INDEX], 0L, 0, TASK_PRIVILEGE);
init_dataseg(&gdt[ES_286_INDEX], 0L, 0, TASK_PRIVILEGE);

/* Build local descriptors in GDT for LDT's in process table.
 * The LDT's are allocated at compile time in the process table, and
 * initialized whenever a process' map is initialized or changed.
 */
for (rp = BEG_PROC_ADDR, ldt_index = FIRST_LDT_INDEX;
    rp < END_PROC_ADDR; ++rp, ldt_index++) {
    init_dataseg(&gdt[ldt_index], vir2phys(rp->p_ldt),
        sizeof(rp->p_ldt), INTR_PRIVILEGE);
    gdt[ldt_index].access = PRESENT | LDT;
    rp->p_ldt_sel = ldt_index * DESC_SIZE;
}

/* Build main TSS.
 * This is used only to record the stack pointer to be used after an
 * interrupt.
 * The pointer is set up so that an interrupt automatically saves the
 * current process's registers ip:cs:f:sp:ss in the correct slots in the
 * process table.
 */
tss.ss0 = DS_SELECTOR;
init_dataseg(&gdt[TSS_INDEX], vir2phys(&tss), sizeof(tss), INTR_PRIVILEGE);
gdt[TSS_INDEX].access = PRESENT | (INTR_PRIVILEGE << DPL_SHIFT) | TSS_TYPE;

/* Build descriptors for interrupt gates in IDT. */
for (gtp = &gate_table[0];
    gtp < &gate_table[sizeof gate_table / sizeof gate_table[0]]; ++gtp) {
    int_gate(gtp->vec_nr, (vir_bytes) gtp->gate,
        PRESENT | INT_GATE_TYPE | (gtp->privilege << DPL_SHIFT));
}

#if _WORD_SIZE == 4
/* Complete building of main TSS. */
tss.iobase = sizeof tss; /* empty i/o permissions map */
#endif
}

/*=====
 *                               init_codeseg                               *
 *=====*/
PUBLIC void init_codeseg(segdp, base, size, privilege)
register struct segdesc_s *segdp;
phys_bytes base;
vir_bytes size;
int privilege;
{
/* Build descriptor for a code segment. */
sdesc(segdp, base, size);
segdp->access = (privilege << DPL_SHIFT)
    | (PRESENT | SEGMENT | EXECUTABLE | READABLE);
/* CONFORMING = 0, ACCESSED = 0 */
}

/*=====
 *                               init_dataseg                               *
 *=====*/
PUBLIC void init_dataseg(segdp, base, size, privilege)
register struct segdesc_s *segdp;
phys_bytes base;
vir_bytes size;
int privilege;
{
/* Build descriptor for a data segment. */
sdesc(segdp, base, size);
segdp->access = (privilege << DPL_SHIFT) | (PRESENT | SEGMENT | WRITEABLE);
/* EXECUTABLE = 0, EXPAND_DOWN = 0, ACCESSED = 0 */
}

```



```

}

/*=====
 *
 *                               sdesc
 *=====*/
PRIVATE void sdesc(segdp, base, size)
register struct segdesc_s *segdp;
phys_bytes base;
vir_bytes size;
{
/* Fill in the size fields (base, limit and granularity) of a descriptor. */
segdp->base_low = base;
segdp->base_middle = base >> BASE_MIDDLE_SHIFT;
segdp->base_high = base >> BASE_HIGH_SHIFT;

#if _WORD_SIZE == 4
--size; /* convert to a limit, 0 size means 4G */
if (size > BYTE_GRAN_MAX) {
segdp->limit_low = size >> PAGE_GRAN_SHIFT;
segdp->granularity = GRANULAR | (size >>
(PAGE_GRAN_SHIFT + GRANULARITY_SHIFT));
} else {
segdp->limit_low = size;
segdp->granularity = size >> GRANULARITY_SHIFT;
}
segdp->granularity |= DEFAULT; /* means BIG for data seg */
#else
segdp->limit_low = size - 1;
#endif
}

/*=====
 *
 *                               seg2phys
 *=====*/
PUBLIC phys_bytes seg2phys(seg)
U16_t seg;
{
/* Return the base address of a segment, with seg being either a 8086 segment
 * register, or a 286/386 segment selector.
 */
phys_bytes base;
struct segdesc_s *segdp;

if (! machine.protected) {
base = hclick_to_physb(seg);
} else {
segdp = &gdt[seg >> 3];
base = ((u32_t) segdp->base_low << 0)
| ((u32_t) segdp->base_middle << 16)
| ((u32_t) segdp->base_high << 24);
}
return base;
}

/*=====
 *
 *                               phys2seg
 *=====*/
PUBLIC void phys2seg(seg, off, phys)
ul6_t *seg;
vir_bytes *off;
phys_bytes phys;
{
/* Return a segment selector and offset that can be used to reach a physical
 * address, for use by a driver doing memory I/O in the A0000 - DFFFF range.
 */
#if _WORD_SIZE == 2
if (! machine.protected) {
*seg = phys / HCLICK_SIZE;
*off = phys % HCLICK_SIZE;
} else {
unsigned bank = phys >> 16;
unsigned index = bank - 0xA + A_INDEX;
init_dataseg(&gdt[index], (phys_bytes) bank << 16, 0, TASK_PRIVILEGE);
*seg = (index * 0x08) | TASK_PRIVILEGE;

```

```

        *off = phys & 0xFFFF;
    }
#else
    *seg = FLAT_DS_SELECTOR;
    *off = phys;
#endif
}

/*=====
 *
 *                      int_gate
 *=====*/
PRIVATE void int_gate(vec_nr, offset, dpl_type)
unsigned vec_nr;
vir_bytes offset;
unsigned dpl_type;
{
    /* Build descriptor for an interrupt gate. */
    register struct gatedesc_s *idp;

    idp = &idt[vec_nr];
    idp->offset_low = offset;
    idp->selector = CS_SELECTOR;
    idp->p_dpl_type = dpl_type;
    #if _WORD_SIZE == 4
        idp->offset_high = offset >> OFFSET_HIGH_SHIFT;
    #endif
}

/*=====
 *
 *                      enable_iop
 *=====*/
PUBLIC void enable_iop(pp)
struct proc *pp;
{
    /* Allow a user process to use I/O instructions. Change the I/O Permission
     * Level bits in the psw. These specify least-privileged Current Permission
     * Level allowed to execute I/O instructions. Users and servers have CPL 3.
     * You can't have less privilege than that. Kernel has CPL 0, tasks CPL 1.
     */
    pp->p_reg.psw |= 0x3000;
}

/*=====
 *
 *                      alloc_segments
 *=====*/
PUBLIC void alloc_segments(rp)
register struct proc *rp;
{
    /* This is called at system initialization from main() and by do_newmap().
     * The code has a separate function because of all hardware-dependencies.
     * Note that IDLE is part of the kernel and gets TASK_PRIVILEGE here.
     */
    phys_bytes code_bytes;
    phys_bytes data_bytes;
    int privilege;

    if (machine.protected) {
        data_bytes = (phys_bytes) (rp->p_memmap[S].mem_vir +
            rp->p_memmap[S].mem_len) << CLICK_SHIFT;
        if (rp->p_memmap[T].mem_len == 0)
            code_bytes = data_bytes; /* common I&D, poor protect */
        else
            code_bytes = (phys_bytes) rp->p_memmap[T].mem_len << CLICK_SHIFT;
        privilege = (iskernelp(rp)) ? TASK_PRIVILEGE : USER_PRIVILEGE;
        init_codeseg(&rp->p_ldt[CS_LDT_INDEX],
            (phys_bytes) rp->p_memmap[T].mem_phys << CLICK_SHIFT,
            code_bytes, privilege);
        init_dataseg(&rp->p_ldt[DS_LDT_INDEX],
            (phys_bytes) rp->p_memmap[D].mem_phys << CLICK_SHIFT,
            data_bytes, privilege);
        rp->p_reg.cs = (CS_LDT_INDEX * DESC_SIZE) | TI | privilege;
    }
    #if _WORD_SIZE == 4
        rp->p_reg.gs =
        rp->p_reg.fs =
    #endif

```

```
#endif
    rp->p_reg.ss =
    rp->p_reg.es =
    rp->p_reg.ds = (DS_LDT_INDEX*DESC_SIZE) | TI | privilege;
} else {
    rp->p_reg.cs = click_to_hclick(rp->p_memmap[T].mem_phys);
    rp->p_reg.ss =
    rp->p_reg.es =
    rp->p_reg.ds = click_to_hclick(rp->p_memmap[D].mem_phys);
}
}
```

```

/* Constants for protected mode. */

/* Table sizes. */
#define GDT_SIZE (FIRST_LDT_INDEX + NR_TASKS + NR_PROCS)
/* spec. and LDT's */
#define IDT_SIZE (IRQ8_VECTOR + 8) /* only up to the highest vector */
#define LDT_SIZE (2 + NR_REMOTE_SEGS) /* CS, DS and remote segments */

/* Fixed global descriptors. 1 to 7 are prescribed by the BIOS. */
#define GDT_INDEX 1 /* GDT descriptor */
#define IDT_INDEX 2 /* IDT descriptor */
#define DS_INDEX 3 /* kernel DS */
#define ES_INDEX 4 /* kernel ES (386: flag 4 Gb at startup) */
#define SS_INDEX 5 /* kernel SS (386: monitor SS at startup) */
#define CS_INDEX 6 /* kernel CS */
#define MON_CS_INDEX 7 /* temp for BIOS (386: monitor CS at startup) */
#define TSS_INDEX 8 /* kernel TSS */
#define DS_286_INDEX 9 /* scratch 16-bit source segment */
#define ES_286_INDEX 10 /* scratch 16-bit destination segment */
#define A_INDEX 11 /* 64K memory segment at A0000 */
#define B_INDEX 12 /* 64K memory segment at B0000 */
#define C_INDEX 13 /* 64K memory segment at C0000 */
#define D_INDEX 14 /* 64K memory segment at D0000 */
#define FIRST_LDT_INDEX 15 /* rest of descriptors are LDT's */

#define GDT_SELECTOR 0x08 /* (GDT_INDEX * DESC_SIZE) bad for asld */
#define IDT_SELECTOR 0x10 /* (IDT_INDEX * DESC_SIZE) */
#define DS_SELECTOR 0x18 /* (DS_INDEX * DESC_SIZE) */
#define ES_SELECTOR 0x20 /* (ES_INDEX * DESC_SIZE) */
#define FLAT_DS_SELECTOR 0x21 /* less privileged ES */
#define SS_SELECTOR 0x28 /* (SS_INDEX * DESC_SIZE) */
#define CS_SELECTOR 0x30 /* (CS_INDEX * DESC_SIZE) */
#define MON_CS_SELECTOR 0x38 /* (MON_CS_INDEX * DESC_SIZE) */
#define TSS_SELECTOR 0x40 /* (TSS_INDEX * DESC_SIZE) */
#define DS_286_SELECTOR 0x49 /* (DS_286_INDEX*DESC_SIZE+TASK_PRIVILEGE) */
#define ES_286_SELECTOR 0x51 /* (ES_286_INDEX*DESC_SIZE+TASK_PRIVILEGE) */

/* Fixed local descriptors. */
#define CS_LDT_INDEX 0 /* process CS */
#define DS_LDT_INDEX 1 /* process DS=ES=FS=GS=SS */
#define EXTRA_LDT_INDEX 2 /* first of the extra LDT entries */

/* Privileges. */
#define INTR_PRIVILEGE 0 /* kernel and interrupt handlers */
#define TASK_PRIVILEGE 1 /* kernel tasks */
#define USER_PRIVILEGE 3 /* servers and user processes */

/* 286 hardware constants. */

/* Exception vector numbers. */
#define BOUNDS_VECTOR 5 /* bounds check failed */
#define INVAL_OP_VECTOR 6 /* invalid opcode */
#define COPROC_NOT_VECTOR 7 /* coprocessor not available */
#define DOUBLE_FAULT_VECTOR 8
#define COPROC_SEG_VECTOR 9 /* coprocessor segment overrun */
#define INVAL_TSS_VECTOR 10 /* invalid TSS */
#define SEG_NOT_VECTOR 11 /* segment not present */
#define STACK_FAULT_VECTOR 12 /* stack exception */
#define PROTECTION_VECTOR 13 /* general protection */

/* Selector bits. */
#define TI 0x04 /* table indicator */
#define RPL 0x03 /* requester privilege level */

/* Descriptor structure offsets. */
#define DESC_BASE 2 /* to base_low */
#define DESC_BASE_MIDDLE 4 /* to base_middle */
#define DESC_ACCESS 5 /* to access byte */
#define DESC_SIZE 8 /* sizeof (struct segdesc_s) */

/* Base and limit sizes and shifts. */
#define BASE_MIDDLE_SHIFT 16 /* shift for base --> base_middle */

/* Access-byte and type-byte bits. */

```

```
#define PRESENT          0x80 /* set for descriptor present */
#define DPL              0x60 /* descriptor privilege level mask */
#define DPL_SHIFT        5
#define SEGMENT          0x10 /* set for segment-type descriptors */

/* Access-byte bits. */
#define EXECUTABLE       0x08 /* set for executable segment */
#define CONFORMING       0x04 /* set for conforming segment if executable */
#define EXPAND_DOWN      0x04 /* set for expand-down segment if !executable */
#define READABLE         0x02 /* set for readable segment if executable */
#define WRITEABLE        0x02 /* set for writeable segment if !executable */
#define TSS_BUSY         0x02 /* set if TSS descriptor is busy */
#define ACCESSED         0x01 /* set if segment accessed */

/* Special descriptor types. */
#define AVL_286_TSS      1 /* available 286 TSS */
#define LDT              2 /* local descriptor table */
#define BUSY_286_TSS     3 /* set transparently to the software */
#define CALL_286_GATE    4 /* not used */
#define TASK_GATE        5 /* only used by debugger */
#define INT_286_GATE     6 /* interrupt gate, used for all vectors */
#define TRAP_286_GATE    7 /* not used */

/* Extra 386 hardware constants. */

/* Exception vector numbers. */
#define PAGE_FAULT_VECTOR 14
#define COPROC_ERR_VECTOR 16 /* coprocessor error */

/* Descriptor structure offsets. */
#define DESC_GRANULARITY  6 /* to granularity byte */
#define DESC_BASE_HIGH    7 /* to base_high */

/* Base and limit sizes and shifts. */
#define BASE_HIGH_SHIFT   24 /* shift for base --> base_high */
#define BYTE_GRAN_MAX     0xFFFFFL /* maximum size for byte granular segment */
#define GRANULARITY_SHIFT 16 /* shift for limit --> granularity */
#define OFFSET_HIGH_SHIFT 16 /* shift for (gate) offset --> offset_high */
#define PAGE_GRAN_SHIFT   12 /* extra shift for page granular limits */

/* Type-byte bits. */
#define DESC_386_BIT      0x08 /* 386 types are obtained by ORing with this */
                                /* LDT's and TASK_GATE's don't need it */

/* Granularity byte. */
#define GRANULAR          0x80 /* set for 4K granularilty */
#define DEFAULT           0x40 /* set for 32-bit defaults (executable seg) */
#define BIG               0x40 /* set for "BIG" (expand-down seg) */
#define AVL               0x10 /* 0 for available */
#define LIMIT_HIGH        0x0F /* mask for high bits of limit */
```

```

/* Function prototypes. */

#ifndef PROTO_H
#define PROTO_H

/* Struct declarations. */
struct proc;
struct timer;

/* clock.c */
_PROTOTYPE( void clock_task, (void) ) ;
_PROTOTYPE( void clock_stop, (void) ) ;
_PROTOTYPE( clock_t get_uptime, (void) ) ;
_PROTOTYPE( unsigned long read_clock, (void) ) ;
_PROTOTYPE( void set_timer, (struct timer *tp, clock_t t, tmr_func_t f) ) ;
_PROTOTYPE( void reset_timer, (struct timer *tp) ) ;

/* main.c */
_PROTOTYPE( void main, (void) ) ;
_PROTOTYPE( void prepare_shutdown, (int how) ) ;

/* utility.c */
_PROTOTYPE( int kprintf, (const char *fmt, ...) ) ;
_PROTOTYPE( void panic, (_CONST char *s, int n) ) ;

/* proc.c */
_PROTOTYPE( int sys_call, (int call_nr, int src_dst,
                           message *m_ptr, long bit_map) ) ;
_PROTOTYPE( int lock_notify, (int src, int dst) ) ;
_PROTOTYPE( int lock_send, (int dst, message *m_ptr) ) ;
_PROTOTYPE( void lock_enqueue, (struct proc *rp) ) ;
_PROTOTYPE( void lock_dequeue, (struct proc *rp) ) ;
_PROTOTYPE( void balance_queues, (struct timer *tp) ) ;
#ifdef DEBUG_ENABLE_IPC_WARNINGS
_PROTOTYPE( int isokendpt_f, (char *file, int line, int e, int *p, int f) );
#define isokendpt_d(e, p, f) isokendpt_f(__FILE__, __LINE__, (e), (p), (f))
#else
_PROTOTYPE( int isokendpt_f, (int e, int *p, int f) ) ;
#define isokendpt_d(e, p, f) isokendpt_f((e), (p), (f))
#endif

/* start.c */
_PROTOTYPE( void cstart, (U16_t cs, U16_t ds, U16_t mds,
                          U16_t parmoft, U16_t parmsize) ) ;

/* system.c */
_PROTOTYPE( int get_priv, (register struct proc *rc, int proc_type) ) ;
_PROTOTYPE( void send_sig, (int proc_nr, int sig_nr) ) ;
_PROTOTYPE( void cause_sig, (int proc_nr, int sig_nr) ) ;
_PROTOTYPE( void sys_task, (void) ) ;
_PROTOTYPE( void get_randomness, (int source) ) ;
_PROTOTYPE( int virtual_copy, (struct vir_addr *src, struct vir_addr *dst,
                              vir_bytes bytes) ) ;
#define numap_local(proc_nr, vir_addr, bytes) \
    umap_local(proc_addr(proc_nr), D, (vir_addr), (bytes))
_PROTOTYPE( phys_bytes umap_local, (struct proc *rp, int seg,
                                   vir_bytes vir_addr, vir_bytes bytes) ) ;
_PROTOTYPE( phys_bytes umap_remote, (struct proc *rp, int seg,
                                   vir_bytes vir_addr, vir_bytes bytes) ) ;
_PROTOTYPE( phys_bytes umap_bios, (struct proc *rp, vir_bytes vir_addr,
                                   vir_bytes bytes) ) ;
_PROTOTYPE( void clear_endpoint, (struct proc *rc) ) ;

/* system/do_newmap.c */
_PROTOTYPE( int newmap, (struct proc *rp, struct mem_map *map_ptr) ) ;

#ifdef (CHIP == INTEL)

/* exception.c */
_PROTOTYPE( void exception, (unsigned vec_nr) ) ;

/* i8259.c */
_PROTOTYPE( void intr_init, (int mine) ) ;
_PROTOTYPE( void intr_handle, (irq_hook_t *hook) ) ;

```

```

_PROTOTYPE( void put_irq_handler, (irq_hook_t *hook, int irq,
                                   irq_handler_t handler) );
_PROTOTYPE( void rm_irq_handler, (irq_hook_t *hook) );

/* klib*.s */
_PROTOTYPE( void int86, (void) );
_PROTOTYPE( void cp_mess, (int src,phys_clicks src_clicks,vir_bytes src_offset,
                          phys_clicks dst_clicks, vir_bytes dst_offset) );
_PROTOTYPE( void enable_irq, (irq_hook_t *hook) );
_PROTOTYPE( int disable_irq, (irq_hook_t *hook) );
_PROTOTYPE( ul6_t mem_rdw, (U16_t segm, vir_bytes offset) );
_PROTOTYPE( void phys_copy, (phys_bytes source, phys_bytes dest,
                             phys_bytes count) );
_PROTOTYPE( void phys_memset, (phys_bytes source, unsigned long pattern,
                              phys_bytes count) );
_PROTOTYPE( void phys_insb, (U16_t port, phys_bytes buf, size_t count) );
_PROTOTYPE( void phys_insw, (U16_t port, phys_bytes buf, size_t count) );
_PROTOTYPE( void phys_outsb, (U16_t port, phys_bytes buf, size_t count) );
_PROTOTYPE( void phys_outsw, (U16_t port, phys_bytes buf, size_t count) );
_PROTOTYPE( void reset, (void) );
_PROTOTYPE( void level0, (void (*func)(void)) );
_PROTOTYPE( void monitor, (void) );
_PROTOTYPE( void read_tsc, (unsigned long *high, unsigned long *low) );
_PROTOTYPE( unsigned long read_cr0, (void) );
_PROTOTYPE( void write_cr0, (unsigned long value) );
_PROTOTYPE( void write_cr3, (unsigned long value) );
_PROTOTYPE( unsigned long read_cpu_flags, (void) );

/* mpx*.s */
_PROTOTYPE( void idle_task, (void) );
_PROTOTYPE( void restart, (void) );

