```
#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 "const.h"
#include "priv.h"

struct proc {
  struct stackframe_s p_reg;    /* process' registers saved in stack frame */
  struct segframe p_seg;        /* segment descriptors */
  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 not 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 */
  clock_t p_recent_time;        /* recent 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 */

  endpoint_t 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 SIGNALED        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 */

/* These runtime flags can be tested and manipulated by these macros. */

```

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

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

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

  /* Clear the process table. Anounce each slot as empty and set up mappings
   * for proc_addr() and proc_nr() macros. Do the same for the table with
   * privilege structures for the system processes.
   */
  for (rp = BEG_PROC_ADDR, i = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++i) {
        rp->p_rts_flags = SLOT_FREE;            /* initialize free slot */
        rp->p_nr = i;                           /* proc number from ptr */
        rp->p_endpoint = _ENDPOINT(0, rp->p_nr); /* generation no. 0 */
        (pproc_addr + NR_TASKS)[i] = rp;        /* proc ptr from number */
        rp->p_recent_time = 0;                  /* recent cpu time */
  }
  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.
   */

```

```
/* The kernel call implemented in this file:
 *   m_type:    SYS_FORK
 *
 * The parameters for this kernel call are:
 *    m1_i1:    PR_SLOT  (child's process table slot)
 *    m1_i2:    PR_ENDPT (parent, process that forked)
 */

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

#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 (_MINIX_CHIP == _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, r;
  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 (isemptyp(rpp) || ! isemptyp(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 (_MINIX_CHIP == _CHIP_INTEL)
  old_ldt_sel = rpc->p_seg.p_ldt_sel;   /* backup local descriptors */
  *rpc = *rpp;                          /* copy 'proc' struct */
  rpc->p_seg.p_ldt_sel = old_ldt_sel;   /* restore descriptors */
#else
  *rpc = *rpp;                          /* copy 'proc' struct */
#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 */

  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;
  rpc->p_recent_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;

```

```
131   PRIVATE void init_clock()
132   {
133     /* First of all init the clock system.
134      *
135      * Here the (a) clock is set to produce a interrupt at
136      * every 1/60 second (ea. 60Hz).
137      *
138      * Running right away.
139      */
140     arch_init_clock();    /* architecture-dependent initialization. */
141
142     /* Initialize the CLOCK's interrupt hook. */
143     clock_hook.proc_nr_e = CLOCK;
144
145     put_irq_handler(&clock_hook, CLOCK_IRQ, clock_handler);
146     enable_irq(&clock_hook);              /* ready for clock interrupts */
147
148     /* Set a watchdog timer to periodically balance the scheduling queues. */
149     balance_queues(NULL);                 /* side-effect sets new timer */
150   }
151
152   /*===========================================================================*
153    *                                clock_handler                              *
154    *===========================================================================*/
155   PRIVATE int clock_handler(hook)
156   irq_hook_t *hook;
157   {
158   /* This executes on each clock tick (i.e., every time the timer chip generates
159    * an interrupt). It does a little bit of work so the clock task does not have
160    * to be called on every tick.  The clock task is called when:
161    *
162    *      (1) the scheduling quantum of the running process has expired, or
163    *      (2) a timer has expired and the watchdog function should be run.
164    *
165    * Many global global and static variables are accessed here.  The safety of
166    * this must be justified. All scheduling and message passing code acquires a
167    * lock by temporarily disabling interrupts, so no conflicts with calls from
168    * the task level can occur. Furthermore, interrupts are not reentrant, the
169    * interrupt handler cannot be bothered by other interrupts.
170    *
171    * Variables that are updated in the clock's interrupt handler:
172    *      lost_ticks:
173    *              Clock ticks counted outside the clock task. This for example
174    *              is used when the boot monitor processes a real mode interrupt.
175    *      realtime:
176    *              The current uptime is incremented with all outstanding ticks.
177    *      proc_ptr, bill_ptr:
178    *              These are used for accounting.  It does not matter if proc.c
179    *              is changing them, provided they are always valid pointers,
180    *              since at worst the previous process would be billed.
181    */
182     register unsigned ticks;
183
184     /* Get number of ticks and update realtime. */
185     ticks = lost_ticks + 1;
186     lost_ticks = 0;
187     realtime += ticks;
188
189     /* Update user and system accounting times. Charge the current process for
190      * user time. If the current process is not billable, that is, if a non-user
191      * process is running, charge the billable process for system time as well.
192      * Thus the unbillable process' user time is the billable user's system time.
193      */
194
195     proc_ptr->p_user_time += ticks;
```

