

```

/* 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
    }
}

/* 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)) {
#ifdef 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)) {

/* CODIGO PRACTICA 3 */
messg_sn[caller_ptr->p_nr + NR_TASKS] += 1;

/* 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 to 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 */

#if 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) { /* add to empty queue */
    rdy_head[q] = rdy_tail[q] = rp; /* create a new queue */
    rp->p_nextready = NIL_PROC; /* mark new end */
}
else if (front) { /* add to head of queue */
    rp->p_nextready = rdy_head[q]; /* chain head of queue */
    rdy_head[q] = rp; /* set new queue head */
}
else { /* add to tail of queue */
    rdy_tail[q]->p_nextready = rp; /* chain tail of queue */
    rdy_tail[q] = rp; /* set new queue tail */
    rp->p_nextready = NIL_PROC; /* 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])      /* queue tail removed */
                rdy_tail[q] = prev_xp; /* 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 (! isempty(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

/* 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(isemptyn(*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, line,
            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;
}

```