/* The following are never called from C (pure asm procs). */

/* Exception handlers (real or protected mode), in numerical order. */
void _PROTOTYPE( int00, (void) ), _PROTOTYPE( divide_error, (void) );
void _PROTOTYPE( int01, (void) ), _PROTOTYPE( single_step_exception, (void) );
void _PROTOTYPE( int02, (void) ), _PROTOTYPE( nmi, (void) );
void _PROTOTYPE( int03, (void) ), _PROTOTYPE( breakpoint_exception, (void) );
void _PROTOTYPE( int04, (void) ), _PROTOTYPE( overflow, (void) );
void _PROTOTYPE( int05, (void) ), _PROTOTYPE( bounds_check, (void) );
void _PROTOTYPE( int06, (void) ), _PROTOTYPE( inval_opcode, (void) );
void _PROTOTYPE( int07, (void) ), _PROTOTYPE( copr_not_available, (void) );
void _PROTOTYPE( double_fault, (void) );
void _PROTOTYPE( copr_seg_overrun, (void) );
void _PROTOTYPE( inval_tss, (void) );
void _PROTOTYPE( segment_not_present, (void) );
void _PROTOTYPE( stack_exception, (void) );
void _PROTOTYPE( general_protection, (void) );
void _PROTOTYPE( page_fault, (void) );
void _PROTOTYPE( copr_error, (void) );

/* Hardware interrupt handlers. */
_PROTOTYPE( void hwint00, (void) );
_PROTOTYPE( void hwint01, (void) );
_PROTOTYPE( void hwint02, (void) );
_PROTOTYPE( void hwint03, (void) );
_PROTOTYPE( void hwint04, (void) );
_PROTOTYPE( void hwint05, (void) );
_PROTOTYPE( void hwint06, (void) );
_PROTOTYPE( void hwint07, (void) );
_PROTOTYPE( void hwint08, (void) );
_PROTOTYPE( void hwint09, (void) );
_PROTOTYPE( void hwint10, (void) );
_PROTOTYPE( void hwint11, (void) );
_PROTOTYPE( void hwint12, (void) );
_PROTOTYPE( void hwint13, (void) );
_PROTOTYPE( void hwint14, (void) );
_PROTOTYPE( void hwint15, (void) );

/* Software interrupt handlers, in numerical order. */
_PROTOTYPE( void trp, (void) );
_PROTOTYPE( void s_call, (void) ), _PROTOTYPE( p_s_call, (void) );
_PROTOTYPE( void level0_call, (void) );

```

```
/* protect.c */
_PROTOTYPE( void prot_init, (void) );
_PROTOTYPE( void init_codeseg, (struct segdesc_s *segdp, phys_bytes base,
                                vir_bytes size, int privilege) );
_PROTOTYPE( void init_dataseg, (struct segdesc_s *segdp, phys_bytes base,
                                vir_bytes size, int privilege) );
_PROTOTYPE( phys_bytes seg2phys, (U16_t seg) );
_PROTOTYPE( void phys2seg, (u16_t *seg, vir_bytes *off, phys_bytes phys) );
_PROTOTYPE( void enable_iop, (struct proc *pp) );
_PROTOTYPE( void alloc_segments, (struct proc *rp) );

/* system/do_vm.c */
_PROTOTYPE( void vm_map_default, (struct proc *pp) );

#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)
/* M68000 specific prototypes go here. */
#endif /* (CHIP == M68000) */

#endif /* PROTO_H */
```



```
! Miscellaneous constants used in assembler code.
W          =          _WORD_SIZE          ! Machine word size.

! Offsets in struct proc. They MUST match proc.h.
P_STACKBASE = 0
#if _WORD_SIZE == 2
ESREG       =          P_STACKBASE
#else
GSREG       =          P_STACKBASE
FSREG       =          GSREG + 2          ! 386 introduces FS and GS segments
ESREG       =          FSREG + 2
#endif
DSREG       =          ESREG + 2
DIREG       =          DSREG + 2
SIREG       =          DIREG + W
BPREG       =          SIREG + W
STREG       =          BPREG + W          ! hole for another SP
BXREG       =          STREG + W
DXREG       =          BXREG + W
CXREG       =          DXREG + W
AXREG       =          CXREG + W
RETADR      =          AXREG + W          ! return address for save() call
PCREG       =          RETADR + W
CSREG       =          PCREG + W
PSWREG      =          CSREG + W
SPREG       =          PSWREG + W
SSREG       =          SPREG + W
P_STACKTOP  =          SSREG + W
P_LDT_SEL   =          P_STACKTOP
P_LDT       =          P_LDT_SEL + W

#if _WORD_SIZE == 2
Msize       =          12                ! size of a message in 16-bit words
#else
Msize       =          9                 ! size of a message in 32-bit words
#endif
```

```

/* This file contains the C startup code for Minix on Intel processors.
 * It cooperates with mpx.s to set up a good environment for main().
 *
 * This code runs in real mode for a 16 bit kernel and may have to switch
 * to protected mode for a 286.
 * For a 32 bit kernel this already runs in protected mode, but the selectors
 * are still those given by the BIOS with interrupts disabled, so the
 * descriptors need to be reloaded and interrupt descriptors made.
 */

#include "kernel.h"
#include "protect.h"
#include "proc.h"
#include <stdlib.h>
#include <string.h>

FORWARD _PROTOTYPE( char *get_value, (_CONST char *params, _CONST char *key));
/*=====
 *
 * cstart
 *=====*/
PUBLIC void cstart(cs, ds, mds, parmoff, parmsize)
U16_t cs, ds;          /* kernel code and data segment */
U16_t mds;             /* monitor data segment */
U16_t parmoff, parmsize; /* boot parameters offset and length */
{
/* Perform system initializations prior to calling main(). Most settings are
 * determined with help of the environment strings passed by MINIX' loader.
 */
char params[128*sizeof(char *)];          /* boot monitor parameters */
register char *value;                      /* value in key=value pair */
extern int etext, end;
int h;

/* Decide if mode is protected; 386 or higher implies protected mode.
 * This must be done first, because it is needed for, e.g., seg2phys().
 * For 286 machines we cannot decide on protected mode, yet. This is
 * done below.
 */
#if _WORD_SIZE != 2
machine.protected = 1;
#endif

/* Record where the kernel and the monitor are. */
kinfo.code_base = seg2phys(cs);
kinfo.code_size = (phys_bytes) &etext;          /* size of code segment */
kinfo.data_base = seg2phys(ds);
kinfo.data_size = (phys_bytes) &end;            /* size of data segment */

/* Initialize protected mode descriptors. */
prot_init();

/* Copy the boot parameters to the local buffer. */
kinfo.params_base = seg2phys(mds) + parmoff;
kinfo.params_size = MIN(parmsize, sizeof(params)-2);
phys_copy(kinfo.params_base, vir2phys(params), kinfo.params_size);

/* Record miscellaneous information for user-space servers. */
kinfo.nr_procs = NR_PROCS;
kinfo.nr_tasks = NR_TASKS;
strncpy(kinfo.release, OS_RELEASE, sizeof(kinfo.release));
kinfo.release[sizeof(kinfo.release)-1] = '\0';
strncpy(kinfo.version, OS_VERSION, sizeof(kinfo.version));
kinfo.version[sizeof(kinfo.version)-1] = '\0';
kinfo.proc_addr = (vir_bytes) proc;
kinfo.kmem_base = vir2phys(0);
kinfo.kmem_size = (phys_bytes) &end;

/* Load average data initialization. */
kloadinfo.proc_last_slot = 0;
for(h = 0; h < _LOAD_HISTORY; h++)
    kloadinfo.proc_load_history[h] = 0;

/* Processor? 86, 186, 286, 386, ...
 * Decide if mode is protected for older machines.

```

```

    */
    machine.processor=atoi(get_value(params, "processor"));
#if _WORD_SIZE == 2
    machine.protected = machine.processor >= 286;
#endif
    if (! machine.protected) mon_return = 0;

    /* XT, AT or MCA bus? */
    value = get_value(params, "bus");
    if (value == NIL_PTR || strcmp(value, "at") == 0) {
        machine.pc_at = TRUE; /* PC-AT compatible hardware */
    } else if (strcmp(value, "mca") == 0) {
        machine.pc_at = machine.ps_mca = TRUE; /* PS/2 with micro channel */
    }

    /* Type of VDU: */
    value = get_value(params, "video"); /* EGA or VGA video unit */
    if (strcmp(value, "ega") == 0) machine.vdu_ega = TRUE;
    if (strcmp(value, "vga") == 0) machine.vdu_vga = machine.vdu_ega = TRUE;

    /* Return to assembler code to switch to protected mode (if 286),
     * reload selectors and call main().
     */
}

/*=====
 *                               get_value                               *
 *=====*/

PRIVATE char *get_value(params, name)
_CONST char *params; /* boot monitor parameters */
_CONST char *name; /* key to look up */
{
    /* Get environment value - kernel version of getenv to avoid setting up the
     * usual environment array.
     */
    register _CONST char *namep;
    register char *envp;

    for (envp = (char *) params; *envp != 0;) {
        for (namep = name; *namep != 0 && *namep == *envp; namep++, envp++)
            ;
        if (*namep == '\0' && *envp == '=') return(envp + 1);
        while (*envp++ != 0)
            ;
    }
    return(NIL_PTR);
}

```

```

/* This task handles the interface between the kernel and user-level servers.
 * System services can be accessed by doing a system call. System calls are
 * transformed into request messages, which are handled by this task. By
 * convention, a sys_call() is transformed in a SYS_CALL request message that
 * is handled in a function named do_call().
 *
 * A private call vector is used to map all system calls to the functions that
 * handle them. The actual handler functions are contained in separate files
 * to keep this file clean. The call vector is used in the system task's main
 * loop to handle all incoming requests.
 *
 * In addition to the main sys_task() entry point, which starts the main loop,
 * there are several other minor entry points:
 *   get_priv:      assign privilege structure to user or system process
 *   send_sig:      send a signal directly to a system process
 *   cause_sig:     take action to cause a signal to occur via PM
 *   umap_local:    map virtual address in LOCAL_SEG to physical
 *   umap_remote:   map virtual address in REMOTE_SEG to physical
 *   umap_bios:     map virtual address in BIOS_SEG to physical
 *   virtual_copy:  copy bytes from one virtual address to another
 *   get_randomness: accumulate randomness in a buffer
 *   clear_endpoint: remove a process' ability to send and receive messages
 *
 * Changes:
 *   Aug 04, 2005    check if system call is allowed (Jorrit N. Herder)
 *   Jul 20, 2005    send signal to services with message (Jorrit N. Herder)
 *   Jan 15, 2005    new, generalized virtual copy function (Jorrit N. Herder)
 *   Oct 10, 2004    dispatch system calls from call vector (Jorrit N. Herder)
 *   Sep 30, 2004    source code documentation updated (Jorrit N. Herder)
 */

```

```

#include "debug.h"
#include "kernel.h"
#include "system.h"
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/sigcontext.h>
#include <minix/endpoint.h>
#if (CHIP == INTEL)
#include <ibm/memory.h>
#include "protect.h"
#endif

```

```

/* Declaration of the call vector that defines the mapping of system calls
 * to handler functions. The vector is initialized in sys_init() with map(),
 * which makes sure the system call numbers are ok. No space is allocated,
 * because the dummy is declared extern. If an illegal call is given, the
 * array size will be negative and this won't compile.
 */

```

```

PUBLIC int (*call_vec[NR_SYS_CALLS])(message *m_ptr);

```

```

#define map(call_nr, handler) \
    {extern int dummy[NR_SYS_CALLS>(unsigned)(call_nr-KERNEL_CALL) ? 1:-1];} \
    call_vec[(call_nr-KERNEL_CALL)] = (handler)

```

```

FORWARD _PROTOTYPE( void initialize, (void));

```

```

/*=====
 *                               sys_task                               *
 *=====*/

```

```

PUBLIC void sys_task()

```

```

{
/* Main entry point of sys_task. Get the message and dispatch on type. */

```

```

    static message m;
    register int result;
    register struct proc *caller_ptr;
    unsigned int call_nr;
    int s;

```

```

    /* Initialize the system task. */
    initialize();

```

```

    while (TRUE) {

```

```

/* Get work. Block and wait until a request message arrives. */
receive(ANY, &m);
call_nr = (unsigned) m.m_type - KERNEL_CALL;
who_e = m.m_source;
okendpt(who_e, &who_p);
caller_ptr = proc_addr(who_p);

/* See if the caller made a valid request and try to handle it. */
if (! (priv(caller_ptr)->s_call_mask & (1<<call_nr))) {
#if DEBUG_ENABLE_IPC_WARNINGS
    kprintf("SYSTEM: request %d from %d denied.\n", call_nr, m.m_source);
#endif
    result = ECALLDENIED; /* illegal message type */
} else if (call_nr >= NR_SYS_CALLS) { /* check call number */
#if DEBUG_ENABLE_IPC_WARNINGS
    kprintf("SYSTEM: illegal request %d from %d.\n", call_nr, m.m_source);
#endif
    result = EBADREQUEST; /* illegal message type */
} else {
    result = (*call_vec[call_nr])(&m); /* handle the system call */
}

/* Send a reply, unless inhibited by a handler function. Use the kernel
 * function lock_send() to prevent a system call trap. The destination
 * is known to be blocked waiting for a message.
 */
if (result != EDONTREPLY) {
    m.m_type = result; /* report status of call */
    if (OK != (s=lock_send(m.m_source, &m))) {
        kprintf("SYSTEM, reply to %d failed: %d\n", m.m_source, s);
    }
}
}
}

/*=====
 * initialize
 *=====*/
PRIVATE void initialize(void)
{
    register struct priv *sp;
    int i;

    /* Initialize IRQ handler hooks. Mark all hooks available. */
    for (i=0; i<NR_IRQ_HOOKS; i++) {
        irq_hooks[i].proc_nr_e = NONE;
    }

    /* Initialize all alarm timers for all processes. */
    for (sp=BEG_PRIV_ADDR; sp < END_PRIV_ADDR; sp++) {
        tmr_inittimer(&(sp->s_alarm_timer));
    }

    /* Initialize the call vector to a safe default handler. Some system calls
     * may be disabled or nonexistant. Then explicitly map known calls to their
     * handler functions. This is done with a macro that gives a compile error
     * if an illegal call number is used. The ordering is not important here.
     */
    for (i=0; i<NR_SYS_CALLS; i++) {
        call_vec[i] = do_unused;
    }

    /* Process management. */
    map(SYS_FORK, do_fork); /* a process forked a new process */
    map(SYS_EXEC, do_exec); /* update process after execute */
    map(SYS_EXIT, do_exit); /* clean up after process exit */
    map(SYS_NICE, do_nice); /* set scheduling priority */
    map(SYS_PRIVCTL, do_privctl); /* system privileges control */
    map(SYS_TRACE, do_trace); /* request a trace operation */

    /* Signal handling. */
    map(SYS_KILL, do_kill); /* cause a process to be signaled */
    map(SYS_GETKSIG, do_getksig); /* PM checks for pending signals */

```

```

map(SYS_ENDKSIG, do_endksig);      /* PM finished processing signal */
map(SYS_SIGSEND, do_sigsend);      /* start POSIX-style signal */
map(SYS_SIGRETURN, do_sigreturn);  /* return from POSIX-style signal */

/* Device I/O. */
map(SYS_IRQCTL, do_irqctl);        /* interrupt control operations */
map(SYS_DEVIO, do_devio);          /* inb, inw, inl, outb, outw, outl */
map(SYS_SDEVIO, do_sdevio);        /* phys_insb, _insw, _outsb, _outsw */
map(SYS_VDEVIO, do_vdevio);        /* vector with devio requests */
map(SYS_INT86, do_int86);          /* real-mode BIOS calls */

/* Memory management. */
map(SYS_NEWMAP, do_newmap);         /* set up a process memory map */
map(SYS_SEGCTL, do_segctl);        /* add segment and get selector */
map(SYS_MEMSET, do_memset);        /* write char to memory area */
map(SYS_VM_SETBUF, do_vm_setbuf);  /* PM passes buffer for page tables */
map(SYS_VM_MAP, do_vm_map);        /* Map/unmap physical (device) memory */

/* Copying. */
map(SYS_UMAP, do_umap);            /* map virtual to physical address */
map(SYS_VIRCOPY, do_vircopy);      /* use pure virtual addressing */
map(SYS_PHYSCOPY, do_physcopy);    /* use physical addressing */
map(SYS_VIRVCOPY, do_virvcopy);    /* vector with copy requests */
map(SYS_PHYSVCOPY, do_physvcopy);  /* vector with copy requests */

/* Clock functionality. */
map(SYS_TIMES, do_times);          /* get uptime and process times */
map(SYS_SETALARM, do_setalarm);    /* schedule a synchronous alarm */

/* System control. */
map(SYS_ABORT, do_abort);          /* abort MINIX */
map(SYS_GETINFO, do_getinfo);      /* request system information */
map(SYS_IOPENABLE, do_iopenable);  /* Enable I/O */
}

/*=====
*                               get_priv                               *
*=====*/
PUBLIC int get_priv(rc, proc_type)
register struct proc *rc;          /* new (child) process pointer */
int proc_type;                    /* system or user process flag */
{
/* Get a privilege structure. All user processes share the same privilege
* structure. System processes get their own privilege structure.
*/
register struct priv *sp;          /* privilege structure */

if (proc_type == SYS_PROC) {      /* find a new slot */
for (sp = BEG_PRIV_ADDR; sp < END_PRIV_ADDR; ++sp)
if (sp->s_proc_nr == NONE && sp->s_id != USER_PRIV_ID) break;
if (sp->s_proc_nr != NONE) return(ENOSPC);
rc->p_priv = sp;                  /* assign new slot */
rc->p_priv->s_proc_nr = proc_nr(rc); /* set association */
rc->p_priv->s_flags = SYS_PROC;      /* mark as privileged */
} else {
rc->p_priv = &priv[USER_PRIV_ID]; /* use shared slot */
rc->p_priv->s_proc_nr = INIT_PROC_NR; /* set association */
rc->p_priv->s_flags = 0;             /* no initial flags */
}
return(OK);
}

/*=====
*                               get_randomness                         *
*=====*/
PUBLIC void get_randomness(source)
int source;
{
/* On machines with the RDTSC (cycle counter read instruction - pentium
* and up), use that for high-resolution raw entropy gathering. Otherwise,
* use the realtime clock (tick resolution).
*
* Unfortunately this test is run-time - we don't want to bother with
* compiling different kernels for different machines.

```

```

*
* On machines without RDTSC, we use read_clock().
*/
int r_next;
unsigned long tsc_high, tsc_low;

source %= RANDOM_SOURCES;
r_next= krandom.bin[source].r_next;
if (machine.processor > 486) {
    read_tsc(&tsc_high, &tsc_low);
    krandom.bin[source].r_buf[r_next] = tsc_low;
} else {
    krandom.bin[source].r_buf[r_next] = read_clock();
}
if (krandom.bin[source].r_size < RANDOM_ELEMENTS) {
    krandom.bin[source].r_size ++;
}
krandom.bin[source].r_next = (r_next + 1) % RANDOM_ELEMENTS;
}

/*=====
*                               send_sig                               *
*=====*/
PUBLIC void send_sig(int proc_nr, int sig_nr)
{
    /* Notify a system process about a signal. This is straightforward. Simply
    * set the signal that is to be delivered in the pending signals map and
    * send a notification with source SYSTEM.
    *
    * Process number is verified to avoid writing in random places, but we
    * don't kprintf() or panic() because that causes send_sig() invocations.
    */
    register struct proc *rp;
    static int n;

    if(!isokprocn(proc_nr) || isemptyn(proc_nr))
        return;

    rp = proc_addr(proc_nr);
    sigaddset(&priv(rp)->s_sig_pending, sig_nr);
    lock_notify(SYSTEM, rp->p_endpoint);
}

/*=====
*                               cause_sig                               *
*=====*/
PUBLIC void cause_sig(proc_nr, sig_nr)
int proc_nr;           /* process to be signalled */
int sig_nr;            /* signal to be sent, 1 to _NSIG */
{
    /* A system process wants to send a signal to a process. Examples are:
    * - HARDWARE wanting to cause a SIGSEGV after a CPU exception
    * - TTY wanting to cause SIGINT upon getting a DEL
    * - FS wanting to cause SIGPIPE for a broken pipe
    * Signals are handled by sending a message to PM. This function handles the
    * signals and makes sure the PM gets them by sending a notification. The
    * process being signaled is blocked while PM has not finished all signals
    * for it.
    * Race conditions between calls to this function and the system calls that
    * process pending kernel signals cannot exist. Signal related functions are
    * only called when a user process causes a CPU exception and from the kernel
    * process level, which runs to completion.
    */
    register struct proc *rp;

    /* Check if the signal is already pending. Process it otherwise. */
    rp = proc_addr(proc_nr);
    if (! sigismember(&rp->p_pending, sig_nr)) {
        sigaddset(&rp->p_pending, sig_nr);
        if (! (rp->p_rts_flags & SIGNED)) {
            if (rp->p_rts_flags == 0) lock_dequeue(rp); /* other pending */
            rp->p_rts_flags |= SIGNED | SIG_PENDING; /* make not ready */
            send_sig(PM_PROC_NR, SIGKSIG); /* update flags */
        }
    }
}

```