```
196     proc_ptr->p_recent_time += ticks;
197     if (priv(proc_ptr)->s_flags & PREEMPTIBLE) {
198         proc_ptr->p_ticks_left -= ticks;
199     }
200     if (! (priv(proc_ptr)->s_flags & BILLABLE)) {
201         bill_ptr->p_sys_time += ticks;
202         bill_ptr->p_ticks_left -= ticks;
203     }
204
205     /* Update load average. */
206     load_update();
207
208     /* Check if do_clocktick() must be called. Done for alarms and scheduling.
209      * Some processes, such as the kernel tasks, cannot be preempted.
210      */
211     if ((next_timeout <= realtime) || (proc_ptr->p_ticks_left <= 0)) {
212         prev_ptr = proc_ptr;                        /* store running process */
213         lock_notify(HARDWARE, CLOCK);               /* send notification */
214     }
215     return(1);                                      /* reenable interrupts */
216   }
217
218   /*===========================================================================*
219    *                                get_uptime                                 *
220    *===========================================================================*/
221   PUBLIC clock_t get_uptime(void)
222   {
223     /* Get and return the current clock uptime in ticks. */
224     return(realtime);
225   }
226
227   /*===========================================================================*
228    *                                set_timer                                  *
229    *===========================================================================*/
230   PUBLIC void set_timer(tp, exp_time, watchdog)
231   struct timer *tp;               /* pointer to timer structure */
232   clock_t exp_time;               /* expiration realtime */
233   tmr_func_t watchdog;            /* watchdog to be called */
234   {
235   /* Insert the new timer in the active timers list. Always update the
236    * next timeout time by setting it to the front of the active list.
237    */
238     tmrs_settimer(&clock_timers, tp, exp_time, watchdog, NULL);
239     next_timeout = clock_timers->tmr_exp_time;
240   }
241
242   /*===========================================================================*
243    *                                reset_timer                                *
244    *===========================================================================*/
245   PUBLIC void reset_timer(tp)
246   struct timer *tp;               /* pointer to timer structure */
247   {
248   /* The timer pointed to by 'tp' is no longer needed. Remove it from both the
249    * active and expired lists. Always update the next timeout time by setting
250    * it to the front of the active list.
251    */
252     tmrs_clrtimer(&clock_timers, tp, NULL);
253     next_timeout = (clock_timers == NULL) ?
254           TMR_NEVER : clock_timers->tmr_exp_time;
255   }
256
257   /*===========================================================================*
258    *                                load_update                                *
259    *===========================================================================*/
260   PRIVATE void load_update(void)
```

```
586     check_runqueues("dequeue2");
587   #endif
588   }
589
590   /*===========================================================================*
591    *                                sched                                      *
592    *===========================================================================*/
593   PRIVATE void sched(rp, queue, front)
594   register struct proc *rp;                       /* process to be scheduled */
595   int *queue;                                     /* return: queue to use */
596   int *front;                                     /* return: front or back */
597   {
598   /* This function determines the scheduling policy.  It is called whenever a
599    * process must be added to one of the scheduling queues to decide where to
600    * insert it.  As a side-effect the process' priority may be updated.
601    */
602     int time_left = (rp->p_ticks_left > 0);       /* quantum fully consumed */
603
604     /* Check whether the process has time left. Otherwise give a new quantum
605      * and lower the process' priority, unless the process already is in the
606      * lowest queue.
607      */
608     if (! time_left) {                            /* quantum consumed ? */
609         rp->p_ticks_left = rp->p_quantum_size;    /* give new quantum */
610         if (rp->p_priority < (IDLE_Q-1)) {
611             rp->p_priority += 1;                  /* lower priority */
612         }
613     }
614
615     /* If there is time left, the process is added to the front of its queue,
616      * so that it can immediately run. The queue to use simply is always the
617      * process' current priority.
618      */
619     *queue = rp->p_priority;
620     *front = time_left;
621   }
622
623   /*===========================================================================*
624    *                                pick_proc                                  *
625    *===========================================================================*/
626   PRIVATE void pick_proc()
627   {
628   /* Decide who to run now.  A new process is selected by setting 'next_ptr'.
629    * When a billable process is selected, record it in 'bill_ptr', so that the
630    * clock task can tell who to bill for system time.
631    */
632     register struct proc *rp;                     /* process to run */
633     int q;                                        /* iterate over queues */
634     register struct proc *low_rp = NIL_PROC;      /* lowest time user process */
635     clock_t low_time = LONG_MAX;                  /* lowest time */
636
637     /* Check each of the scheduling queues except for the idle queue for ready
638      * processes. The number of queues is defined in proc.h, and priorities are
639      * set in the task table.
640      */
641     for (q = 0; q < NR_SCHED_QUEUES - 1; q++) {
642           for (rp = rdy_head[q]; rp != NIL_PROC; rp = rp->p_nextready) {
643                   /* System process. */
644                   if (priv(rp)->s_flags & SYS_PROC) {
645                           next_ptr = rp;          /* run process 'rp' next */
646                           return;
647                   }
648                   /* User process. */
649                   if (rp->p_recent_time < low_time) {
650                           low_time = rp->p_recent_time;   /* record lowest */
```

```
651                             low_rp = rp;                    /* time and proc */
652                     }
653             }
654     }
655     /* The function has scanned through the scheduling queues and determined
656      * that no system tasks wish to run while also recording the user process
657      * (if any) with the lowest recent CPU time. Now test to see if there ARE
658      * any user processes that wish to run.
659      */
660     if (low_rp != NIL_PROC) {
661             next_ptr = low_rp;                      /* run process 'low_rp' next */
662             bill_ptr = low_rp;                      /* bill for system time */
663             return;
664     }
665     /* No other process wants to run, so attempt to run the idle process. */
666     if ( (rp = rdy_head[q]) != NIL_PROC) {
667             next_ptr = rp;                          /* run process 'rp' next */
668             bill_ptr = rp;                          /* bill for system time */
669             return;
670     }
671     panic("no ready process", NO_NUM);
672   }
673
674   /*==========================================================================*
675    *                              balance_queues                              *
676    *==========================================================================*/
677   #define Q_BALANCE_TICKS  100
678   PUBLIC void balance_queues(tp)
679   timer_t *tp;                                      /* watchdog timer pointer */
680   {
681   /* Check entire process table and give all process a higher priority. This
682    * effectively means giving a new quantum. If a process already is at its
683    * maximum priority, its quantum will be renewed.
684    */
685     static timer_t queue_timer;                     /* timer structure to use */
686     register struct proc* rp;                       /* process table pointer  */
687     clock_t next_period;                            /* time of next period  */
688     int ticks_added = 0;                            /* total time added */
689
690     for (rp=BEG_PROC_ADDR; rp<END_PROC_ADDR; rp++) {
691         if (! isemptyp(rp)) {                               /* check slot use */
692             rp->p_recent_time >>= 1;                        /* decay recent cpu */
693             lock(5,"balance_queues");
694             if (rp->p_priority > rp->p_max_priority) {      /* update priority? */
695                 if (rp->p_rts_flags == 0) dequeue(rp);      /* take off queue */
696                 ticks_added += rp->p_quantum_size;          /* do accounting */
697                 rp->p_priority -= 1;                        /* raise priority */
698                 if (rp->p_rts_flags == 0) enqueue(rp);      /* put on queue */
699             }
700             else {
701                 ticks_added += rp->p_quantum_size - rp->p_ticks_left;
702                 rp->p_ticks_left = rp->p_quantum_size;      /* give new quantum */
703             }
704             unlock(5);
705         }
706     }
707   #if DEBUG
708     kprintf("ticks_added: %d\n", ticks_added);
709   #endif
710
711     /* Now schedule a new watchdog timer to balance the queues again.  The
712      * period depends on the total amount of quantum ticks added.
713      */
714     next_period = MAX(Q_BALANCE_TICKS, ticks_added);        /* calculate next */
715     set_timer(&queue_timer, get_uptime() + next_period, balance_queues);
```