```

}
}

/*=====
*                               umap_bios                               *
*=====*/
PUBLIC phys_bytes umap_bios(rp, vir_addr, bytes)
register struct proc *rp;      /* pointer to proc table entry for process */
vir_bytes vir_addr;           /* virtual address in BIOS segment */
vir_bytes bytes;              /* # of bytes to be copied */
{
/* Calculate the physical memory address at the BIOS. Note: currently, BIOS
* address zero (the first BIOS interrupt vector) is not considered, as an
* error here, but since the physical address will be zero as well, the
* calling function will think an error occurred. This is not a problem,
* since no one uses the first BIOS interrupt vector.
*/

/* Check all acceptable ranges. */
if (vir_addr >= BIOS_MEM_BEGIN && vir_addr + bytes <= BIOS_MEM_END)
    return (phys_bytes) vir_addr;
else if (vir_addr >= BASE_MEM_TOP && vir_addr + bytes <= UPPER_MEM_END)
    return (phys_bytes) vir_addr;

#if DEAD_CODE /* brutal fix, if the above is too restrictive */
    if (vir_addr >= BIOS_MEM_BEGIN && vir_addr + bytes <= UPPER_MEM_END)
        return (phys_bytes) vir_addr;
#endif

    kprintf("Warning, error in umap_bios, virtual address 0x%x\n", vir_addr);
    return 0;
}

/*=====
*                               umap_local                               *
*=====*/
PUBLIC phys_bytes umap_local(rp, seg, vir_addr, bytes)
register struct proc *rp;      /* pointer to proc table entry for process */
int seg;                      /* T, D, or S segment */
vir_bytes vir_addr;           /* virtual address in bytes within the seg */
vir_bytes bytes;              /* # of bytes to be copied */
{
/* Calculate the physical memory address for a given virtual address. */
    vir_clicks vc;             /* the virtual address in clicks */
    phys_bytes pa;             /* intermediate variables as phys_bytes */
#if (CHIP == INTEL)
    phys_bytes seg_base;
#endif

/* If 'seg' is D it could really be S and vice versa. T really means T.
* If the virtual address falls in the gap, it causes a problem. On the
* 8088 it is probably a legal stack reference, since "stackfaults" are
* not detected by the hardware. On 8088s, the gap is called S and
* accepted, but on other machines it is called D and rejected.
* The Atari ST behaves like the 8088 in this respect.
*/

    if (bytes <= 0) return( (phys_bytes) 0);
    if (vir_addr + bytes <= vir_addr) return 0; /* overflow */
    vc = (vir_addr + bytes - 1) >> CLICK_SHIFT; /* last click of data */

#if (CHIP == INTEL) || (CHIP == M68000)
    if (seg != T)
        seg = (vc < rp->p_memmap[D].mem_vir + rp->p_memmap[D].mem_len ? D : S);
#else
    if (seg != T)
        seg = (vc < rp->p_memmap[S].mem_vir ? D : S);
#endif

    if ((vir_addr >> CLICK_SHIFT) >= rp->p_memmap[seg].mem_vir +
        rp->p_memmap[seg].mem_len) return( (phys_bytes) 0 );

    if (vc >= rp->p_memmap[seg].mem_vir +
        rp->p_memmap[seg].mem_len) return( (phys_bytes) 0 );

```



```

#if (CHIP == INTEL)
    seg_base = (phys_bytes) rp->p_memmap[seg].mem_phys;
    seg_base = seg_base << CLICK_SHIFT; /* segment origin in bytes */
#endif
    pa = (phys_bytes) vir_addr;
#if (CHIP != M68000)
    pa -= rp->p_memmap[seg].mem_vir << CLICK_SHIFT;
    return(seg_base + pa);
#endif
#if (CHIP == M68000)
    pa -= (phys_bytes)rp->p_memmap[seg].mem_vir << CLICK_SHIFT;
    pa += (phys_bytes)rp->p_memmap[seg].mem_phys << CLICK_SHIFT;
    return(pa);
#endif
}

/*=====
 *                               umap_remote                               *
 *=====*/
PUBLIC phys_bytes umap_remote(rp, seg, vir_addr, bytes)
register struct proc *rp; /* pointer to proc table entry for process */
int seg; /* index of remote segment */
vir_bytes vir_addr; /* virtual address in bytes within the seg */
vir_bytes bytes; /* # of bytes to be copied */
{
    /* Calculate the physical memory address for a given virtual address. */
    struct far_mem *fm;

    if (bytes <= 0) return( (phys_bytes) 0);
    if (seg < 0 || seg >= NR_REMOTE_SEGS) return( (phys_bytes) 0);

    fm = &rp->p_priv->s_farmem[seg];
    if (! fm->in_use) return( (phys_bytes) 0);
    if (vir_addr + bytes > fm->mem_len) return( (phys_bytes) 0);

    return(fm->mem_phys + (phys_bytes) vir_addr);
}

/*=====
 *                               virtual_copy                               *
 *=====*/
PUBLIC int virtual_copy(src_addr, dst_addr, bytes)
struct vir_addr *src_addr; /* source virtual address */
struct vir_addr *dst_addr; /* destination virtual address */
vir_bytes bytes; /* # of bytes to copy */
{
    /* Copy bytes from virtual address src_addr to virtual address dst_addr.
     * Virtual addresses can be in ABS, LOCAL_SEG, REMOTE_SEG, or BIOS_SEG.
     */
    struct vir_addr *vir_addr[2]; /* virtual source and destination address */
    phys_bytes phys_addr[2]; /* absolute source and destination */
    int seg_index;
    int i;

    /* Check copy count. */
    if (bytes <= 0) return(EDOM);

    /* Do some more checks and map virtual addresses to physical addresses. */
    vir_addr[_SRC_] = src_addr;
    vir_addr[_DST_] = dst_addr;
    for (i=_SRC_; i<=_DST_; i++) {
        int proc_nr, type;
        struct proc *p;

        type = vir_addr[i]->segment & SEGMENT_TYPE;
        if(type != PHYS_SEG && isokendpt(vir_addr[i]->proc_nr_e, &proc_nr))
            p = proc_addr(proc_nr);
        else
            p = NULL;

        /* Get physical address. */
        switch(type) {
            case LOCAL_SEG:

```

```

        if(!p) return EDEADSRCDST;
        seg_index = vir_addr[i]->segment & SEGMENT_INDEX;
        phys_addr[i] = umap_local(p, seg_index, vir_addr[i]->offset, bytes);
        break;
    case REMOTE_SEG:
        if(!p) return EDEADSRCDST;
        seg_index = vir_addr[i]->segment & SEGMENT_INDEX;
        phys_addr[i] = umap_remote(p, seg_index, vir_addr[i]->offset, bytes);
        break;
    case BIOS_SEG:
        if(!p) return EDEADSRCDST;
        phys_addr[i] = umap_bios(p, vir_addr[i]->offset, bytes );
        break;
    case PHYS_SEG:
        phys_addr[i] = vir_addr[i]->offset;
        break;
    default:
        return(EINVAL);
}

/* Check if mapping succeeded. */
if (phys_addr[i] <= 0 && vir_addr[i]->segment != PHYS_SEG)
    return(EFAULT);
}

/* Now copy bytes between physical addresses. */
phys_copy(phys_addr[_SRC_], phys_addr[_DST_], (phys_bytes) bytes);
return(OK);
}

/*=====
 *                               clear_endpoint                               *
 *=====*/
PUBLIC void clear_endpoint(rc)
register struct proc *rc;                /* slot of process to clean up */
{
    register struct proc *rp;            /* iterate over process table */
    register struct proc **xpp;          /* iterate over caller queue */
    int i;
    int sys_id;

    if(isempty(rc)) panic("clear_proc: empty process", proc_nr(rc));

    /* Make sure that the exiting process is no longer scheduled. */
    if (rc->p_rts_flags == 0) lock_dequeue(rc);
    rc->p_rts_flags |= NO_ENDPOINT;

    /* If the process happens to be queued trying to send a
     * message, then it must be removed from the message queues.
     */
    if (rc->p_rts_flags & SENDING) {
        int target_proc;

        okendpt(rc->p_sendto_e, &target_proc);
        xpp = &proc_addr(target_proc)->p_caller_q; /* destination's queue */
        while (*xpp != NIL_PROC) {                 /* check entire queue */
            if (*xpp == rc) {                       /* process is on the queue */
                *xpp = (*xpp)->p_q_link;            /* replace by next process */
            }
        }
#ifdef DEBUG_ENABLE_IPC_WARNINGS
        kprintf("Proc %d removed from queue at %d\n",
            proc_nr(rc), rc->p_sendto_e);
#endif
        break; /* can only be queued once */
    }
    xpp = &(*xpp)->p_q_link; /* proceed to next queued */
}
rc->p_rts_flags &= ~SENDING;
}
rc->p_rts_flags &= ~RECEIVING;

/* Likewise, if another process was sending or receive a message to or from
 * the exiting process, it must be alerted that process no longer is alive.
 * Check all processes.

```

```
*/
for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
    if (isempty(rp))
        continue;

    /* Unset pending notification bits. */
    unset_sys_bit(priv(rp)->s_notify_pending, priv(rc)->s_id);

    /* Check if process is receiving from exiting process. */
    if ((rp->p_rts_flags & RECEIVING) && rp->p_getfrom_e == rc->p_endpoint) {
        rp->p_reg.retreg = ESRCDIED;          /* report source died */
        rp->p_rts_flags &= ~RECEIVING;        /* no longer receiving */
    }
    #if DEBUG_ENABLE_IPC_WARNINGS
        kprintf("Proc %d receive dead src %d\n", proc_nr(rp), proc_nr(rc));
    #endif
    if (rp->p_rts_flags == 0) lock_enqueue(rp); /* let process run again */
}
if ((rp->p_rts_flags & SENDING) && rp->p_sendto_e == rc->p_endpoint) {
    rp->p_reg.retreg = EDSTDIED;          /* report destination died */
    rp->p_rts_flags &= ~SENDING;          /* no longer sending */
}
    #if DEBUG_ENABLE_IPC_WARNINGS
        kprintf("Proc %d send dead dst %d\n", proc_nr(rp), proc_nr(rc));
    #endif
    if (rp->p_rts_flags == 0) lock_enqueue(rp); /* let process run again */
}
}
```

```
/* Function prototypes for the system library. The prototypes in this file
 * are undefined to do_unused if the kernel call is not enabled in config.h.
 * The implementation is contained in src/kernel/system/.
 *
 * The system library allows to access system services by doing a kernel call.
 * System calls are transformed into request messages to the SYS task that is
 * responsible for handling the call. By convention, sys_call() is transformed
 * into a message with type SYS_CALL that is handled in a function do_call().
 *
 * Changes:
 * Jul 30, 2005    created SYS_INT86 to support BIOS driver (Philip Homburg)
 * Jul 13, 2005    created SYS_PRIVCTL to manage services (Jorrit N. Herder)
 * Jul 09, 2005    updated SYS_KILL to signal services (Jorrit N. Herder)
 * Jun 21, 2005    created SYS_NICE for nice(2) kernel call (Ben J. Gras)
 * Jun 21, 2005    created SYS_MEMSET to speed up exec(2) (Ben J. Gras)
 * Apr 12, 2005    updated SYS_VCOPY for virtual_copy() (Jorrit N. Herder)
 * Jan 20, 2005    updated SYS_COPY for virtual_copy() (Jorrit N. Herder)
 * Oct 24, 2004    created SYS_GETKSIG to support PM (Jorrit N. Herder)
 * Oct 10, 2004    created handler for unused calls (Jorrit N. Herder)
 * Sep 09, 2004    updated SYS_EXIT to let services exit (Jorrit N. Herder)
 * Aug 25, 2004    rewrote SYS_SETALARM to clean up code (Jorrit N. Herder)
 * Jul 13, 2004    created SYS_SEGCTL to support drivers (Jorrit N. Herder)
 * May 24, 2004    created SYS_SDEVIO to support drivers (Jorrit N. Herder)
 * May 24, 2004    created SYS_GETINFO to retrieve info (Jorrit N. Herder)
 * Apr 18, 2004    created SYS_VDEVIO to support drivers (Jorrit N. Herder)
 * Feb 24, 2004    created SYS_IRQCTL to support drivers (Jorrit N. Herder)
 * Feb 02, 2004    created SYS_DEVIO to support drivers (Jorrit N. Herder)
 */
```

```
#ifndef SYSTEM_H
#define SYSTEM_H
```

```
/* Common includes for the system library. */
```

```
#include "debug.h"
#include "kernel.h"
#include "proto.h"
#include "proc.h"
```

```
/* Default handler for unused kernel calls. */
_PROTOTYPE( int do_unused, (message *m_ptr) );
```

```
_PROTOTYPE( int do_exec, (message *m_ptr) );
```

```
#if ! USE_EXEC
#define do_exec do_unused
#endif
```

```
_PROTOTYPE( int do_fork, (message *m_ptr) );
```

```
#if ! USE_FORK
#define do_fork do_unused
#endif
```

```
_PROTOTYPE( int do_newmap, (message *m_ptr) );
```

```
#if ! USE_NEWMAP
#define do_newmap do_unused
#endif
```

```
_PROTOTYPE( int do_exit, (message *m_ptr) );
```

```
#if ! USE_EXIT
#define do_exit do_unused
#endif
```

```
_PROTOTYPE( int do_trace, (message *m_ptr) );
```

```
#if ! USE_TRACE
#define do_trace do_unused
#endif
```

```
_PROTOTYPE( int do_nice, (message *m_ptr) );
```

```
#if ! USE_NICE
#define do_nice do_unused
#endif
```

```
_PROTOTYPE( int do_copy, (message *m_ptr) );
```

```
#define do_vircopy do_copy
#define do_physcopy do_copy
```

```
#if ! (USE_VIRCOPY || USE_PHYSCOPY)
#define do_copy do_unused
#endif

__PROTOTYPE( int do_vcopy, (message *m_ptr) );
#define do_virvcopy do_vcopy
#define do_physvcopy do_vcopy
#if ! (USE_VIRVCOPY || USE_PHYSVCOPY)
#define do_vcopy do_unused
#endif

__PROTOTYPE( int do_umap, (message *m_ptr) );
#if ! USE_UMAP
#define do_umap do_unused
#endif

__PROTOTYPE( int do_memset, (message *m_ptr) );
#if ! USE_MEMSET
#define do_memset do_unused
#endif

__PROTOTYPE( int do_vm_setbuf, (message *m_ptr) );
__PROTOTYPE( int do_vm_map, (message *m_ptr) );

__PROTOTYPE( int do_abort, (message *m_ptr) );
#if ! USE_ABORT
#define do_abort do_unused
#endif

__PROTOTYPE( int do_getinfo, (message *m_ptr) );
#if ! USE_GETINFO
#define do_getinfo do_unused
#endif

__PROTOTYPE( int do_privctl, (message *m_ptr) );
#if ! USE_PRIVCTL
#define do_privctl do_unused
#endif

__PROTOTYPE( int do_segctl, (message *m_ptr) );
#if ! USE_SEGCTL
#define do_segctl do_unused
#endif

__PROTOTYPE( int do_irqctl, (message *m_ptr) );
#if ! USE_IRQCTL
#define do_irqctl do_unused
#endif

__PROTOTYPE( int do_devio, (message *m_ptr) );
#if ! USE_DEVIO
#define do_devio do_unused
#endif

__PROTOTYPE( int do_vdevio, (message *m_ptr) );
#if ! USE_VDEVIO
#define do_vdevio do_unused
#endif

__PROTOTYPE( int do_int86, (message *m_ptr) );

__PROTOTYPE( int do_sdevio, (message *m_ptr) );
#if ! USE_SDEVIO
#define do_sdevio do_unused
#endif

__PROTOTYPE( int do_kill, (message *m_ptr) );
#if ! USE_KILL
#define do_kill do_unused
#endif

__PROTOTYPE( int do_getksig, (message *m_ptr) );
#if ! USE_GETKSIG
#define do_getksig do_unused
```

```
#endif

_PROTOTYPE( int do_endksig, (message *m_ptr) );
#if ! USE_ENDKSIG
#define do_endksig do_unused
#endif

_PROTOTYPE( int do_sigsend, (message *m_ptr) );
#if ! USE_SIGSEND
#define do_sigsend do_unused
#endif

_PROTOTYPE( int do_sigreturn, (message *m_ptr) );
#if ! USE_SIGRETURN
#define do_sigreturn do_unused
#endif

_PROTOTYPE( int do_times, (message *m_ptr) );
#if ! USE_TIMES
#define do_times do_unused
#endif

_PROTOTYPE( int do_setalarm, (message *m_ptr) );
#if ! USE_SETALARM
#define do_setalarm do_unused
#endif

_PROTOTYPE( int do_iopenable, (message *m_ptr) );

#endif /* SYSTEM_H */
```

```

/* The object file of "table.c" contains most kernel data. Variables that
 * are declared in the *.h files appear with EXTERN in front of them, as in
 *
 *     EXTERN int x;
 *
 * Normally EXTERN is defined as extern, so when they are included in another
 * file, no storage is allocated. If EXTERN were not present, but just say,
 *
 *     int x;
 *
 * then including this file in several source files would cause 'x' to be
 * declared several times. While some linkers accept this, others do not,
 * so they are declared extern when included normally. However, it must be
 * declared for real somewhere. That is done here, by redefining EXTERN as
 * the null string, so that inclusion of all *.h files in table.c actually
 * generates storage for them.
 *
 * Various variables could not be declared EXTERN, but are declared PUBLIC
 * or PRIVATE. The reason for this is that extern variables cannot have a
 * default initialization. If such variables are shared, they must also be
 * declared in one of the *.h files without the initialization. Examples
 * include 'boot_image' (this file) and 'idt' and 'gdt' (protect.c).
 *
 * Changes:
 *     Aug 02, 2005    set privileges and minimal boot image (Jorrit N. Herder)
 *     Oct 17, 2004    updated above and tasktab comments (Jorrit N. Herder)
 *     May 01, 2004    changed struct for system image (Jorrit N. Herder)
 */
#define _TABLE

#include "kernel.h"
#include "proc.h"
#include "ipc.h"
#include <minix/com.h>
#include <ibm/int86.h>

/* Define stack sizes for the kernel tasks included in the system image. */
#define NO_STACK      0
#define SMALL_STACK   (128 * sizeof(char *))
#define IDL_S         SMALL_STACK      /* 3 intr, 3 temps, 4 db for Intel */
#define HRD_S         NO_STACK         /* dummy task, uses kernel stack */
#define TSK_S         SMALL_STACK      /* system and clock task */

/* Stack space for all the task stacks. Declared as (char *) to align it. */
#define TOT_STACK_SPACE (IDL_S + HRD_S + (2 * TSK_S))
PUBLIC char *t_stack[TOT_STACK_SPACE / sizeof(char *)];

/* Define flags for the various process types. */
#define IDL_F (SYS_PROC | PREEMPTIBLE | BILLABLE) /* idle task */
#define TSK_F (SYS_PROC) /* kernel tasks */
#define SRV_F (SYS_PROC | PREEMPTIBLE) /* system services */
#define USR_F (BILLABLE | PREEMPTIBLE) /* user processes */

/* Define system call traps for the various process types. These call masks
 * determine what system call traps a process is allowed to make.
 */
#define TSK_T (1 << RECEIVE) /* clock and system */
#define SRV_T (~0) /* system services */
#define USR_T ((1 << SENDREC) | (1 << ECHO)) /* user processes */

/* Send masks determine to whom processes can send messages or notifications.
 * The values here are used for the processes in the boot image. We rely on
 * the initialization code in main() to match the s_nr_to_id() mapping for the
 * processes in the boot image, so that the send mask that is defined here
 * can be directly copied onto map[0] of the actual send mask. Privilege
 * structure 0 is shared by user processes.
 */
#define s(n) (1 << s_nr_to_id(n))
#define SRV_M (~0)
#define SYS_M (~0)
#define USR_M (s(PM_PROC_NR) | s(FS_PROC_NR) | s(RS_PROC_NR) | s(SYSTEM))
#define DRV_M (USR_M | s(SYSTEM) | s(CLOCK) | s(DS_PROC_NR) | s(LOG_PROC_NR) | s(TTY_PRO
_NNR))

```

```

/* Define kernel calls that processes are allowed to make. This is not looking
 * very nice, but we need to define the access rights on a per call basis.
 * Note that the reincarnation server has all bits on, because it should
 * be allowed to distribute rights to services that it starts.
 */
#define c(n)      (1 << ((n)-KERNEL_CALL))
#define RS_C      ~0
#define DS_C      ~0
#define PM_C      ~(c(SYS_DEVIO) | c(SYS_SDEVIO) | c(SYS_VDEVIO) | c(SYS_IRQCTL) | c(SYS_INT86))
#define FS_C      (c(SYS_KILL) | c(SYS_VIRCOPY) | c(SYS_VIRVCOPY) | c(SYS_UMAP) | c(SYS_GETINFO) | c(SYS_EXIT) | c(SYS_TIMES) | c(SYS_SETALARM) | c(SYS_TRACE))
#define DRV_C      (FS_C | c(SYS_SEGCTL) | c(SYS_IRQCTL) | c(SYS_INT86) | c(SYS_DEVIO) | c(SYS_SDEVIO) | c(SYS_VDEVIO))
#define TTY_C      (DRV_C | c(SYS_ABORT) | c(SYS_VM_MAP) | c(SYS_IOPENABLE))
#define MEM_C      (DRV_C | c(SYS_PHYSCOPY) | c(SYS_PHYSVCOPY) | c(SYS_VM_MAP) | \
                    c(SYS_IOPENABLE))

/* The system image table lists all programs that are part of the boot image.
 * The order of the entries here MUST agree with the order of the programs
 * in the boot image and all kernel tasks must come first.
 *
 * Each entry provides the process number, flags, quantum size, scheduling
 * queue, allowed traps, ipc mask, and a name for the process table. The
 * initial program counter and stack size is also provided for kernel tasks.
 *
 * Note: the quantum size must be positive in all cases!
 */
PUBLIC struct boot_image image[] = {
/* process nr,  pc, flags, qs,  queue, stack, traps, ipcto, call,  name */
{ IDLE, idle_task, IDL_F,  8, IDLE_Q, IDL_S,   0,   0,   0, "idle" },
{ CLOCK, clock_task, TSK_F,  8, TASK_Q, TSK_S, TSK_T,   0,   0, "clock" },
{ SYSTEM, sys_task, TSK_F,  8, TASK_Q, TSK_S, TSK_T,   0,   0, "system" },
{ HARDWARE, 0, TSK_F,  8, TASK_Q, HRD_S,   0,   0,   0, "kernel" },
{ PM_PROC_NR, 0, SRV_F, 32,   3, 0,   SRV_T, SRV_M, PM_C, "pm" },
{ FS_PROC_NR, 0, SRV_F, 32,   4, 0,   SRV_T, SRV_M, FS_C, "fs" },
{ RS_PROC_NR, 0, SRV_F,  4,   3, 0,   SRV_T, SYS_M, RS_C, "rs" },
{ DS_PROC_NR, 0, SRV_F,  4,   3, 0,   SRV_T, SYS_M, DS_C, "ds" },
{ TTY_PROC_NR, 0, SRV_F,  4,   1, 0,   SRV_T, SYS_M, TTY_C, "tty" },
{ MEM_PROC_NR, 0, SRV_F,  4,   2, 0,   SRV_T, SYS_M, MEM_C, "mem" },
{ LOG_PROC_NR, 0, SRV_F,  4,   2, 0,   SRV_T, SYS_M, DRV_C, "log" },
{ INIT_PROC_NR, 0, USR_F,  8, USER_Q, 0,   USR_T, USR_M,   0, "init" },
};

/* Verify the size of the system image table at compile time. Also verify that
 * the first chunk of the ipc mask has enough bits to accommodate the processes
 * in the image.
 * If a problem is detected, the size of the 'dummy' array will be negative,
 * causing a compile time error. Note that no space is actually allocated
 * because 'dummy' is declared extern.
 */
extern int dummy[(NR_BOOT_PROCS==sizeof(image)/
                 sizeof(struct boot_image))?1:-1];
extern int dummy[(BITCHUNK_BITS > NR_BOOT_PROCS - 1) ? 1 : -1];

```



```

#ifndef TYPE_H
#define TYPE_H

typedef _PROTOTYPE( void task_t, (void) );

/* Process table and system property related types. */
typedef int proc_nr_t; /* process table entry number */
typedef short sys_id_t; /* system process index */
typedef struct { /* bitmap for system indexes */
    bithunk_t chunk[BITMAP_CHUNKS(NR_SYS_PROCS)];
} sys_map_t;

struct boot_image {
    proc_nr_t proc_nr; /* process number to use */
    task_t *initial_pc; /* start function for tasks */
    int flags; /* process flags */
    unsigned char quantum; /* quantum (tick count) */
    int priority; /* scheduling priority */
    int stksize; /* stack size for tasks */
    short trap_mask; /* allowed system call traps */
    bithunk_t ipc_to; /* send mask protection */
    long call_mask; /* system call protection */
    char proc_name[P_NAME_LEN]; /* name in process table */
    int endpoint; /* endpoint number when started */
};

struct memory {
    phys_clicks base; /* start address of chunk */
    phys_clicks size; /* size of memory chunk */
};

/* The kernel outputs diagnostic messages in a circular buffer. */
struct kmessages {
    int km_next; /* next index to write */
    int km_size; /* current size in buffer */
    char km_buf[KMESS_BUF_SIZE]; /* buffer for messages */
};

struct randomness {
    struct {
        int r_next; /* next index to write */
        int r_size; /* number of random elements */
        unsigned short r_buf[RANDOM_ELEMENTS]; /* buffer for random info */
    } bin[RANDOM_SOURCES];
};

#if (CHIP == INTEL)
typedef unsigned reg_t; /* machine register */

/* The stack frame layout is determined by the software, but for efficiency
 * it is laid out so the assembly code to use it is as simple as possible.
 * 80286 protected mode and all real modes use the same frame, built with
 * 16-bit registers. Real mode lacks an automatic stack switch, so little
 * is lost by using the 286 frame for it. The 386 frame differs only in
 * having 32-bit registers and more segment registers. The same names are
 * used for the larger registers to avoid differences in the code.
 */
struct stackframe_s { /* proc_ptr points here */
#if _WORD_SIZE == 4
    ul6_t gs; /* last item pushed by save */
    ul6_t fs; /* ^ */
#endif
    ul6_t es; /* | */
    ul6_t ds; /* | */
    reg_t di; /* di through cx are not accessed in C */
    reg_t si; /* order is to match pusha/popa */
    reg_t fp; /* bp */
    reg_t st; /* hole for another copy of sp */
    reg_t bx; /* | */
    reg_t dx; /* | */
    reg_t cx; /* | */
    reg_t retreg; /* ax and above are all pushed by save */
    reg_t retadr; /* return address for assembly code save() */
    reg_t pc; /* ^ last item pushed by interrupt */

```

```
reg_t cs;          /* | */
reg_t psw;         /* | */
reg_t sp;          /* | */
reg_t ss;          /* these are pushed by CPU during interrupt */
};

struct segdesc_s { /* segment descriptor for protected mode */
    u16_t limit_low;
    u16_t base_low;
    u8_t base_middle;
    u8_t access;    /* |P|DL|1|X|E|R|A| */
    u8_t granularity; /* |G|X|0|A|LIMT| */
    u8_t base_high;
};

typedef unsigned long irq_policy_t;
typedef unsigned long irq_id_t;

typedef struct irq_hook {
    struct irq_hook *next; /* next hook in chain */
    int (*handler)(struct irq_hook *); /* interrupt handler */
    int irq; /* IRQ vector number */
    int id; /* id of this hook */
    int proc_nr_e; /* (endpoint) NONE if not in use */
    irq_id_t notify_id; /* id to return on interrupt */
    irq_policy_t policy; /* bit mask for policy */
} irq_hook_t;

typedef int (*irq_handler_t)(struct irq_hook *);

#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)
/* M68000 specific types go here. */
#endif /* (CHIP == M68000) */

#endif /* TYPE_H */
```

```
/* This file contains a collection of miscellaneous procedures:
 *   panic:          abort MINIX due to a fatal error
 */

#include "kernel.h"
#include <unistd.h>

/*=====
 *                               panic                               *
 *=====*/

PUBLIC void panic(mess,nr)
_CONST char *mess;
int nr;
{
/* The system has run aground of a fatal kernel error. Terminate execution. */
static int panicking = 0;
if (panicking++) return;          /* prevent recursive panics */

if (mess != NULL) {
    kprintf("\nKernel panic: %s", mess);
    if (nr != NO_NUM) kprintf(" %d", nr);
    kprintf("\n",NO_NUM);
}

/* Abort MINIX. */
prepare_shutdown(RBT_PANIC);
}
```

```
# Makefile for system library implementation

# Directories
u = /usr
i = $u/include

# Programs, flags, etc.
CC =      exec cc $(CFLAGS) -c
CPP =    $1/cpp
LD =     $(CC) -.o
CFLAGS = -I$i
LDFLAGS = -i

SYSTEM = ../system.a

# What to make.
all build install: $(SYSTEM)

OBJECTS = \
    $(SYSTEM)(do_unused.o) \
    $(SYSTEM)(do_fork.o) \
    $(SYSTEM)(do_exec.o) \
    $(SYSTEM)(do_newmap.o) \
    $(SYSTEM)(do_exit.o) \
    $(SYSTEM)(do_trace.o) \
    $(SYSTEM)(do_nice.o) \
    $(SYSTEM)(do_times.o) \
    $(SYSTEM)(do_setalarm.o) \
    $(SYSTEM)(do_irqctl.o) \
    $(SYSTEM)(do_devio.o) \
    $(SYSTEM)(do_vdevio.o) \
    $(SYSTEM)(do_int86.o) \
    $(SYSTEM)(do_sdevio.o) \
    $(SYSTEM)(do_copy.o) \
    $(SYSTEM)(do_vcopy.o) \
    $(SYSTEM)(do_umap.o) \
    $(SYSTEM)(do_memset.o) \
    $(SYSTEM)(do_privctl.o) \
    $(SYSTEM)(do_segctl.o) \
    $(SYSTEM)(do_getksig.o) \
    $(SYSTEM)(do_endksig.o) \
    $(SYSTEM)(do_kill.o) \
    $(SYSTEM)(do_sigsend.o) \
    $(SYSTEM)(do_sigreturn.o) \
    $(SYSTEM)(do_abort.o) \
    $(SYSTEM)(do_getinfo.o) \
    $(SYSTEM)(do_iopenable.o) \
    $(SYSTEM)(do_vm.o) \
    $(SYSTEM)(do_vm_setbuf.o) \

$(SYSTEM):      $(OBJECTS)
               aal cr $@ *.o

clean:
               rm -f $(SYSTEM) *.o *~ *.bak

depend:
               /usr/bin/mkdep "$(CC) -E $(CPPFLAGS)" *.c > .depend

# Include generated dependencies.
include .depend

$(SYSTEM)(do_unused.o): do_unused.c
                        $(CC) do_unused.c

$(SYSTEM)(do_fork.o):   do_fork.c
                        $(CC) do_fork.c

$(SYSTEM)(do_exec.o):   do_exec.c
                        $(CC) do_exec.c

$(SYSTEM)(do_newmap.o): do_newmap.c
                        $(CC) do_newmap.c
```

```
$(SYSTEM)(do_exit.o): do_exit.c
$(CC) do_exit.c

$(SYSTEM)(do_trace.o): do_trace.c
$(CC) do_trace.c

$(SYSTEM)(do_nice.o): do_nice.c
$(CC) do_nice.c

$(SYSTEM)(do_times.o): do_times.c
$(CC) do_times.c

$(SYSTEM)(do_setalarm.o): do_setalarm.c
$(CC) do_setalarm.c

$(SYSTEM)(do_irqctl.o): do_irqctl.c
$(CC) do_irqctl.c

$(SYSTEM)(do_devio.o): do_devio.c
$(CC) do_devio.c

$(SYSTEM)(do_sdevio.o): do_sdevio.c
$(CC) do_sdevio.c

$(SYSTEM)(do_vdevio.o): do_vdevio.c
$(CC) do_vdevio.c

$(SYSTEM)(do_int86.o): do_int86.c
$(CC) do_int86.c

$(SYSTEM)(do_copy.o): do_copy.c
$(CC) do_copy.c

$(SYSTEM)(do_vcopy.o): do_vcopy.c
$(CC) do_vcopy.c

$(SYSTEM)(do_umap.o): do_umap.c
$(CC) do_umap.c

$(SYSTEM)(do_memset.o): do_memset.c
$(CC) do_memset.c

$(SYSTEM)(do_getksig.o): do_getksig.c
$(CC) do_getksig.c

$(SYSTEM)(do_endksig.o): do_endksig.c
$(CC) do_endksig.c

$(SYSTEM)(do_kill.o): do_kill.c
$(CC) do_kill.c

$(SYSTEM)(do_sigsend.o): do_sigsend.c
$(CC) do_sigsend.c

$(SYSTEM)(do_sigreturn.o): do_sigreturn.c
$(CC) do_sigreturn.c

$(SYSTEM)(do_getinfo.o): do_getinfo.c
$(CC) do_getinfo.c

$(SYSTEM)(do_abort.o): do_abort.c
$(CC) do_abort.c

$(SYSTEM)(do_privctl.o): do_privctl.c
$(CC) do_privctl.c

$(SYSTEM)(do_segctl.o): do_segctl.c
$(CC) do_segctl.c

$(SYSTEM)(do_iopenable.o): do_iopenable.c
$(CC) do_iopenable.c

$(SYSTEM)(do_vm.o): do_vm.o
```

```
do_vm.o:      do_vm.c
      $(CC) do_vm.c

$(SYSTEM)(do_vm_setbuf.o):      do_vm_setbuf.c
      $(CC) do_vm_setbuf.c
```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_ABORT
 *
 * The parameters for this kernel call are:
 *   m1_i1:      ABRT_HOW          (how to abort, possibly fetch monitor params)
 *   m1_i2:      ABRT_MON_PROC    (proc nr to get monitor params from)
 *   m1_i3:      ABRT_MON_LEN     (length of monitor params)
 *   m1_p1:      ABRT_MON_ADDR    (virtual address of params)
 */

#include "../system.h"
#include <unistd.h>

#ifdef USE_ABORT

/*=====
 *                               do_abort                               *
 *=====*/
PUBLIC int do_abort(m_ptr)
message *m_ptr;                /* pointer to request message */
{
    /* Handle sys_abort. MINIX is unable to continue. This can originate e.g.
     * in the PM (normal abort or panic) or TTY (after CTRL-ALT-DEL).
     */
    int how = m_ptr->ABRT_HOW;
    int proc_nr;
    int length;
    phys_bytes src_phys;

    /* See if the monitor is to run the specified instructions. */
    if (how == RBT_MONITOR) {

        if (!isokendpt(m_ptr->ABRT_MON_ENDPT, &proc_nr)) return(EDEADSRCDST);
        length = m_ptr->ABRT_MON_LEN + 1;
        if (length > kinfo.params_size) return(E2BIG);
        src_phys = numap_local(proc_nr, (vir_bytes)m_ptr->ABRT_MON_ADDR, length);
        if (!src_phys) return(EFAULT);

        /* Parameters seem ok, copy them and prepare shutting down. */
        phys_copy(src_phys, kinfo.params_base, (phys_bytes) length);
    }

    /* Now prepare to shutdown MINIX. */
    prepare_shutdown(how);
    return(OK);                /* pro-forma (really EDISASTER) */
}

#endif /* USE_ABORT */

```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_VIRCOPY, SYS_PHYSCOPY
 *
 * The parameters for this kernel call are:
 *   m5_c1:       CP_SRC_SPACE           source virtual segment
 *   m5_l1:       CP_SRC_ADDR           source offset within segment
 *   m5_i1:       CP_SRC_PROC_NR       source process number
 *   m5_c2:       CP_DST_SPACE           destination virtual segment
 *   m5_l2:       CP_DST_ADDR           destination offset within segment
 *   m5_i2:       CP_DST_PROC_NR       destination process number
 *   m5_l3:       CP_NR_BYTES           number of bytes to copy
 */

#include "../system.h"
#include <minix/type.h>

#if (USE_VIRCOPY || USE_PHYSCOPY)

/*=====
 *                               do_copy                               *
 *=====*/
PUBLIC int do_copy(m_ptr)
register message *m_ptr;          /* pointer to request message */
{
/* Handle sys_vircopy() and sys_physcopy(). Copy data using virtual or
 * physical addressing. Although a single handler function is used, there
 * are two different kernel calls so that permissions can be checked.
 */
    struct vir_addr vir_addr[2]; /* virtual source and destination address */
    phys_bytes bytes;           /* number of bytes to copy */
    int i;

    /* Dismember the command message. */
    vir_addr[_SRC_].proc_nr_e = m_ptr->CP_SRC_ENDPT;
    vir_addr[_SRC_].segment = m_ptr->CP_SRC_SPACE;
    vir_addr[_SRC_].offset = (vir_bytes) m_ptr->CP_SRC_ADDR;
    vir_addr[_DST_].proc_nr_e = m_ptr->CP_DST_ENDPT;
    vir_addr[_DST_].segment = m_ptr->CP_DST_SPACE;
    vir_addr[_DST_].offset = (vir_bytes) m_ptr->CP_DST_ADDR;
    bytes = (phys_bytes) m_ptr->CP_NR_BYTES;

    /* Now do some checks for both the source and destination virtual address.
     * This is done once for _SRC_, then once for _DST_.
     */
    for (i=_SRC_; i<=_DST_; i++) {
        int p;
        /* Check if process number was given implicitly with SELF and is valid. */
        if (vir_addr[i].proc_nr_e == SELF)
            vir_addr[i].proc_nr_e = m_ptr->m_source;
        if (vir_addr[i].segment != PHYS_SEG &&
            !isokendpt(vir_addr[i].proc_nr_e, &p))
            return(EINVAL);

        /* Check if physical addressing is used without SYS_PHYSCOPY. */
        if ((vir_addr[i].segment & PHYS_SEG) &&
            m_ptr->m_type != SYS_PHYSCOPY) return(EPERM);
    }

    /* Check for overflow. This would happen for 64K segments and 16-bit
     * vir_bytes. Especially copying by the PM on do_fork() is affected.
     */
    if (bytes != (vir_bytes) bytes) return(E2BIG);

    /* Now try to make the actual virtual copy. */
    return( virtual_copy(&vir_addr[_SRC_], &vir_addr[_DST_], bytes) );
}
#endif /* (USE_VIRCOPY || USE_PHYSCOPY) */

```



```

/* The kernel call implemented in this file:
 *   m_type:      SYS_DEVIO
 *
 * The parameters for this kernel call are:
 *   m2_i3:       DIO_REQUEST      (request input or output)
 *   m2_i1:       DIO_TYPE         (flag indicating byte, word, or long)
 *   m2_l1:       DIO_PORT         (port to read/ write)
 *   m2_l2:       DIO_VALUE        (value to write/ return value read)
 */

#include "../system.h"
#include <minix/devio.h>
#include <minix/endpoint.h>

#if USE_DEVIO

/*=====
 *                               do_devio                               *
 *=====*/
PUBLIC int do_devio(m_ptr)
register message *m_ptr;          /* pointer to request message */
{
    struct proc *rp;
    struct priv *privp;
    port_t port;
    struct io_range *iorp;
    int i, size, nr_io_range;

    rp= proc_addr(who_p);
    privp= priv(rp);
    if (!privp)
    {
        kprintf("no priv structure!\n");
        goto doit;
    }
    if (privp->s_flags & CHECK_IO_PORT)
    {
        switch (m_ptr->DIO_TYPE)
        {
            case DIO_BYTE: size= 1; break;
            case DIO_WORD: size= 2; break;
            case DIO_LONG: size= 4; break;
            default: size= 4; break;          /* Be conservative */
        }
        port= m_ptr->DIO_PORT;
        nr_io_range= privp->s_nr_io_range;
        for (i= 0, iorp= privp->s_io_tab; i<nr_io_range; i++, iorp++)
        {
            if (port >= iorp->ior_base && port+size-1 <= iorp->ior_limit)
                break;
        }
        if (i >= nr_io_range)
        {
            kprintf(
                "do_devio: I/O port check failed for proc %d, port 0x%x\n",
                m_ptr->m_source, port);
            return EPERM;
        }
    }
}

doit:

/* Process a single I/O request for byte, word, and long values. */
if (m_ptr->DIO_REQUEST == DIO_INPUT) {
    switch (m_ptr->DIO_TYPE) {
        case DIO_BYTE: m_ptr->DIO_VALUE = inb(m_ptr->DIO_PORT); break;
        case DIO_WORD: m_ptr->DIO_VALUE = inw(m_ptr->DIO_PORT); break;
        case DIO_LONG: m_ptr->DIO_VALUE = inl(m_ptr->DIO_PORT); break;
        default: return(EINVAL);
    }
} else {
    switch (m_ptr->DIO_TYPE) {
        case DIO_BYTE: outb(m_ptr->DIO_PORT, m_ptr->DIO_VALUE); break;
        case DIO_WORD: outw(m_ptr->DIO_PORT, m_ptr->DIO_VALUE); break;

```

```
        case DIO_LONG: outl(m_ptr->DIO_PORT, m_ptr->DIO_VALUE); break;
        default: return(EINVAL);
    }
}
return(OK);
}

#endif /* USE_DEVIO */
```

```

/* The kernel call that is implemented in this file:
 *   m_type:      SYS_ENDKSIG
 *
 * The parameters for this kernel call are:
 *   m2_i1:      SIG_ENDPT      # process for which PM is done
 */

#include "../system.h"
#include <signal.h>
#include <sys/sigcontext.h>

#if USE_ENDKSIG

/*=====
 *                               do_endksig                               *
 *=====*/
PUBLIC int do_endksig(m_ptr)
message *m_ptr;                /* pointer to request message */
{
/* Finish up after a kernel type signal, caused by a SYS_KILL message or a
 * call to cause_sig by a task. This is called by the PM after processing a
 * signal it got with SYS_GETKSIG.
 */
    register struct proc *rp;
    int proc;