```
1   /* Function prototypes. */
2
3   /* main.c */
4   _PROTOTYPE( int  main, (int argc, char **argv)                    );
5
6   /* dmp.c */
7   _PROTOTYPE( int do_fkey_pressed, (message *m)                     );
8   _PROTOTYPE( void mapping_dmp, (void)                              );
9
10  /* dmp_kernel.c */
11  _PROTOTYPE( void proctab_dmp, (void)                              );
12  _PROTOTYPE( void memmap_dmp, (void)                               );
13  _PROTOTYPE( void privileges_dmp, (void)                           );
14  _PROTOTYPE( void sendmask_dmp, (void)                             );
15  _PROTOTYPE( void image_dmp, (void)                                );
16  _PROTOTYPE( void irqtab_dmp, (void)                               );
17  _PROTOTYPE( void kmessages_dmp, (void)                            );
18  _PROTOTYPE( void sched_dmp, (void)                                );
19  _PROTOTYPE( void monparams_dmp, (void)                            );
20  _PROTOTYPE( void kenv_dmp, (void)                                 );
21  _PROTOTYPE( void timing_dmp, (void)                               );
22  _PROTOTYPE( void recent_dmp, (void)                               );
23
24  /* dmp_pm.c */
25  _PROTOTYPE( void mproc_dmp, (void)                                );
26  _PROTOTYPE( void sigaction_dmp, (void)                            );
27  _PROTOTYPE( void holes_dmp, (void)                                );
28
29  /* dmp_fs.c */
30  _PROTOTYPE( void dtab_dmp, (void)                                 );
31  _PROTOTYPE( void fproc_dmp, (void)                                );
32
33  /* dmp_rs.c */
34  _PROTOTYPE( void rproc_dmp, (void)                                );
35
36  /* dmp_ds.c */
37  _PROTOTYPE( void data_store_dmp, (void)                           );
38
```

```
/* This file contains information dump procedures. During the initialization
 * of the Information Service 'known' function keys are registered at the TTY
 * server in order to receive a notification if one is pressed. Here, the
 * corresponding dump procedure is called.
 *
 * The entry points into this file are
 *   handle_fkey:       handle a function key pressed notification
 */

#include "inc.h"

/* Define hooks for the debugging dumps. This table maps function keys
 * onto a specific dump and provides a description for it.
 */
#define NHOOKS 19

struct hook_entry {
        int key;
        void (*function)(void);
        char *name;
} hooks[NHOOKS] = {
        { F1,   proctab_dmp, "Kernel process table" },
        { F2,   memmap_dmp, "Process memory maps" },
        { F3,   image_dmp, "System image" },
        { F4,   privileges_dmp, "Process privileges" },
        { F5,   monparams_dmp, "Boot monitor parameters" },
        { F6,   irqtab_dmp, "IRQ hooks and policies" },
        { F7,   kmessages_dmp, "Kernel messages" },
        { F9,   sched_dmp, "Scheduling queues" },
        { F10,  kenv_dmp, "Kernel parameters" },
        { F11,  timing_dmp, "Timing details (if enabled)" },
        { F12,  recent_dmp, "Recent CPU time" },
        { SF1,  mproc_dmp, "Process manager process table" },
        { SF2,  sigaction_dmp, "Signals" },
        { SF3,  fproc_dmp, "Filesystem process table" },
        { SF4,  dtab_dmp, "Device/Driver mapping" },
        { SF5,  mapping_dmp, "Print key mappings" },
        { SF6,  rproc_dmp, "Reincarnation server process table" },
        { SF7,  holes_dmp, "Memory free list" },
        { SF8,  data_store_dmp, "Data store contents" },
};