    /* Get process pointer and verify that it had signals pending. If the
     * process is already dead its flags will be reset.
     */
    if(!isokendpt(m_ptr->SIG_ENDPT, &proc))
        return EINVAL;

    rp = proc_addr(proc);
    if (! (rp->p_rts_flags & SIG_PENDING)) return(EINVAL);

    /* PM has finished one kernel signal. Perhaps process is ready now? */
    if (! (rp->p_rts_flags & SINGALED))      /* new signal arrived */
        if ((rp->p_rts_flags & ~SIG_PENDING)==0) /* remove pending flag */
            lock_enqueue(rp);                /* ready if no flags */
    return(OK);
}

#endif /* USE_ENDKSIG */

```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_EXEC
 *
 * The parameters for this kernel call are:
 *   ml_i1:       PR_ENDPT      (process that did exec call)
 *   ml_p1:       PR_STACK_PTR  (new stack pointer)
 *   ml_p2:       PR_NAME_PTR   (pointer to program name)
 *   ml_p3:       PR_IP_PTR     (new instruction pointer)
 */
#include "../system.h"
#include <string.h>
#include <signal.h>
#include <minix/endpoint.h>

#if USE_EXEC

/*=====
 *                               do_exec                               *
 *=====*/
PUBLIC int do_exec(m_ptr)
register message *m_ptr;      /* pointer to request message */
{
/* Handle sys_exec().  A process has done a successful EXEC. Patch it up. */
register struct proc *rp;
reg_t sp;                    /* new sp */
phys_bytes phys_name;
char *np;
int proc;

if(!isokendpt(m_ptr->PR_ENDPT, &proc))
    return EINVAL;

rp = proc_addr(proc);
sp = (reg_t) m_ptr->PR_STACK_PTR;
rp->p_reg.sp = sp;            /* set the stack pointer */
#if (CHIP == M68000)
rp->p_splow = sp;             /* set the stack pointer low water */
#endif
#ifdef FPP
/* Initialize fpp for this process */
fpp_new_state(rp);
#endif
#endif
/* wipe extra LDT entries */
phys_memset(vir2phys(&rp->p_ldt[EXTRA_LDT_INDEX]), 0,
            (LDT_SIZE - EXTRA_LDT_INDEX) * sizeof(rp->p_ldt[0]));
#endif
rp->p_reg.pc = (reg_t) m_ptr->PR_IP_PTR;      /* set pc */
rp->p_rts_flags &= ~RECEIVING;                /* PM does not reply to EXEC call */
if (rp->p_rts_flags == 0) lock_enqueue(rp);
/* Save command name for debugging, ps(1) output, etc. */
phys_name = numap_local(who_p, (vir_bytes) m_ptr->PR_NAME_PTR,
                        (vir_bytes) P_NAME_LEN - 1);

if (phys_name != 0) {
    phys_copy(phys_name, vir2phys(rp->p_name), (phys_bytes) P_NAME_LEN - 1);
    for (np = rp->p_name; (*np & BYTE) >= ' '; np++) {}
    *np = 0;                                /* mark end */
} else {
    strncpy(rp->p_name, "<unset>", P_NAME_LEN);
}
return(OK);
}
#endif /* USE_EXEC */

```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_EXIT
 *
 * The parameters for this kernel call are:
 *   ml_il:       PR_ENDPT          (slot number of exiting process)
 */

#include "../system.h"

#include <minix/endpoint.h>

#if USE_EXIT

FORWARD _PROTOTYPE( void clear_proc, (register struct proc *rc));

/*=====
 *                               do_exit                               *
 *=====*/
PUBLIC int do_exit(m_ptr)
message *m_ptr;                /* pointer to request message */
{
    /* Handle sys_exit. A user process has exited or a system process requests
     * to exit. Only the PM can request other process slots to be cleared.
     * The routine to clean up a process table slot cancels outstanding timers,
     * possibly removes the process from the message queues, and resets certain
     * process table fields to the default values.
     */
    int exit_e;

    /* Determine what process exited. User processes are handled here. */
    if (PM_PROC_NR == who_p) {
        if (m_ptr->PR_ENDPT != SELF) {                /* PM tries to exit self */
            if(!isokendpt(m_ptr->PR_ENDPT, &exit_e)) /* get exiting process */
                return EINVAL;
            clear_proc(proc_addr(exit_e));            /* exit a user process */
            return(OK);                                /* report back to PM */
        }
    }

    /* The PM or some other system process requested to be exited. */
    clear_proc(proc_addr(who_p));
    return(EDONTREPLY);
}

/*=====
 *                               clear_proc                             *
 *=====*/
PRIVATE void clear_proc(rc)
register struct proc *rc;        /* slot of process to clean up */
{
    register struct proc *rp;    /* iterate over process table */
    register struct proc **xpp; /* iterate over caller queue */
    int i;
    int sys_id;
    char saved_rts_flags;

    /* Don't clear if already cleared. */
    if(isempty(rc)) return;

    /* Remove the process' ability to send and receive messages */
    clear_endpoint(rc);

    /* Turn off any alarm timers at the clock. */
    reset_timer(&priv(rc)->s_alarm_timer);

    /* Make sure that the exiting process is no longer scheduled. */
    if (rc->p_rts_flags == 0) lock_dequeue(rc);

    /* Check the table with IRQ hooks to see if hooks should be released. */
    for (i=0; i < NR_IRQ_HOOKS; i++) {
        int proc;
        if (rc->p_endpoint == irq_hooks[i].proc_nr_e) {
            rm_irq_handler(&irq_hooks[i]); /* remove interrupt handler */
            irq_hooks[i].proc_nr_e = NONE; /* mark hook as free */
        }
    }
}

```

```
    }  
}  
  
/* Release the process table slot. If this is a system process, also  
 * release its privilege structure. Further cleanup is not needed at  
 * this point. All important fields are reinitialized when the  
 * slots are assigned to another, new process.  
 */  
saved_rts_flags = rc->p_rts_flags;  
rc->p_rts_flags = SLOT_FREE;  
if (priv(rc)->s_flags & SYS_PROC) priv(rc)->s_proc_nr = NONE;  
  
/* Clean up virtual memory */  
if (rc->p_misc_flags & MF_VM)  
    vm_map_default(rc);  
}  
  
#endif /* USE_EXIT */
```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_FORK
 *
 * The parameters for this kernel call are:
 *   ml_i1:      PR_SLOT  (child's process table slot)
 *   ml_i2:      PR_ENDPT (parent, process that forked)
 */

#include "../system.h"
#include <signal.h>
#if (CHIP == INTEL)
#include "../protect.h"
#endif

#include <minix/endpoint.h>

#if USE_FORK

/*=====
 *                               do_fork                               *
 *=====*/
PUBLIC int do_fork(m_ptr)
register message *m_ptr;          /* pointer to request message */
{
/* Handle sys_fork(). PR_ENDPT has forked. The child is PR_SLOT. */
#if (CHIP == INTEL)
    reg_t old_ldt_sel;
#endif
    register struct proc *rpc;          /* child process pointer */
    struct proc *rpp;                  /* parent process pointer */
    struct mem_map *map_ptr;           /* virtual address of map inside caller (PM) */
    int i, gen;
    int p_proc;

    if(!isokendpt(m_ptr->PR_ENDPT, &p_proc))
        return EINVAL;
    rpp = proc_addr(p_proc);
    rpc = proc_addr(m_ptr->PR_SLOT);
    if (isempty(rpp) || ! isempty(rpc)) return(EINVAL);

    map_ptr= (struct mem_map *) m_ptr->PR_MEM_PTR;

    /* Copy parent 'proc' struct to child. And reinitialize some fields. */
    gen = _ENDPOINT_G(rpc->p_endpoint);
    #if (CHIP == INTEL)
        old_ldt_sel = rpc->p_ldt_sel;          /* backup local descriptors */
        *rpc = *rpp;                          /* copy 'proc' struct */
        rpc->p_ldt_sel = old_ldt_sel;          /* restore descriptors */
    #else
        *rpc = *rpp;                          /* copy 'proc' struct */
    #endif
    #endif
    if(++gen >= _ENDPOINT_MAX_GENERATION) /* increase generation */
        gen = 1;                          /* generation number wraparound */
    rpc->p_nr = m_ptr->PR_SLOT;              /* this was obliterated by copy */
    rpc->p_endpoint = _ENDPOINT(gen, rpc->p_nr); /* new endpoint of slot */

    /* Only one in group should have SINGALED, child doesn't inherit tracing. */
    rpc->p_rts_flags &= ~(SINGALED | SIG_PENDING | P_STOP);
    sigemptyset(&rpc->p_pending);

    rpc->p_reg.retreg = 0;                  /* child sees pid = 0 to know it is child */
    rpc->p_user_time = 0;                   /* set all the accounting times to 0 */
    rpc->p_sys_time = 0;

    /* Parent and child have to share the quantum that the forked process had,
     * so that queued processes do not have to wait longer because of the fork.
     * If the time left is odd, the child gets an extra tick.
     */
    rpc->p_ticks_left = (rpc->p_ticks_left + 1) / 2;
    rpp->p_ticks_left = rpp->p_ticks_left / 2;

    /* If the parent is a privileged process, take away the privileges from the
     * child process and inhibit it from running by setting the NO_PRIV flag.
     * The caller should explicitly set the new privileges before executing.

```

```
    */
    if (priv(rpp)->s_flags & SYS_PROC) {
        rpc->p_priv = priv_addr(USER_PRIV_ID);
        rpc->p_rts_flags |= NO_PRIV;
    }

    /* Calculate endpoint identifier, so caller knows what it is. */
    m_ptr->PR_ENDPT = rpc->p_endpoint;

    /* Install new map */
    return newmap(rpc, map_ptr);
}

#endif /* USE_FORK */
```



```

/* The kernel call implemented in this file:
 *   m_type:      SYS_GETINFO
 *
 * The parameters for this kernel call are:
 *   ml_i3:       I_REQUEST      (what info to get)
 *   ml_p1:       I_VAL_PTR      (where to put it)
 *   ml_i1:       I_VAL_LEN      (maximum length expected, optional)
 *   ml_p2:       I_VAL_PTR2     (second, optional pointer)
 *   ml_i2:       I_VAL_LEN2_E   (second length or process nr)
 */

#include "../system.h"

static unsigned long bios_buf[1024]; /* 4K, what about alignment */
static vir_bytes bios_buf_vir, bios_buf_len;

#if USE_GETINFO

/*=====
 *                               do_getinfo                               *
 *=====*/
PUBLIC int do_getinfo(m_ptr)
register message *m_ptr; /* pointer to request message */
{
/* Request system information to be copied to caller's address space. This
 * call simply copies entire data structures to the caller.
 */
    size_t length;
    phys_bytes src_phys;
    phys_bytes dst_phys;
    int proc_nr, nr_e, nr;

    /* Set source address and length based on request type. */
    switch (m_ptr->I_REQUEST) {
        case GET_MACHINE: {
            length = sizeof(struct machine);
            src_phys = vir2phys(&machine);
            break;
        }
        case GET_KINFO: {
            length = sizeof(struct kinfo);
            src_phys = vir2phys(&kinfo);
            break;
        }
        case GET_LOADINFO: {
            length = sizeof(struct loadinfo);
            src_phys = vir2phys(&kloadinfo);
            break;
        }
        case GET_IMAGE: {
            length = sizeof(struct boot_image) * NR_BOOT_PROCS;
            src_phys = vir2phys(image);
            break;
        }
        case GET_IRQHOOKS: {
            length = sizeof(struct irq_hook) * NR_IRQ_HOOKS;
            src_phys = vir2phys(irq_hooks);
            break;
        }
        case GET_SCHEDINFO: {
            /* This is slightly complicated because we need two data structures
             * at once, otherwise the scheduling information may be incorrect.
             * Copy the queue heads and fall through to copy the process table.
             */
            length = sizeof(struct proc *) * NR_SCHED_QUEUES;
            src_phys = vir2phys(rdy_head);
            okendpt(m_ptr->m_source, &proc_nr);
            dst_phys = numap_local(proc_nr, (vir_bytes) m_ptr->I_VAL_PTR2,
                                   length);
            if (src_phys == 0 || dst_phys == 0) return(EFAULT);
            phys_copy(src_phys, dst_phys, length);
            /* fall through */
        }
        case GET_PROCTAB: {

```

```

    length = sizeof(struct proc) * (NR_PROCS + NR_TASKS);
    src_phys = vir2phys(proc);
    break;
}
case GET_PRIVTAB: {
    length = sizeof(struct priv) * (NR_SYS_PROCS);
    src_phys = vir2phys(priv);
    break;
}
case GET_PROC: {
    nr_e = (m_ptr->I_VAL_LEN2_E == SELF) ?
            m_ptr->m_source : m_ptr->I_VAL_LEN2_E;
    if(!isokendpt(nr_e, &nr)) return EINVAL; /* validate request */
    length = sizeof(struct proc);
    src_phys = vir2phys(proc_addr(nr));
    break;
}
case GET_MONPARAMS: {
    src_phys = kinfo.params_base;          /* already is a physical */
    length = kinfo.params_size;
    break;
}
case GET_RANDOMNESS: {
    static struct randomness copy;          /* copy to keep counters */
    int i;

    copy = krandom;
    for (i= 0; i<RANDOM_SOURCES; i++) {
        krandom.bin[i].r_size = 0;          /* invalidate random data */
        krandom.bin[i].r_next = 0;
    }
    length = sizeof(struct randomness);
    src_phys = vir2phys(&copy);
    break;
}
case GET_KMESSAGES: {
    length = sizeof(struct kmessages);
    src_phys = vir2phys(&kmess);
    break;
}
#if DEBUG_TIME_LOCKS
case GET_LOCKTIMING: {
    length = sizeof(timingdata);
    src_phys = vir2phys(timingdata);
    break;
}
#endif
case GET_BIOSBUFFER:
    bios_buf_vir = (vir_bytes)bios_buf;
    bios_buf_len = sizeof(bios_buf);

    length = sizeof(bios_buf_len);
    src_phys = vir2phys(&bios_buf_len);
    if (length != m_ptr->I_VAL_LEN2_E) return (EINVAL);
    if(!isokendpt(m_ptr->m_source, &proc_nr))
        panic("bogus source", m_ptr->m_source);
    dst_phys = numap_local(proc_nr, (vir_bytes) m_ptr->I_VAL_PTR2, length);
    if (src_phys == 0 || dst_phys == 0) return(EFAULT);
    phys_copy(src_phys, dst_phys, length);

    length = sizeof(bios_buf_vir);
    src_phys = vir2phys(&bios_buf_vir);
    break;

case GET_IRQACTIDS: {
    length = sizeof(irq_actids);
    src_phys = vir2phys(irq_actids);
    break;
}

default:
    return(EINVAL);
}

```

```
/* Try to make the actual copy for the requested data. */
if (m_ptr->I_VAL_LEN > 0 && length > m_ptr->I_VAL_LEN) return (E2BIG);
if(!isokendpt(m_ptr->m_source, &proc_nr))
    panic("bogus source", m_ptr->m_source);
dst_phys = numap_local(proc_nr, (vir_bytes) m_ptr->I_VAL_PTR, length);
if (src_phys == 0 || dst_phys == 0) return(EFAULT);
phys_copy(src_phys, dst_phys, length);
return(OK);
}

#endif /* USE_GETINFO */
```

```

/* The kernel call that is implemented in this file:
 *   m_type:      SYS_GETKSIG
 *
 * The parameters for this kernel call are:
 *   m2_i1:      SIG_ENDPT      # process with pending signals
 *   m2_l1:      SIG_MAP        # bit map with pending signals
 */

#include "../system.h"
#include <signal.h>
#include <sys/sigcontext.h>
#include <minix/endpoint.h>

#if USE_GETKSIG

/*=====
 *                               do_getksig                               *
 *=====*/
PUBLIC int do_getksig(m_ptr)
message *m_ptr;                /* pointer to request message */
{
    /* PM is ready to accept signals and repeatedly does a kernel call to get
     * one. Find a process with pending signals. If no signals are available,
     * return NONE in the process number field.
     * It is not sufficient to ready the process when PM is informed, because
     * PM can block waiting for FS to do a core dump.
     */
    register struct proc *rp;

    /* Find the next process with pending signals. */
    for (rp = BEG_USER_ADDR; rp < END_PROC_ADDR; rp++) {
        if (rp->p_rts_flags & SIGNED) {
            /* store signaled process' endpoint */
            m_ptr->SIG_ENDPT = rp->p_endpoint;
            m_ptr->SIG_MAP = rp->p_pending;          /* pending signals map */
            sigemptyset(&rp->p_pending);           /* ball is in PM's court */
            rp->p_rts_flags &= ~SIGNED;            /* blocked by SIG_PENDING */
            return(OK);
        }
    }

    /* No process with pending signals was found. */
    m_ptr->SIG_ENDPT = NONE;
    return(OK);
}
#endif /* USE_GETKSIG */

```

```
/* The kernel call implemented in this file:
 *   m_type:      SYS_INT86
 *
 * The parameters for this kernel call are:
 *   ml_p1:      INT86_REG86
 */

#include "../system.h"
#include <minix/type.h>
#include <minix/endpoint.h>
#include <ibm/int86.h>

struct reg86u reg86;

/*=====
 *                               do_int86                               *
 *=====*/
PUBLIC int do_int86(m_ptr)
register message *m_ptr;      /* pointer to request message */
{
    vir_bytes caller_vir;
    phys_bytes caller_phys, kernel_phys;

    caller_vir = (vir_bytes) m_ptr->INT86_REG86;
    caller_phys = umap_local(proc_addr(who_p), D, caller_vir, sizeof(reg86));
    if (0 == caller_phys) return(EFAULT);
    kernel_phys = vir2phys(&reg86);
    phys_copy(caller_phys, kernel_phys, (phys_bytes) sizeof(reg86));

    level0(int86);

    /* Copy results back to the caller */
    phys_copy(kernel_phys, caller_phys, (phys_bytes) sizeof(reg86));

    /* The BIOS call eats interrupts. Call get_randomness to generate some
     * entropy. Normally, get_randomness is called from an interrupt handler.
     * Figuring out the exact source is too complicated. CLOCK_IRQ is normally
     * not very random.
     */
    lock(0, "do_int86");
    get_randomness(CLOCK_IRQ);
    unlock(0);

    return(OK);
}
```

```
/* The system call implemented in this file:
 *   m_type:      SYS_IOPENABLE
 *
 * The parameters for this system call are:
 *   m2_i2:      IO_ENDPT      (process to give I/O Protection Level bits)
 *
 * Author:
 *   Jorrit N. Herder <jnherder@cs.vu.nl>
 */

#include "../system.h"
#include "../kernel.h"

/*=====*
 *                               do_iopenable                               *
 *=====*/
PUBLIC int do_iopenable(m_ptr)
register message *m_ptr;      /* pointer to request message */
{
    int proc_nr;

    #if 1 /* ENABLE_USERPRIV && ENABLE_USERIOPL */
        if (m_ptr->IO_ENDPT == SELF) {
            proc_nr = who_p;
        } else if (!isokendpt(m_ptr->IO_ENDPT, &proc_nr))
            return(EINVAL);
        enable_iop(proc_addr(proc_nr));
        return(OK);
    #else
        return(EPERM);
    #endif
}
```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_IRQCTL
 *
 * The parameters for this kernel call are:
 *   m5_c1:      IRQ_REQUEST      (control operation to perform)
 *   m5_c2:      IRQ_VECTOR       (irq line that must be controlled)
 *   m5_i1:      IRQ_POLICY       (irq policy allows reenabling interrupts)
 *   m5_l3:      IRQ_HOOK_ID      (provides index to be returned on interrupt)
 *   ' ' ' ' ' ' ' ' ' ' ' '    (returns index of irq hook assigned at kernel)
 */

#include "../system.h"

#include <minix/endpoint.h>

#if USE_IRQCTL

FORWARD _PROTOTYPE(int generic_handler, (irq_hook_t *hook));

/*=====
 *                               do_irqctl                               *
 *=====*/
PUBLIC int do_irqctl(m_ptr)
register message *m_ptr;          /* pointer to request message */
{
    /* Dismember the request message. */
    int irq_vec;
    int irq_hook_id;
    int notify_id;
    int r = OK;
    int i;
    irq_hook_t *hook_ptr;
    struct proc *rp;
    struct priv *privp;

    /* Hook identifiers start at 1 and end at NR_IRQ_HOOKS. */
    irq_hook_id = (unsigned) m_ptr->IRQ_HOOK_ID - 1;
    irq_vec = (unsigned) m_ptr->IRQ_VECTOR;

    /* See what is requested and take needed actions. */
    switch(m_ptr->IRQ_REQUEST) {

    /* Enable or disable IRQs. This is straightforward. */
    case IRQ_ENABLE:
    case IRQ_DISABLE:
        if (irq_hook_id >= NR_IRQ_HOOKS || irq_hook_id < 0 ||
            irq_hooks[irq_hook_id].proc_nr_e == NONE) return(EINVAL);
        if (irq_hooks[irq_hook_id].proc_nr_e != m_ptr->m_source) return(EPERM);
        if (m_ptr->IRQ_REQUEST == IRQ_ENABLE)
            enable_irq(&irq_hooks[irq_hook_id]);
        else
            disable_irq(&irq_hooks[irq_hook_id]);
        break;

    /* Control IRQ policies. Set a policy and needed details in the IRQ table.
     * This policy is used by a generic function to handle hardware interrupts.
     */
    case IRQ_SETPOLICY:

        /* Check if IRQ line is acceptable. */
        if (irq_vec < 0 || irq_vec >= NR_IRQ_VECTORS) return(EINVAL);

        rp= proc_addr(who_p);
        privp= priv(rp);
        if (!privp)
        {
            kprintf("no priv structure!\n");
            return EPERM;
        }
        if (privp->s_flags & CHECK_IRQ)
        {
            for (i= 0; i<privp->s_nr_irq; i++)
            {
                if (irq_vec == privp->s_irq_tab[i])

```

```

        break;
    }
    if (i >= privp->s_nr_irq)
    {
        kprintf(
            "do_irqctl: IRQ check failed for proc %d, IRQ %d\n",
            m_ptr->m_source, irq_vec);
        return EPERM;
    }
}

/* Find a free IRQ hook for this mapping. */
hook_ptr = NULL;
for (irq_hook_id=0; irq_hook_id<NR_IRQ_HOOKS; irq_hook_id++) {
    if (irq_hooks[irq_hook_id].proc_nr_e == NONE) {
        hook_ptr = &irq_hooks[irq_hook_id];    /* free hook */
        break;
    }
}
if (hook_ptr == NULL) return(ENOSPC);

/* When setting a policy, the caller must provide an identifier that
 * is returned on the notification message if a interrupt occurs.
 */
notify_id = (unsigned) m_ptr->IRQ_HOOK_ID;
if (notify_id > CHAR_BIT * sizeof(irq_id_t) - 1) return(EINVAL);

/* Install the handler. */
hook_ptr->proc_nr_e = m_ptr->m_source;    /* process to notify */
hook_ptr->notify_id = notify_id;        /* identifier to pass */
hook_ptr->policy = m_ptr->IRQ_POLICY;    /* policy for interrupts */
put_irq_handler(hook_ptr, irq_vec, generic_handler);

/* Return index of the IRQ hook in use. */
m_ptr->IRQ_HOOK_ID = irq_hook_id + 1;
break;

case IRQ_RMPOLICY:
    if (irq_hook_id < 0 || irq_hook_id >= NR_IRQ_HOOKS ||
        irq_hooks[irq_hook_id].proc_nr_e == NONE) {
        return(EINVAL);
    } else if (m_ptr->m_source != irq_hooks[irq_hook_id].proc_nr_e) {
        return(EPERM);
    }
    /* Remove the handler and return. */
    rm_irq_handler(&irq_hooks[irq_hook_id]);
    break;

default:
    r = EINVAL;    /* invalid IRQ_REQUEST */
}
return(r);
}

/*=====
 *                                generic_handler                                *
 *=====*/
PRIVATE int generic_handler(hook)
irq_hook_t *hook;
{
    /* This function handles hardware interrupt in a simple and generic way. All
     * interrupts are transformed into messages to a driver. The IRQ line will be
     * reenabled if the policy says so.
     */
    int proc;

    /* As a side-effect, the interrupt handler gathers random information by
     * timestamping the interrupt events. This is used for /dev/random.
     */
    get_randomness(hook->irq);

    /* Check if the handler is still alive. If not, forget about the
     * interrupt. This should never happen, as processes that die
     * automatically get their interrupt hooks unhooked.

```



```
    */
    if(!isokendpt(hook->proc_nr_e, &proc)) {
        hook->proc_nr_e = NONE;
        return 0;
    }

    /* Add a bit for this interrupt to the process' pending interrupts. When
     * sending the notification message, this bit map will be magically set
     * as an argument.
     */
    priv(proc_addr(proc))->s_int_pending |= (1 << hook->notify_id);

    /* Build notification message and return. */
    lock_notify(HARDWARE, hook->proc_nr_e);
    return(hook->policy & IRQ_REENABLE);
}

#endif /* USE_IRQCTL */
```

```

/* The kernel call that is implemented in this file:
 *   m_type:      SYS_KILL
 *
 * The parameters for this kernel call are:
 *   m2_i1:      SIG_ENDPT      # process to signal/ pending
 *   m2_i2:      SIG_NUMBER     # signal number to send to process
 */

#include "../system.h"
#include <signal.h>
#include <sys/sigcontext.h>

#if USE_KILL

/*=====
 *                               do_kill
 *=====*/

PUBLIC int do_kill(m_ptr)
message *m_ptr;                /* pointer to request message */
{
    /* Handle sys_kill(). Cause a signal to be sent to a process. The PM is the
     * central server where all signals are processed and handler policies can
     * be registered. Any request, except for PM requests, is added to the map
     * of pending signals and the PM is informed about the new signal.
     * Since system servers cannot use normal POSIX signal handlers (because they
     * are usually blocked on a RECEIVE), they can request the PM to transform
     * signals into messages. This is done by the PM with a call to sys_kill().
     */
    proc_nr_t proc_nr, proc_nr_e;
    int sig_nr = m_ptr->SIG_NUMBER;

    proc_nr_e = m_ptr->SIG_ENDPT;

    if (proc_nr_e == SELF)
        proc_nr_e = m_ptr->m_source;

    if (!isokendpt(proc_nr_e, &proc_nr)) return(EINVAL);

    if (sig_nr > _NSIG) return(EINVAL);
    if (iskerneln(proc_nr)) return(EPERM);

    /* Set pending signal to be processed by the PM. */
    cause_sig(proc_nr, sig_nr);
    if (sig_nr == SIGKILL)
        clear_endpoint(proc_addr(proc_nr));
    return(OK);
}

#endif /* USE_KILL */

```

```
/* The kernel call implemented in this file:
 *   m_type:      SYS_MEMSET
 *
 * The parameters for this kernel call are:
 *   m2_p1:      MEM_PTR          (virtual address)
 *   m2_l1:      MEM_COUNT        (returns physical address)
 *   m2_l2:      MEM_PATTERN      (size of datastructure)
 */

#include "../system.h"

#if USE_MEMSET

/*=====
 *                               do_memset                               *
 *=====*/
PUBLIC int do_memset(m_ptr)
register message *m_ptr;
{
/* Handle sys_memset(). This writes a pattern into the specified memory. */
  unsigned long p;
  unsigned char c = m_ptr->MEM_PATTERN;
  p = c | (c << 8) | (c << 16) | (c << 24);
  phys_memset((phys_bytes) m_ptr->MEM_PTR, p, (phys_bytes) m_ptr->MEM_COUNT);
  return(OK);
}

#endif /* USE_MEMSET */
```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_NEWMAP
 *
 * The parameters for this kernel call are:
 *   ml_i1:       PR_ENDPT      (install new map for this process)
 *   ml_p1:       PR_MEM_PTR    (pointer to the new memory map)
 */
#include "../system.h"
#include <minix/endpoint.h>

#if USE_NEWMAP

/*=====
 *                               do_newmap                               *
 *=====*/
PUBLIC int do_newmap(m_ptr)
message *m_ptr;                /* pointer to request message */
{
    /* Handle sys_newmap(). Fetch the memory map from PM. */
    register struct proc *rp;   /* process whose map is to be loaded */
    struct mem_map *map_ptr;    /* virtual address of map inside caller (PM) */
    phys_bytes src_phys;        /* physical address of map at the PM */
    int old_flags;              /* value of flags before modification */
    int proc;

    map_ptr = (struct mem_map *) m_ptr->PR_MEM_PTR;
    if (!isokendpt(m_ptr->PR_ENDPT, &proc)) return(EINVAL);
    if (iskerneln(proc)) return(EPERM);
    rp = proc_addr(proc);

    return newmap(rp, map_ptr);
}

/*=====
 *                               newmap                               *
 *=====*/
PUBLIC int newmap(rp, map_ptr)
struct proc *rp;               /* process whose map is to be loaded */
struct mem_map *map_ptr;       /* virtual address of map inside caller (PM) */
{
    /* Fetch the memory map from PM. */
    phys_bytes src_phys;        /* physical address of map at the PM */
    int old_flags;              /* value of flags before modification */
    int proc;

    /* Copy the map from PM. */
    src_phys = umap_local(proc_addr(who_p), D, (vir_bytes) map_ptr,
        sizeof(rp->p_memmap));
    if (src_phys == 0) return(EFAULT);
    phys_copy(src_phys, vir2phys(rp->p_memmap), (phys_bytes) sizeof(rp->p_memmap));

    #if (CHIP != M68000)
        alloc_segments(rp);
    #else
        pmmu_init_proc(rp);
    #endif
    old_flags = rp->p_rts_flags; /* save the previous value of the flags */
    if (old_flags != 0 && rp->p_rts_flags == 0) lock_enqueue(rp);

    return(OK);
}
#endif /* USE_NEWMAP */

```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_NICE
 *
 * The parameters for this kernel call are:
 *   ml_i1:      PR_ENDPT      process number to change priority
 *   ml_i2:      PR_PRIORITY   the new priority
 */

#include "../system.h"
#include <minix/type.h>
#include <sys/resource.h>

#if USE_NICE

/*=====
 *                               do_nice
 *=====*/
PUBLIC int do_nice(message *m_ptr)
{
    /* Change process priority or stop the process. */
    int proc_nr, pri, new_q;
    register struct proc *rp;

    /* Extract the message parameters and do sanity checking. */
    if(!isokendpt(m_ptr->PR_ENDPT, &proc_nr)) return EINVAL;
    if (iskerneln(proc_nr)) return(EPERM);
    pri = m_ptr->PR_PRIORITY;
    rp = proc_addr(proc_nr);

    if (pri == PRIO_STOP) {
        /* Take process off the scheduling queues. */
        lock_dequeue(rp);
        rp->p_rts_flags |= NO_PRIORITY;
        return(OK);
    }
    else if (pri >= PRIO_MIN && pri <= PRIO_MAX) {
        /* The value passed in is currently between PRIO_MIN and PRIO_MAX.
         * We have to scale this between MIN_USER_Q and MAX_USER_Q to match
         * the kernel's scheduling queues.
         */
        new_q = MAX_USER_Q + (pri-PRIO_MIN) * (MIN_USER_Q-MAX_USER_Q+1) /
            (PRIO_MAX-PRIO_MIN+1);
        if (new_q < MAX_USER_Q) new_q = MAX_USER_Q;      /* shouldn't happen */
        if (new_q > MIN_USER_Q) new_q = MIN_USER_Q;      /* shouldn't happen */

        /* Make sure the process is not running while changing its priority.
         * Put the process back in its new queue if it is runnable.
         */
        lock_dequeue(rp);
        rp->p_rts_flags &= ~NO_PRIORITY;
        rp->p_max_priority = rp->p_priority = new_q;
        if (!rp->p_rts_flags) lock_enqueue(rp);

        return(OK);
    }
    return(EINVAL);
}

#endif /* USE_NICE */

```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_PRIVCTL
 *
 * The parameters for this kernel call are:
 *   ml_il:       PR_ENDPT      (process number of caller)
 */

#include "../system.h"
#include "../ipc.h"
#include <signal.h>

#if USE_PRIVCTL

#define FILLED_MASK      (~0)

/*=====
 *                               do_privctl                               *
 *=====*/
PUBLIC int do_privctl(m_ptr)
message *m_ptr;          /* pointer to request message */
{
    /* Handle sys_privctl(). Update a process' privileges. If the process is not
     * yet a system process, make sure it gets its own privilege structure.
     */
    register struct proc *caller_ptr;
    register struct proc *rp;
    register struct priv *sp;
    int proc_nr;
    int priv_id;
    int old_flags;
    int i;
    phys_bytes caller_phys, kernel_phys;
    struct io_range io_range;
    struct mem_range mem_range;

    /* Check whether caller is allowed to make this call. Privileged proceses
     * can only update the privileges of processes that are inhibited from
     * running by the NO_PRIV flag. This flag is set when a privileged process
     * forks.
     */
    caller_ptr = proc_addr(who_p);
    if (! (priv(caller_ptr)->s_flags & SYS_PROC)) return(EPERM);
    if (!isokendpt(m_ptr->PR_ENDPT, &proc_nr)) return(EINVAL);
    rp = proc_addr(proc_nr);

    switch(m_ptr->CTL_REQUEST)
    {
    case SYS_PRIV_INIT:
        if (! (rp->p_rts_flags & NO_PRIV)) return(EPERM);

        /* Make sure this process has its own privileges structure. This may
         * fail, since there are only a limited number of system processes.
         * Then copy the privileges from the caller and restore some defaults.
         */
        if ((i=get_priv(rp, SYS_PROC)) != OK) return(i);
        priv_id = priv(rp)->s_id;          /* backup privilege id */
        *priv(rp) = *priv(caller_ptr);    /* copy from caller */
        priv(rp)->s_id = priv_id;         /* restore privilege id */
        priv(rp)->s_proc_nr = proc_nr;    /* reassociate process nr */

        for (i=0; i< BITMAP_CHUNKS(NR_SYS_PROCS); i++) /* remove pending: */
            priv(rp)->s_notify_pending.chunk[i] = 0; /* - notifications */
        priv(rp)->s_int_pending = 0;          /* - interrupts */
        sigemptyset(&priv(rp)->s_sig_pending); /* - signals */

        /* Now update the process' privileges as requested. */
        rp->p_priv->s_trap_mask = FILLED_MASK;
        for (i=0; i<BITMAP_CHUNKS(NR_SYS_PROCS); i++) {
            rp->p_priv->s_ipc_to.chunk[i] = FILLED_MASK;
        }
        unset_sys_bit(rp->p_priv->s_ipc_to, USER_PRIV_ID);

        /* All process that this process can send to must be able to reply.
         * Therefore, their send masks should be updated as well.

```

```

    */
    for (i=0; i<NR_SYS_PROCS; i++) {
        if (get_sys_bit(rp->p_priv->s_ipc_to, i)) {
            set_sys_bit(priv_addr(i)->s_ipc_to, priv_id(rp));
        }
    }

    /* No I/O resources, no memory resources, no IRQs */
    priv(rp)->s_nr_io_range= 0;
    priv(rp)->s_nr_mem_range= 0;
    priv(rp)->s_nr_irq= 0;

    /* Done. Privileges have been set. Allow process to run again. */
    old_flags = rp->p_rts_flags;          /* save value of the flags */
    rp->p_rts_flags &= ~NO_PRIV;
    if (old_flags != 0 && rp->p_rts_flags == 0) lock_enqueue(rp);
    return(OK);
case SYS_PRIV_ADD_IO:
    if (rp->p_rts_flags & NO_PRIV)
        return(EPERM);

    /* Only system processes get I/O resources? */
    if (!(priv(rp)->s_flags & SYS_PROC))
        return EPERM;

    /* Get the I/O range */
    caller_phys = umap_local(caller_ptr, D, (vir_bytes) m_ptr->CTL_ARG_PTR,
        sizeof(io_range));
    if (caller_phys == 0)
        return EFAULT;
    kernel_phys = vir2phys(&io_range);
    phys_copy(caller_phys, kernel_phys, sizeof(io_range));
    priv(rp)->s_flags |= CHECK_IO_PORT;    /* Check I/O accesses */
    i= priv(rp)->s_nr_io_range;
    if (i >= NR_IO_RANGE)
        return ENOMEM;

    priv(rp)->s_io_tab[i].ior_base= io_range.ior_base;
    priv(rp)->s_io_tab[i].ior_limit= io_range.ior_limit;
    priv(rp)->s_nr_io_range++;

    return OK;
case SYS_PRIV_ADD_MEM:
    if (rp->p_rts_flags & NO_PRIV)
        return(EPERM);

    /* Only system processes get memory resources? */
    if (!(priv(rp)->s_flags & SYS_PROC))
        return EPERM;

    /* Get the memory range */
    caller_phys = umap_local(caller_ptr, D, (vir_bytes) m_ptr->CTL_ARG_PTR,
        sizeof(mem_range));
    if (caller_phys == 0)
        return EFAULT;
    kernel_phys = vir2phys(&mem_range);
    phys_copy(caller_phys, kernel_phys, sizeof(mem_range));
    priv(rp)->s_flags |= CHECK_MEM; /* Check I/O accesses */
    i= priv(rp)->s_nr_mem_range;
    if (i >= NR_MEM_RANGE)
        return ENOMEM;

    priv(rp)->s_mem_tab[i].mr_base= mem_range.mr_base;
    priv(rp)->s_mem_tab[i].mr_limit= mem_range.mr_limit;
    priv(rp)->s_nr_mem_range++;

    return OK;
case SYS_PRIV_ADD_IRQ:
    if (rp->p_rts_flags & NO_PRIV)
        return(EPERM);

```

```
/* Only system processes get IRQs? */
if (!(priv(rp)->s_flags & SYS_PROC))
    return EPERM;

priv(rp)->s_flags |= CHECK_IRQ; /* Check IRQs */

i= priv(rp)->s_nr_irq;
if (i >= NR_IRQ)
    return ENOMEM;
priv(rp)->s_irq_tab[i]= m_ptr->CTL_MM_PRIV;
priv(rp)->s_nr_irq++;

return OK;

default:
    kprintf("do_privctl: bad request %d\n", m_ptr->CTL_REQUEST);
    return EINVAL;
}
}

#endif /* USE_PRIVCTL */
```



```

/* The kernel call implemented in this file:
 *   m_type:      SYS_SDEVIO
 *
 * The parameters for this kernel call are:
 *   m2_i3:      DIO_REQUEST      (request input or output)
 *   m2_i1:      DIO_TYPE        (flag indicating byte, word, or long)
 *   m2_l1:      DIO_PORT        (port to read/ write)
 *   m2_p1:      DIO_VEC_ADDR    (virtual address of buffer)
 *   m2_l2:      DIO_VEC_SIZE    (number of elements)
 *   m2_i2:      DIO_VEC_PROC    (process where buffer is)
 */

#include "../system.h"
#include <minix/devio.h>
#include <minix/endpoint.h>

#if USE_SDEVIO

/*=====
 *                               do_sdevio                               *
 *=====*/
PUBLIC int do_sdevio(m_ptr)
register message *m_ptr;          /* pointer to request message */
{
    int proc_nr, proc_nr_e = m_ptr->DIO_VEC_ENDPT;
    int count = m_ptr->DIO_VEC_SIZE;
    long port = m_ptr->DIO_PORT;
    phys_bytes phys_buf;

    /* Check if process endpoint is OK.
     * A driver may directly provide a pointer to a buffer at the user-process
     * that initiated the device I/O. Kernel processes, of course, are denied.
     */
    if (proc_nr_e == SELF)
        proc_nr = who_p;
    else
        if (!isokendpt(proc_nr_e, &proc_nr))
            return(EINVAL);
    if (iskerneln(proc_nr)) return(EPERM);

    /* Get and check physical address. */
    if ((phys_buf = numap_local(proc_nr, (vir_bytes) m_ptr->DIO_VEC_ADDR, count)) == 0)
        return(EFAULT);

    /* Perform device I/O for bytes and words. Longs are not supported. */
    if (m_ptr->DIO_REQUEST == DIO_INPUT) {
        switch (m_ptr->DIO_TYPE) {
            case DIO_BYTE: phys_insb(port, phys_buf, count); break;
            case DIO_WORD: phys_insw(port, phys_buf, count); break;
            default: return(EINVAL);
        }
    } else if (m_ptr->DIO_REQUEST == DIO_OUTPUT) {
        switch (m_ptr->DIO_TYPE) {
            case DIO_BYTE: phys_outsb(port, phys_buf, count); break;
            case DIO_WORD: phys_outsw(port, phys_buf, count); break;
            default: return(EINVAL);
        }
    }
    else {
        return(EINVAL);
    }
    return(OK);
}

#endif /* USE_SDEVIO */

```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_SEGCTL
 *
 * The parameters for this kernel call are:
 *   m4_l3:      SEG_PHYS      (physical base address)
 *   m4_l4:      SEG_SIZE      (size of segment)
 *   m4_l1:      SEG_SELECT    (return segment selector here)
 *   m4_l2:      SEG_OFFSET    (return offset within segment here)
 *   m4_l5:      SEG_INDEX     (return index into remote memory map here)
 */
#include "../system.h"
#include "../protect.h"

#if USE_SEGCTL

/*=====
 *                               do_segctl                               *
 *=====*/
PUBLIC int do_segctl(m_ptr)
register message *m_ptr;      /* pointer to request message */
{
/* Return a segment selector and offset that can be used to reach a physical
 * address, for use by a driver doing memory I/O in the A0000 - DFFFF range.
 */
    ul6_t selector;
    vir_bytes offset;
    int i, index;
    register struct proc *rp;
    phys_bytes phys = (phys_bytes) m_ptr->SEG_PHYS;
    vir_bytes size = (vir_bytes) m_ptr->SEG_SIZE;
    int result;