/*===========================================================================*
 *                              handle_fkey                                  *
 *===========================================================================*/
#define pressed(k) ((F1<=(k)&&(k)<=F12 && bit_isset(m->FKEY_FKEYS,((k)-F1+1)))\
        || (SF1<=(k) && (k)<=SF12 && bit_isset(m->FKEY_SFKEYS, ((k)-SF1+1))))
PUBLIC int do_fkey_pressed(m)
message *m;                                     /* notification message */
{
  int s, h;

  /* The notification message does not convey any information, other
   * than that some function keys have been pressed. Ask TTY for details.
   */
  m->m_type = FKEY_CONTROL;
  m->FKEY_REQUEST = FKEY_EVENTS;
  if (OK != (s=sendrec(TTY_PROC_NR, m)))
      report("IS", "warning, sendrec to TTY failed", s);

  /* Now check which keys were pressed: F1-F12, SF1-SF12. */
  for(h=0; h < NHOOKS; h++)
      if(pressed(hooks[h].key))
          hooks[h].function();

```

```
196               if (i % 8 == 7) ipc_to[++j] = ' ';
197           }
198           ipc_to[j] = '\0';
199         printf("%8s %4d   %s   %s  %3d %7lu %7lu   %s\n",
200             ip->proc_name, ip->proc_nr,
201                  s_flags_str(ip->flags), s_traps_str(ip->trap_mask),
202           ip->priority, (long)ip->initial_pc, ip->stksize, ipc_to);
203     }
204     printf("\n");
205   }
206
207   /*==========================================================================*
208    *                                recent_dmp                                *
209    *==========================================================================*/
210   PUBLIC void recent_dmp()
211   {
212     register struct proc *rp;
213     static struct proc *oldrp = BEG_PROC_ADDR;
214     register struct priv *sp;
215     int r, i, n = 0;
216
217     /* First obtain a fresh copy of the current process and system table. */
218     if ((r = sys_getprivtab(priv)) != OK) {
219           report("IS","warning: couldn't get copy of system privileges table", r);
220           return;
221     }
222     if ((r = sys_getproctab(proc)) != OK) {
223           report("IS","warning: couldn't get copy of process table", r);
224           return;
225     }
226
227     printf("\n-nr---id--name----flags-pri-privaddr-recent\n");
228
229     for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
230           if (isemptyp(rp)) continue;
231           if (++n > 23) break;
232           if (proc_nr(rp) == IDLE)        printf("(%2d) ", proc_nr(rp));
233           else if (proc_nr(rp) < 0)       printf("[%2d] ", proc_nr(rp));
234           else                            printf(" %2d  ", proc_nr(rp));
235           r = -1;
236           for (sp = &priv[0]; sp < &priv[NR_SYS_PROCS]; sp++)
237                   if (sp->s_proc_nr == rp->p_nr) { r ++; break; }
238           if (r == -1 && ! (rp->p_rts_flags & SLOT_FREE)) {
239                   sp = &priv[USER_PRIV_ID];
240           }
241           printf("(%02u) %-7.7s %s  %2d %4x     %6d",
242                   sp->s_id, rp->p_name,
243                   s_flags_str(sp->s_flags),
244                   rp->p_priority,
245                   rp->p_priv,
246                   rp->p_recent_time
247                   );
248           printf("\n");
249     }
250     if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
251     oldrp = rp;
252   }
253
254   /*==========================================================================*
255    *                                sched_dmp                                 *
256    *==========================================================================*/
257   PUBLIC void sched_dmp()
258   {
259     struct proc *rdy_head[NR_SCHED_QUEUES];
260     struct kinfo kinfo;
```