    /* First check if there is a slot available for this segment. */
    rp = proc_addr(who_p);
    index = -1;
    for (i=0; i < NR_REMOTE_SEGS; i++) {
        if (! rp->p_priv->s_farmem[i].in_use) {
            index = i;
            rp->p_priv->s_farmem[i].in_use = TRUE;
            rp->p_priv->s_farmem[i].mem_phys = phys;
            rp->p_priv->s_farmem[i].mem_len = size;
            break;
        }
    }
    if (index < 0) return(ENOSPC);

    if (! machine.protected) {
        selector = phys / HCLICK_SIZE;
        offset = phys % HCLICK_SIZE;
        result = OK;
    } else {
        /* Check if the segment size can be recorded in bytes, that is, check
         * if descriptor's limit field can delimited the allowed memory region
         * precisely. This works up to 1MB. If the size is larger, 4K pages
         * instead of bytes are used.
         */
        if (size < BYTE_GRAN_MAX) {
            init_dataseg(&rp->p_ldt[EXTRA_LDT_INDEX+i], phys, size,
                        USER_PRIVILEGE);
            selector = ((EXTRA_LDT_INDEX+i)*0x08) | (1*0x04) | USER_PRIVILEGE;
            offset = 0;
            result = OK;
        } else {
            init_dataseg(&rp->p_ldt[EXTRA_LDT_INDEX+i], phys & ~0xFFFF, 0,
                        USER_PRIVILEGE);
            selector = ((EXTRA_LDT_INDEX+i)*0x08) | (1*0x04) | USER_PRIVILEGE;
            offset = phys & 0xFFFF;
            result = OK;
        }
    }

    /* Request successfully done. Now return the result. */
    m_ptr->SEG_INDEX = index | REMOTE_SEG;
    m_ptr->SEG_SELECT = selector;

```

```
    m_ptr->SEG_OFFSET = offset;  
    return(result);  
}  
  
#endif /* USE_SEGCTL */
```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_SETALARM
 *
 * The parameters for this kernel call are:
 *   m2_l1:      ALRM_EXP_TIME      (alarm's expiration time)
 *   m2_i2:      ALRM_ABS_TIME      (expiration time is absolute?)
 *   m2_l1:      ALRM_TIME_LEFT     (return seconds left of previous)
 */

#include "../system.h"

#include <minix/endpoint.h>

#ifdef USE_SETALARM

FORWARD _PROTOTYPE( void cause_alarm, (timer_t *tp) );

/*=====
 *                               do_setalarm                               *
 *=====*/
PUBLIC int do_setalarm(m_ptr)
message *m_ptr;                /* pointer to request message */
{
    /* A process requests a synchronous alarm, or wants to cancel its alarm. */
    register struct proc *rp;    /* pointer to requesting process */
    long exp_time;               /* expiration time for this alarm */
    int use_abs_time;            /* use absolute or relative time */
    timer_t *tp;                /* the process' timer structure */
    clock_t uptime;              /* placeholder for current uptime */

    /* Extract shared parameters from the request message. */
    exp_time = m_ptr->ALRM_EXP_TIME; /* alarm's expiration time */
    use_abs_time = m_ptr->ALRM_ABS_TIME; /* flag for absolute time */
    rp = proc_addr(who_p);
    if (! (priv(rp)->s_flags & SYS_PROC)) return(EPERM);

    /* Get the timer structure and set the parameters for this alarm. */
    tp = &(priv(rp)->s_alarm_timer);
    tmr_arg(tp)->ta_int = m_ptr->m_source;
    tp->tmr_func = cause_alarm;

    /* Return the ticks left on the previous alarm. */
    uptime = get_uptime();
    if ((tp->tmr_exp_time != TMR_NEVER) && (uptime < tp->tmr_exp_time) ) {
        m_ptr->ALRM_TIME_LEFT = (tp->tmr_exp_time - uptime);
    } else {
        m_ptr->ALRM_TIME_LEFT = 0;
    }

    /* Finally, (re)set the timer depending on the expiration time. */
    if (exp_time == 0) {
        reset_timer(tp);
    } else {
        tp->tmr_exp_time = (use_abs_time) ? exp_time : exp_time + get_uptime();
        set_timer(tp, tp->tmr_exp_time, tp->tmr_func);
    }
    return(OK);
}

/*=====
 *                               cause_alarm                               *
 *=====*/
PRIVATE void cause_alarm(tp)
timer_t *tp;
{
    /* Routine called if a timer goes off and the process requested a synchronous
     * alarm. The process number is stored in timer argument 'ta_int'. Notify that
     * process with a notification message from CLOCK.
     */
    int proc_nr_e = tmr_arg(tp)->ta_int; /* get process number */
    lock_notify(CLOCK, proc_nr_e);       /* notify process */
}

#endif /* USE_SETALARM */

```

```

/* The kernel call that is implemented in this file:
 *   m_type:      SYS_SIGRETURN
 *
 * The parameters for this kernel call are:
 *   m2_i1:      SIG_ENDPT      # process returning from handler
 *   m2_p1:      SIG_CTXT_PTR   # pointer to sigcontext structure
 */

#include "../system.h"
#include <string.h>
#include <signal.h>
#include <sys/sigcontext.h>

#if USE_SIGRETURN

/*=====
 *                               do_sigreturn                               *
 *=====*/
PUBLIC int do_sigreturn(m_ptr)
message *m_ptr;                /* pointer to request message */
{
    /* POSIX style signals require sys_sigreturn to put things in order before
     * the signalled process can resume execution
     */
    struct sigcontext sc;
    register struct proc *rp;
    phys_bytes src_phys;
    int proc;

    if (!isokendpt(m_ptr->SIG_ENDPT, &proc)) return(EINVAL);
    if (iskerneln(proc)) return(EPERM);
    rp = proc_addr(proc);

    /* Copy in the sigcontext structure. */
    src_phys = umap_local(rp, D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
        (vir_bytes) sizeof(struct sigcontext));
    if (src_phys == 0) return(EFAULT);
    phys_copy(src_phys, vir2phys(&sc), (phys_bytes) sizeof(struct sigcontext));

    /* Make sure that this is not just a jump buffer. */
    if ((sc.sc_flags & SC_SIGCONTEXT) == 0) return(EINVAL);

    /* Fix up only certain key registers if the compiler doesn't use
     * register variables within functions containing setjmp.
     */
    if (sc.sc_flags & SC_NOREGLOCALS) {
        rp->p_reg.retreg = sc.sc_retreg;
        rp->p_reg.fp = sc.sc_fp;
        rp->p_reg.pc = sc.sc_pc;
        rp->p_reg.sp = sc.sc_sp;
        return(OK);
    }
    sc.sc_psw = rp->p_reg.psw;

#if (CHIP == INTEL)
    /* Don't panic kernel if user gave bad selectors. */
    sc.sc_cs = rp->p_reg.cs;
    sc.sc_ds = rp->p_reg.ds;
    sc.sc_es = rp->p_reg.es;
#endif
    if _WORD_SIZE == 4
        sc.sc_fs = rp->p_reg.fs;
        sc.sc_gs = rp->p_reg.gs;
    #endif
    #endif

    /* Restore the registers. */
    memcpy(&rp->p_reg, &sc.sc_regs, sizeof(struct sigregs));
    return(OK);
}
#endif /* USE_SIGRETURN */

```

```

/* The kernel call that is implemented in this file:
 *   m_type:      SYS_SIGSEND
 *
 * The parameters for this kernel call are:
 *   m2_i1:      SIG_ENDPT      # process to call signal handler
 *   m2_p1:      SIG_CTXT_PTR   # pointer to sigcontext structure
 *   m2_i3:      SIG_FLAGS      # flags for S_SIGRETURN call
 *
 */

#include "../system.h"
#include <signal.h>
#include <string.h>
#include <sys/sigcontext.h>

#if USE_SIGSEND

/*=====
 *                               do_sigsend                               *
 *=====*/
PUBLIC int do_sigsend(m_ptr)
message *m_ptr;                /* pointer to request message */
{
    /* Handle sys_sigsend, POSIX-style signal handling. */

    struct sigmsg smsg;
    register struct proc *rp;
    phys_bytes src_phys, dst_phys;
    struct sigcontext sc, *scp;
    struct sigframe fr, *frp;
    int proc;

    if (!isokendpt(m_ptr->SIG_ENDPT, &proc)) return(EINVAL);
    if (iskerneln(proc)) return(EPERM);
    rp = proc_addr(proc);

    /* Get the sigmsg structure into our address space. */
    src_phys = umap_local(proc_addr(PM_PROC_NR), D, (vir_bytes)
        m_ptr->SIG_CTXT_PTR, (vir_bytes) sizeof(struct sigmsg));
    if (src_phys == 0) return(EFAULT);
    phys_copy(src_phys, vir2phys(&smsg), (phys_bytes) sizeof(struct sigmsg));

    /* Compute the user stack pointer where sigcontext will be stored. */
    scp = (struct sigcontext *) smsg.sm_stkptr - 1;

    /* Copy the registers to the sigcontext structure. */
    memcpy(&sc.sc_regs, (char *) &rp->p_reg, sizeof(struct sigregs));

    /* Finish the sigcontext initialization. */
    sc.sc_flags = SC_SIGCONTEXT;
    sc.sc_mask = smsg.sm_mask;

    /* Copy the sigcontext structure to the user's stack. */
    dst_phys = umap_local(rp, D, (vir_bytes) scp,
        (vir_bytes) sizeof(struct sigcontext));
    if (dst_phys == 0) return(EFAULT);
    phys_copy(vir2phys(&sc), dst_phys, (phys_bytes) sizeof(struct sigcontext));

    /* Initialize the sigframe structure. */
    frp = (struct sigframe *) scp - 1;
    fr.sf_scpcopy = scp;
    fr.sf_retadr2 = (void (*)()) rp->p_reg.pc;
    fr.sf_fp = rp->p_reg.fp;
    rp->p_reg.fp = (reg_t) &frp->sf_fp;
    fr.sf_scp = scp;
    fr.sf_code = 0;          /* XXX - should be used for type of FP exception */
    fr.sf_signo = smsg.sm_signo;
    fr.sf_retadr = (void (*)()) smsg.sm_sigreturn;

    /* Copy the sigframe structure to the user's stack. */
    dst_phys = umap_local(rp, D, (vir_bytes) frp,
        (vir_bytes) sizeof(struct sigframe));
    if (dst_phys == 0) return(EFAULT);
    phys_copy(vir2phys(&fr), dst_phys, (phys_bytes) sizeof(struct sigframe));

```

```
/* Reset user registers to execute the signal handler. */
rp->p_reg.sp = (reg_t) frp;
rp->p_reg.pc = (reg_t) smsg.sm_sighandler;

return(OK);
}

#endif /* USE_SIGSEND */
```

```
/* The kernel call implemented in this file:
 *   m_type:      SYS_TIMES
 *
 * The parameters for this kernel call are:
 *   m4_l1:      T_ENDPT      (get info for this process)
 *   m4_l1:      T_USER_TIME  (return values ...)
 *   m4_l2:      T_SYSTEM_TIME
 *   m4_l5:      T_BOOT_TICKS
 */

#include "../system.h"

#include <minix/endpoint.h>

#if USE_TIMES

/*=====
 *                               do_times                               *
 *=====*/
PUBLIC int do_times(m_ptr)
register message *m_ptr;      /* pointer to request message */
{
    /* Handle sys_times().  Retrieve the accounting information. */
    register struct proc *rp;
    int proc_nr, e_proc_nr;

    /* Insert the times needed by the SYS_TIMES kernel call in the message.
     * The clock's interrupt handler may run to update the user or system time
     * while in this code, but that cannot do any harm.
     */
    e_proc_nr = (m_ptr->T_ENDPT == SELF) ? m_ptr->m_source : m_ptr->T_ENDPT;
    if(e_proc_nr != NONE && isokendpt(e_proc_nr, &proc_nr)) {
        rp = proc_addr(proc_nr);
        m_ptr->T_USER_TIME = rp->p_user_time;
        m_ptr->T_SYSTEM_TIME = rp->p_sys_time;
    }
    m_ptr->T_BOOT_TICKS = get_uptime();
    return(OK);
}

#endif /* USE_TIMES */
```



```

/* The kernel call implemented in this file:
 *   m_type:      SYS_TRACE
 *
 * The parameters for this kernel call are:
 *   m2_i1:      CTL_ENDPT      process that is traced
 *   m2_i2:      CTL_REQUEST    trace request
 *   m2_l1:      CTL_ADDRESS    address at traced process' space
 *   m2_l2:      CTL_DATA       data to be written or returned here
 */

#include "../system.h"
#include <sys/ptrace.h>

#if USE_TRACE

/*=====
 *                               do_trace                               *
 *=====*/
#define TR_VLSIZE      ((vir_bytes) sizeof(long))

PUBLIC int do_trace(m_ptr)
register message *m_ptr;
{
/* Handle the debugging commands supported by the ptrace system call
 * The commands are:
 *   T_STOP      stop the process
 *   T_OK        enable tracing by parent for this process
 *   T_GETINS     return value from instruction space
 *   T_GETDATA   return value from data space
 *   T_GETUSER   return value from user process table
 *   T_SETINS     set value from instruction space
 *   T_SETDATA   set value from data space
 *   T_SETUSER   set value in user process table
 *   T_RESUME    resume execution
 *   T_EXIT      exit
 *   T_STEP      set trace bit
 *
 * The T_OK and T_EXIT commands are handled completely by the process manager,
 * all others come here.
 */

register struct proc *rp;
phys_bytes src, dst;
vir_bytes tr_addr = (vir_bytes) m_ptr->CTL_ADDRESS;
long tr_data = m_ptr->CTL_DATA;
int tr_request = m_ptr->CTL_REQUEST;
int tr_proc_nr_e = m_ptr->CTL_ENDPT, tr_proc_nr;
int i;

if(!isokendpt(tr_proc_nr_e, &tr_proc_nr)) return(EINVAL);
if (iskerneln(tr_proc_nr)) return(EPERM);

rp = proc_addr(tr_proc_nr);
if (isemtyp(rp)) return(EIO);
switch (tr_request) {
case T_STOP: /* stop process */
    if (rp->p_rts_flags == 0) lock_dequeue(rp);
    rp->p_rts_flags |= P_STOP;
    rp->p_reg.psw &= ~TRACEBIT; /* clear trace bit */
    return(OK);

case T_GETINS: /* return value from instruction space */
    if (rp->p_memmap[T].mem_len != 0) {
        if ((src = umap_local(rp, T, tr_addr, TR_VLSIZE)) == 0) return(EIO);
        phys_copy(src, vir2phys(&tr_data), (phys_bytes) sizeof(long));
        m_ptr->CTL_DATA = tr_data;
        break;
    }
    /* Text space is actually data space - fall through. */

case T_GETDATA: /* return value from data space */
    if ((src = umap_local(rp, D, tr_addr, TR_VLSIZE)) == 0) return(EIO);
    phys_copy(src, vir2phys(&tr_data), (phys_bytes) sizeof(long));
    m_ptr->CTL_DATA = tr_data;

```

```

        break;

case T_GETUSER:
    /* return value from process table */
    if ((tr_addr & (sizeof(long) - 1)) != 0 ||
        tr_addr > sizeof(struct proc) - sizeof(long))
        return(EIO);
    m_ptr->CTL_DATA = *(long *) ((char *) rp + (int) tr_addr);
    break;

case T_SETINS:
    /* set value in instruction space */
    if (rp->p_memmap[T].mem_len != 0) {
        if ((dst = umap_local(rp, T, tr_addr, TR_VLSIZE)) == 0) return(EIO);
        phys_copy(vir2phys(&tr_data), dst, (phys_bytes) sizeof(long));
        m_ptr->CTL_DATA = 0;
        break;
    }
    /* Text space is actually data space - fall through. */

case T_SETDATA:
    /* set value in data space */
    if ((dst = umap_local(rp, D, tr_addr, TR_VLSIZE)) == 0) return(EIO);
    phys_copy(vir2phys(&tr_data), dst, (phys_bytes) sizeof(long));
    m_ptr->CTL_DATA = 0;
    break;

case T_SETUSER:
    /* set value in process table */
    if ((tr_addr & (sizeof(reg_t) - 1)) != 0 ||
        tr_addr > sizeof(struct stackframe_s) - sizeof(reg_t))
        return(EIO);
    i = (int) tr_addr;
#endif /* CHIP == INTEL */
    /* Altering segment registers might crash the kernel when it
     * tries to load them prior to restarting a process, so do
     * not allow it.
     */
    if (i == (int) &((struct proc *) 0)->p_reg.cs ||
        i == (int) &((struct proc *) 0)->p_reg.ds ||
        i == (int) &((struct proc *) 0)->p_reg.es ||
#ifdef _WORD_SIZE == 4
        i == (int) &((struct proc *) 0)->p_reg.gs ||
        i == (int) &((struct proc *) 0)->p_reg.fs ||
#endif
        i == (int) &((struct proc *) 0)->p_reg.ss)
        return(EIO);
#ifdef _WORD_SIZE == 4
    if (i == (int) &((struct proc *) 0)->p_reg.psw)
        /* only selected bits are changeable */
        SETPSW(rp, tr_data);
    else
        *(reg_t *) ((char *) &rp->p_reg + i) = (reg_t) tr_data;
    m_ptr->CTL_DATA = 0;
    break;
#endif

case T_RESUME:
    /* resume execution */
    rp->p_rts_flags &= ~P_STOP;
    if (rp->p_rts_flags == 0) lock_enqueue(rp);
    m_ptr->CTL_DATA = 0;
    break;

case T_STEP:
    /* set trace bit */
    rp->p_reg.psw |= TRACEBIT;
    rp->p_rts_flags &= ~P_STOP;
    if (rp->p_rts_flags == 0) lock_enqueue(rp);
    m_ptr->CTL_DATA = 0;
    break;

default:
    return(EIO);
}
return(OK);
}
#endif /* USE_TRACE */

```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_UMAP
 *
 * The parameters for this kernel call are:
 *   m5_i1:      CP_SRC_PROC_NR   (process number)
 *   m5_c1:      CP_SRC_SPACE     (segment where address is: T, D, or S)
 *   m5_l1:      CP_SRC_ADDR      (virtual address)
 *   m5_l2:      CP_DST_ADDR      (returns physical address)
 *   m5_l3:      CP_NR_BYTES      (size of datastructure)
 */

#include "../system.h"

#ifdef USE_UMAP

/*=====
 *                               do_umap                               *
 *=====*/
PUBLIC int do_umap(m_ptr)
register message *m_ptr;          /* pointer to request message */
{
    /* Map virtual address to physical, for non-kernel processes. */
    int seg_type = m_ptr->CP_SRC_SPACE & SEGMENT_TYPE;
    int seg_index = m_ptr->CP_SRC_SPACE & SEGMENT_INDEX;
    vir_bytes offset = m_ptr->CP_SRC_ADDR;
    int count = m_ptr->CP_NR_BYTES;
    int endpt = (int) m_ptr->CP_SRC_ENDPT;
    int proc_nr;
    phys_bytes phys_addr;

    /* Verify process number. */
    if (endpt == SELF)
        proc_nr = who_p;
    else
        if (!isokendpt(endpt, &proc_nr))
            return(EINVAL);

    /* See which mapping should be made. */
    switch(seg_type) {
    case LOCAL_SEG:
        phys_addr = umap_local(proc_addr(proc_nr), seg_index, offset, count);
        break;
    case REMOTE_SEG:
        phys_addr = umap_remote(proc_addr(proc_nr), seg_index, offset, count);
        break;
    case BIOS_SEG:
        phys_addr = umap_bios(proc_addr(proc_nr), offset, count);
        break;
    default:
        return(EINVAL);
    }
    m_ptr->CP_DST_ADDR = phys_addr;
    return (phys_addr == 0) ? EFAULT: OK;
}

#endif /* USE_UMAP */

```

```
/* This file provides a catch-all handler for unused kernel calls. A kernel
 * call may be unused when it is not defined or when it is disabled in the
 * kernel's configuration.
 */
#include "../system.h"

/*=====*
 *                               do_unused                               *
 *=====*/
PUBLIC int do_unused(m)
message *m;                               /* pointer to request message */
{
    kprintf("SYSTEM: got unused request %d from %d", m->m_type, m->m_source);
    return(EBADREQUEST);                  /* illegal message type */
}
```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_VIRVCOPY, SYS_PHYSVCOPY
 *
 * The parameters for this kernel call are:
 *   ml_i3:       VCP_VEC_SIZE      size of copy request vector
 *   ml_p1:       VCP_VEC_ADDR      address of vector at caller
 *   ml_i2:       VCP_NR_OK         number of successfull copies
 */

#include "../system.h"
#include <minix/type.h>

#if (USE_VIRVCOPY || USE_PHYSVCOPY)

/* Buffer to hold copy request vector from user. */
PRIVATE struct vir_cp_req vir_cp_req[VCOPY_VEC_SIZE];

/*=====
 *                               do_vcopy
 *=====*/
PUBLIC int do_vcopy(m_ptr)
register message *m_ptr;      /* pointer to request message */
{
/* Handle sys_virvcopy() and sys_physvcopy() that pass a vector with copy
 * requests. Although a single handler function is used, there are two
 * different kernel calls so that permissions can be checked.
 */
    int nr_req;
    vir_bytes caller_vir;
    phys_bytes caller_phys;
    phys_bytes kernel_phys;
    phys_bytes bytes;
    int i,s;
    struct vir_cp_req *req;

    /* Check if request vector size is ok. */
    nr_req = (unsigned) m_ptr->VCP_VEC_SIZE;
    if (nr_req > VCOPY_VEC_SIZE) return(EINVAL);
    bytes = nr_req * sizeof(struct vir_cp_req);

    /* Calculate physical addresses and copy (port,value)-pairs from user. */
    caller_vir = (vir_bytes) m_ptr->VCP_VEC_ADDR;
    caller_phys = umap_local(proc_addr(who_p), D, caller_vir, bytes);
    if (0 == caller_phys) return(EFAULT);
    kernel_phys = vir2phys(vir_cp_req);
    phys_copy(caller_phys, kernel_phys, (phys_bytes) bytes);

    /* Assume vector with requests is correct. Try to copy everything. */
    m_ptr->VCP_NR_OK = 0;
    for (i=0; i<nr_req; i++) {

        req = &vir_cp_req[i];

        /* Check if physical addressing is used without SYS_PHYSVCOPY. */
        if (((req->src.segment | req->dst.segment) & PHYS_SEG) &&
            m_ptr->m_type != SYS_PHYSVCOPY) return(EPERM);
        if ((s=virtual_copy(&req->src, &req->dst, req->count)) != OK)
            return(s);
        m_ptr->VCP_NR_OK ++;
    }
    return(OK);
}

#endif /* (USE_VIRVCOPY || USE_PHYSVCOPY) */

```

```

/* The kernel call implemented in this file:
 *   m_type:      SYS_VDEVIO
 *
 * The parameters for this kernel call are:
 *   m2_i3:      DIO_REQUEST      (request input or output)
 *   m2_i1:      DIO_TYPE        (flag indicating byte, word, or long)
 *   m2_p1:      DIO_VEC_ADDR    (pointer to port/ value pairs)
 *   m2_i2:      DIO_VEC_SIZE    (number of ports to read or write)
 */

#include "../system.h"
#include <minix/devio.h>
#include <minix/endpoint.h>

#if USE_VDEVIO

/* Buffer for SYS_VDEVIO to copy (port,value)-pairs from/ to user. */
PRIVATE char vdevio_buf[VDEVIO_BUF_SIZE];
PRIVATE pvb_pair_t *pvb = (pvb_pair_t *) vdevio_buf;
PRIVATE pvw_pair_t *pvw = (pvw_pair_t *) vdevio_buf;
PRIVATE pvl_pair_t *pvl = (pvl_pair_t *) vdevio_buf;

/*=====
 *                               do_vdevio                               *
 *=====*/
PUBLIC int do_vdevio(m_ptr)
register message *m_ptr;          /* pointer to request message */
{
/* Perform a series of device I/O on behalf of a non-kernel process. The
 * I/O addresses and I/O values are fetched from and returned to some buffer
 * in user space. The actual I/O is wrapped by lock() and unlock() to prevent
 * that I/O batch from being interrupted.
 * This is the counterpart of do_devio, which performs a single device I/O.
 */
    int vec_size;                /* size of vector */
    int io_in;                   /* true if input */
    size_t bytes;                /* # bytes to be copied */
    vir_bytes caller_vir;        /* virtual address at caller */
    phys_bytes caller_phys;      /* physical address at caller */
    int i;

    /* Get the request, size of the request vector, and check the values. */
    if (m_ptr->DIO_REQUEST == DIO_INPUT) io_in = TRUE;
    else if (m_ptr->DIO_REQUEST == DIO_OUTPUT) io_in = FALSE;
    else return(EINVAL);
    if ((vec_size = m_ptr->DIO_VEC_SIZE) <= 0) return(EINVAL);
    switch (m_ptr->DIO_TYPE) {
        case DIO_BYTE: bytes = vec_size * sizeof(pvb_pair_t); break;
        case DIO_WORD: bytes = vec_size * sizeof(pvw_pair_t); break;
        case DIO_LONG: bytes = vec_size * sizeof(pvl_pair_t); break;
        default: return(EINVAL); /* check type once and for all */
    }
    if (bytes > sizeof(vdevio_buf)) return(E2BIG);

    /* Calculate physical addresses and copy (port,value)-pairs from user. */
    caller_vir = (vir_bytes) m_ptr->DIO_VEC_ADDR;
    caller_phys = umap_local(proc_addr(who_p), D, caller_vir, bytes);
    if (0 == caller_phys) return(EFAULT);
    phys_copy(caller_phys, vir2phys(vdevio_buf), (phys_bytes) bytes);

    /* Perform actual device I/O for byte, word, and long values. Note that
     * the entire switch is wrapped in lock() and unlock() to prevent the I/O
     * batch from being interrupted.
     */
    lock(13, "do_vdevio");
    switch (m_ptr->DIO_TYPE) {
        case DIO_BYTE: /* byte values */
            if (io_in) for (i=0; i<vec_size; i++) pvb[i].value = inb(pvb[i].port);
            else for (i=0; i<vec_size; i++) outb(pvb[i].port, pvb[i].value);
            break;
        case DIO_WORD: /* word values */
            if (io_in) for (i=0; i<vec_size; i++) pvw[i].value = inw(pvw[i].port);
            else for (i=0; i<vec_size; i++) outw(pvw[i].port, pvw[i].value);
            break;
    }
}

```

```
default:                                     /* long values */
    if (io_in) for (i=0; i<vec_size; i++) pvl[i].value = inl(pvl[i].port);
    else      for (i=0; i<vec_size; i++) outl(pvb[i].port, pvl[i].value);
}
unlock(13);

/* Almost done, copy back results for input requests. */
if (io_in) phys_copy(vir2phys(vdevio_buf), caller_phys, (phys_bytes) bytes);
return(OK);
}

#endif /* USE_VDEVIO */
```

```

/* The system call implemented in this file:
 *   m_type:      SYS_VM_MAP
 *
 * The parameters for this system call are:
 *   m4_l1:       Process that requests map (VM_MAP_ENDPT)
 *   m4_l2:       Map (TRUE) or unmap (FALSE) (VM_MAP_MAPUNMAP)
 *   m4_l3:       Base address (VM_MAP_BASE)
 *   m4_l4:       Size (VM_MAP_SIZE)
 *   m4_l5:       Memory address (VM_MAP_ADDR)
 */
#include "../system.h"

#include <sys/vm.h>

PRIVATE int vm_needs_init= 1;
PRIVATE u32_t vm_cr3;

FORWARD _PROTOTYPE( void vm_init, (void) ) ;
FORWARD _PROTOTYPE( void phys_put32, (phys_bytes addr, u32_t value) ) ;
FORWARD _PROTOTYPE( u32_t phys_get32, (phys_bytes addr) ) ;
FORWARD _PROTOTYPE( void vm_set_cr3, (u32_t value) ) ;
FORWARD _PROTOTYPE( void set_cr3, (void) ) ;
FORWARD _PROTOTYPE( void vm_enable_paging, (void) ) ;
FORWARD _PROTOTYPE( void map_range, (u32_t base, u32_t size,
                                     u32_t offset) ) ;

/*=====
 *                               do_vm_map
 *=====*/
PUBLIC int do_vm_map(m_ptr)
message *m_ptr;          /* pointer to request message */
{
    int proc_nr, do_map;
    phys_bytes base, size, offset, p_phys;
    struct proc *pp;

    /* do_serial_debug= 1; */

    if (vm_needs_init)
    {
        vm_needs_init= 0;
        vm_init();
    }

    if (m_ptr->VM_MAP_ENDPT == SELF) {
        proc_nr = who_p;
    } else {
        if (!isokendpt(m_ptr->VM_MAP_ENDPT, &proc_nr))
            return EINVAL;
    }

    do_map= m_ptr->VM_MAP_MAPUNMAP;
    base= m_ptr->VM_MAP_BASE;
    size= m_ptr->VM_MAP_SIZE;
    offset= m_ptr->VM_MAP_ADDR;

    pp= proc_addr(proc_nr);
    p_phys= umap_local(pp, D, base, size);
    if (p_phys == 0)
        return EFAULT;

    if (do_map)
    {
        pp->p_misc_flags |= MF_VM;

        map_range(p_phys, size, offset);
    }
    else
    {
        map_range(p_phys, size, p_phys);
    }
    vm_set_cr3(vm_cr3);

    return OK;
}

```



```

}

/*=====
 *                               vm_map_default                               *
 *=====*/
PUBLIC void vm_map_default(pp)
struct proc *pp;
{
    phys_bytes base_clicks, size_clicks;

    if (vm_needs_init)
        panic("vm_map_default: VM not initialized?", NO_NUM);
    pp->p_misc_flags &= ~MF_VM;
    base_clicks= pp->p_memmap[D].mem_phys;
    size_clicks= pp->p_memmap[S].mem_phys+pp->p_memmap[S].mem_len -
        base_clicks;
    map_range(base_clicks << CLICK_SHIFT, size_clicks << CLICK_SHIFT,
        base_clicks << CLICK_SHIFT);
    vm_set_cr3(vm_cr3);
}

PRIVATE void vm_init(void)
{
    int o;
    phys_bytes p, pt_size;
    phys_bytes vm_dir_base, vm_pt_base, phys_mem;
    u32_t entry;
    unsigned pages;

    if (!vm_size)
        panic("vm_init: no space for page tables", NO_NUM);

    /* Align page directory */
    o= (vm_base % PAGE_SIZE);
    if (o != 0)
        o= PAGE_SIZE-o;
    vm_dir_base= vm_base+o;

    /* Page tables start after the page directory */
    vm_pt_base= vm_dir_base+PAGE_SIZE;

    pt_size= (vm_base+vm_size)-vm_pt_base;
    pt_size -= (pt_size % PAGE_SIZE);

    /* Compute the number of pages based on vm_mem_high */
    pages= (vm_mem_high-1)/PAGE_SIZE + 1;

    if (pages * I386_VM_PT_ENT_SIZE > pt_size)
        panic("vm_init: page table too small", NO_NUM);

    for (p= 0; p*I386_VM_PT_ENT_SIZE < pt_size; p++)
    {
        phys_mem= p*PAGE_SIZE;
        entry= phys_mem | I386_VM_USER | I386_VM_WRITE |
            I386_VM_PRESENT;
        if (phys_mem >= vm_mem_high)
            entry= 0;
        phys_put32(vm_pt_base + p*I386_VM_PT_ENT_SIZE, entry);
    }

    for (p= 0; p < I386_VM_DIR_ENTRIES; p++)
    {
        phys_mem= vm_pt_base + p*PAGE_SIZE;
        entry= phys_mem | I386_VM_USER | I386_VM_WRITE |
            I386_VM_PRESENT;
        if (phys_mem >= vm_pt_base + pt_size)
            entry= 0;
        phys_put32(vm_dir_base + p*I386_VM_PT_ENT_SIZE, entry);
    }
    vm_set_cr3(vm_dir_base);
    level0(vm_enable_paging);
}

PRIVATE void phys_put32(addr, value)

```

```
phys_bytes addr;
u32_t value;
{
    phys_copy(vir2phys((vir_bytes)&value), addr, sizeof(value));
}

PRIVATE u32_t phys_get32(addr)
phys_bytes addr;
{
    u32_t value;

    phys_copy(addr, vir2phys((vir_bytes)&value), sizeof(value));

    return value;
}

PRIVATE void vm_set_cr3(value)
u32_t value;
{
    vm_cr3= value;
    level0(set_cr3);
}

PRIVATE void set_cr3()
{
    write_cr3(vm_cr3);
}

PRIVATE void vm_enable_paging(void)
{
    u32_t cr0;

    cr0= read_cr0();
    write_cr0(cr0 | I386_CR0_PG);
}

PRIVATE void map_range(base, size, offset)
u32_t base;
u32_t size;
u32_t offset;
{
    u32_t curr_pt, curr_pt_addr, entry;
    int dir_ent, pt_ent;

    if (base % PAGE_SIZE != 0)
        panic("map_range: bad base", base);
    if (size % PAGE_SIZE != 0)
        panic("map_range: bad size", size);
    if (offset % PAGE_SIZE != 0)
        panic("map_range: bad offset", offset);

    curr_pt= -1;
    curr_pt_addr= 0;
    while (size != 0)
    {
        dir_ent= (base >> I386_VM_DIR_ENT_SHIFT);
        pt_ent= (base >> I386_VM_PT_ENT_SHIFT) & I386_VM_PT_ENT_MASK;
        if (dir_ent != curr_pt)
        {
            /* Get address of page table */
            curr_pt= dir_ent;
            curr_pt_addr= phys_get32(vm_cr3 +
                                   dir_ent * I386_VM_PT_ENT_SIZE);
            curr_pt_addr &= I386_VM_ADDR_MASK;
        }
        entry= offset | I386_VM_USER | I386_VM_WRITE |
              I386_VM_PRESENT;
    #if 0    /* Do we need this for memory mapped I/O? */
        entry |= I386_VM_PCD | I386_VM_PWT;
    #endif

    phys_put32(curr_pt_addr + pt_ent * I386_VM_PT_ENT_SIZE, entry);
    offset += PAGE_SIZE;
    base += PAGE_SIZE;
    size -= PAGE_SIZE;
}
```

```
}  
}
```

```
/* The system call implemented in this file:
 *   m_type:      SYS_VM_SETBUF
 *
 * The parameters for this system call are:
 *   m4_l1:       Start of the buffer
 *   m4_l2:       Length of the buffer
 *   m4_l3:       End of main memory
 */
#include "../system.h"

#define VM_DEBUG 0                /* enable/ disable debug output */

/*=====*
 *                               do_vm_setbuf                               *
 *=====*/
PUBLIC int do_vm_setbuf(m_ptr)
message *m_ptr;                  /* pointer to request message */
{
    vm_base= m_ptr->m4_l1;
    vm_size= m_ptr->m4_l2;
    vm_mem_high= m_ptr->m4_l3;

    #if VM_DEBUG
        kprintf("do_vm_setbuf: got 0x%x @ 0x%x for 0x%x\n",
                vm_size, vm_base, vm_mem_high);
    #endif

    return OK;
}
```

**Table of Contents**

1	<i>Makefile</i> .....	sheets	1 to	1 ( 1)	pages	1- 1	52 lines
2	<i>clock.c</i> .....	sheets	2 to	6 ( 5)	pages	2- 6	317 lines
3	<i>config.h</i> .....	sheets	7 to	8 ( 2)	pages	7- 8	83 lines
4	<i>const.h</i> .....	sheets	9 to	10 ( 2)	pages	9- 10	92 lines
5	<i>debug.c</i> .....	sheets	11 to	13 ( 3)	pages	11- 13	167 lines
6	<i>debug.h</i> .....	sheets	14 to	14 ( 1)	pages	14- 14	74 lines
7	<i>exception.c</i> .....	sheets	15 to	16 ( 2)	pages	15- 16	88 lines
8	<i>glo.h</i> .....	sheets	17 to	17 ( 1)	pages	17- 17	69 lines
9	<i>i8259.c</i> .....	sheets	18 to	20 ( 3)	pages	18- 20	194 lines
10	<i>ipc.h</i> .....	sheets	21 to	21 ( 1)	pages	21- 21	35 lines
11	<i>kernel.h</i> .....	sheets	22 to	22 ( 1)	pages	22- 22	35 lines
12	<i>klib.s</i> .....	sheets	23 to	23 ( 1)	pages	23- 23	10 lines
13	<i>klib386.s</i> .....	sheets	24 to	32 ( 9)	pages	24- 32	616 lines
14	<i>klib88.s</i> .....	sheets	32 to	32 ( 1)	pages	32- 32	1 lines
15	<i>kprintf.c</i> .....	sheets	33 to	33 ( 1)	pages	33- 33	72 lines
16	<i>main.c</i> .....	sheets	34 to	37 ( 4)	pages	34- 37	263 lines
17	<i>mpx.s</i> .....	sheets	38 to	38 ( 1)	pages	38- 38	10 lines
18	<i>mpx386.s</i> .....	sheets	39 to	46 ( 8)	pages	39- 46	536 lines
19	<i>mpx88.s</i> .....	sheets	46 to	46 ( 1)	pages	46- 46	1 lines
20	<i>priv.h</i> .....	sheets	47 to	48 ( 2)	pages	47- 48	101 lines
21	<i>proc.c</i> .....	sheets	49 to	59 (11)	pages	49- 59	791 lines
22	<i>proc.h</i> .....	sheets	60 to	61 ( 2)	pages	60- 61	119 lines
23	<i>protect.c</i> .....	sheets	62 to	67 ( 6)	pages	62- 67	383 lines
24	<i>protect.h</i> .....	sheets	68 to	69 ( 2)	pages	68- 69	125 lines
25	<i>proto.h</i> .....	sheets	70 to	72 ( 3)	pages	70- 72	171 lines
26	<i>sconst.h</i> .....	sheets	73 to	73 ( 1)	pages	73- 73	37 lines
27	<i>start.c</i> .....	sheets	74 to	75 ( 2)	pages	74- 75	123 lines
28	<i>system.c</i> .....	sheets	76 to	83 ( 8)	pages	76- 83	548 lines
29	<i>system.h</i> .....	sheets	84 to	86 ( 3)	pages	84- 86	180 lines
30	<i>table.c</i> .....	sheets	87 to	88 ( 2)	pages	87- 88	126 lines
31	<i>type.h</i> .....	sheets	89 to	90 ( 2)	pages	89- 90	112 lines
32	<i>utility.c</i> .....	sheets	91 to	91 ( 1)	pages	91- 91	28 lines
33	<i>Makefile</i> .....	sheets	92 to	94 ( 3)	pages	92- 94	154 lines
34	<i>do_abort.c</i> .....	sheets	95 to	95 ( 1)	pages	95- 95	50 lines
35	<i>do_copy.c</i> .....	sheets	96 to	96 ( 1)	pages	96- 96	69 lines
36	<i>do_devio.c</i> .....	sheets	97 to	98 ( 2)	pages	97- 98	83 lines
37	<i>do_endksig.c</i> .....	sheets	99 to	99 ( 1)	pages	99- 99	45 lines
38	<i>do_exec.c</i> .....	sheets	100 to	100 ( 1)	pages	100-100	64 lines
39	<i>do_exit.c</i> .....	sheets	101 to	102 ( 2)	pages	101-102	94 lines
40	<i>do_fork.c</i> .....	sheets	103 to	104 ( 2)	pages	103-104	90 lines
41	<i>do_getinfo.c</i> .....	sheets	105 to	107 ( 3)	pages	105-107	161 lines
42	<i>do_getksig.c</i> .....	sheets	108 to	108 ( 1)	pages	108-108	48 lines
43	<i>do_int86.c</i> .....	sheets	109 to	109 ( 1)	pages	109-109	46 lines
44	<i>do_iopenable.c</i> .....	sheets	110 to	110 ( 1)	pages	110-110	35 lines
45	<i>do_irqctl.c</i> .....	sheets	111 to	113 ( 3)	pages	111-113	168 lines
46	<i>do_kill.c</i> .....	sheets	114 to	114 ( 1)	pages	114-114	51 lines
47	<i>do_memset.c</i> .....	sheets	115 to	115 ( 1)	pages	115-115	30 lines
48	<i>do_newmap.c</i> .....	sheets	116 to	116 ( 1)	pages	116-116	65 lines
49	<i>do_nice.c</i> .....	sheets	117 to	117 ( 1)	pages	117-117	63 lines
50	<i>do_privctl.c</i> .....	sheets	118 to	120 ( 3)	pages	118-120	172 lines
51	<i>do_sdevio.c</i> .....	sheets	121 to	121 ( 1)	pages	121-121	67 lines
52	<i>do_segctl.c</i> .....	sheets	122 to	123 ( 2)	pages	122-123	81 lines
53	<i>do_setalarm.c</i> .....	sheets	124 to	124 ( 1)	pages	124-124	75 lines
54	<i>do_sigreturn.c</i> .....	sheets	125 to	125 ( 1)	pages	125-125	73 lines
55	<i>do_sigsend.c</i> .....	sheets	126 to	127 ( 2)	pages	126-127	85 lines
56	<i>do_times.c</i> .....	sheets	128 to	128 ( 1)	pages	128-128	43 lines
57	<i>do_trace.c</i> .....	sheets	129 to	130 ( 2)	pages	129-130	147 lines
58	<i>do_umap.c</i> .....	sheets	131 to	131 ( 1)	pages	131-131	57 lines
59	<i>do_unused.c</i> .....	sheets	132 to	132 ( 1)	pages	132-132	17 lines
60	<i>do_vcopy.c</i> .....	sheets	133 to	133 ( 1)	pages	133-133	66 lines
61	<i>do_vdevio.c</i> .....	sheets	134 to	135 ( 2)	pages	134-135	88 lines
62	<i>do_vm.c</i> .....	sheets	136 to	139 ( 4)	pages	136-139	225 lines
63	<i>do_vm_setbuf.c</i> .....	sheets	140 to	140 ( 1)	pages	140-140	30 lines