

```

1  #ifndef PROC_H
2  #define PROC_H
3
4  /* Here is the declaration of the process table.  It contains all process
5   * data, including registers, flags, scheduling priority, memory map,
6   * accounting, message passing (IPC) information, and so on.
7   *
8   * Many assembly code routines reference fields in it.  The offsets to these
9   * fields are defined in the assembler include file sconst.h.  When changing
10  * struct proc, be sure to change sconst.h to match.
11  */
12  #include <minix/com.h>
13  #include "const.h"
14  #include "priv.h"
15
16  struct proc {
17      struct stackframe_s p_reg;      /* process' registers saved in stack frame */
18      struct segframe p_seg;          /* segment descriptors */
19      proc_nr_t p_nr;                 /* number of this process (for fast access) */
20      struct priv *p_priv;             /* system privileges structure */
21      short p_rts_flags;              /* process is runnable only if zero */
22      short p_misc_flags;             /* flags that do not suspend the process */
23
24      char p_priority;                /* current scheduling priority */
25      char p_max_priority;            /* maximum scheduling priority */
26      char p_ticks_left;              /* number of scheduling ticks left */
27      char p_quantum_size;            /* quantum size in ticks */
28
29      struct mem_map p_memmap[NR_LOCAL_SEGS]; /* memory map (T, D, S) */
30
31      clock_t p_user_time;            /* user time in ticks */
32      clock_t p_sys_time;            /* sys time in ticks */
33
34      struct proc *p_nextready;       /* pointer to next ready process */
35      struct proc *p_caller_q;        /* head of list of procs wishing to send */
36      struct proc *p_q_link;          /* link to next proc wishing to send */
37      message *p_messbuf;             /* pointer to passed message buffer */
38      int p_getfrom_e;                /* from whom does process want to receive? */
39      int p_sendto_e;                /* to whom does process want to send? */
40
41      sigset_t p_pending;             /* bit map for pending kernel signals */
42
43      char p_name[P_NAME_LEN];        /* name of the process, including \0 */
44
45      endpoint_t p_endpoint;          /* endpoint number, generation-aware */
46
47      unsigned long p_mess_sent[NR_TASKS + NR_PROCS]; /* number of messages sent to other
48      processes */
49
50  #if DEBUG_SCHED_CHECK
51      int p_ready, p_found;
52  #endif
53  };
54
55  /* Bits for the runtime flags.  A process is runnable iff p_rts_flags == 0. */
56  #define SLOT_FREE      0x01 /* process slot is free */
57  #define NO_PRIORITY    0x02 /* process has been stopped */
58  #define SENDING        0x04 /* process blocked trying to send */
59  #define RECEIVING      0x08 /* process blocked trying to receive */
60  #define SIGNALLED      0x10 /* set when new kernel signal arrives */
61  #define SIG_PENDING    0x20 /* unready while signal being processed */
62  #define P_STOP         0x40 /* set when process is being traced */
63  #define NO_PRIV        0x80 /* keep forked system process from running */
64  #define NO_ENDPOINT    0x100 /* process cannot send or receive messages */

```

```

1  /* This file contains the main program of MINIX as well as its shutdown code.
2  * The routine main() initializes the system and starts the ball rolling by
3  * setting up the process table, interrupt vectors, and scheduling each task
4  * to run to initialize itself.
5  * The routine shutdown() does the opposite and brings down MINIX.
6  *
7  * The entries into this file are:
8  *   main:             MINIX main program
9  *   prepare_shutdown: prepare to take MINIX down
10 /*
11 #include "kernel.h"
12 #include <signal.h>
13 #include <string.h>
14 #include <unistd.h>
15 #include <a.out.h>
16 #include <minix/callnr.h>
17 #include <minix/com.h>
18 #include <minix/endpoint.h>
19 #include "proc.h"
20
21 /* Prototype declarations for PRIVATE functions. */
22 FORWARD _PROTOTYPE( void announce, (void));
23 FORWARD _PROTOTYPE( void shutdown, (timer_t *));
24
25 /*=====*
26 *                               main                               *
27 *=====*/
28 PUBLIC void main()
29 {
30 /* Start the ball rolling. */
31 struct boot_image *ip;      /* boot image pointer */
32 register struct proc *rp;    /* process pointer */
33 register struct priv *sp;    /* privilege structure pointer */
34 register int i, s;
35 int hdrindex;               /* index to array of a.out headers */
36 phys_clicks text_base;
37 vir_clicks text_clicks, data_clicks, st_clicks;
38 reg_t ktsb;                 /* kernel task stack base */
39 struct exec e_hdr;          /* for a copy of an a.out header */
40
41 /* Clear the process table. Anounce each slot as empty and set up mappings
42 * for proc_addr() and proc_nr() macros. Do the same for the table with
43 * privilege structures for the system processes.
44 */
45 for (rp = BEG_PROC_ADDR, i = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++i) {
46     rp->p_rts_flags = SLOT_FREE;      /* initialize free slot */
47     rp->p_nr = i;                     /* proc number from ptr */
48     rp->p_endpoint = _ENDPOINT(0, rp->p_nr); /* generation no. 0 */
49     (pproc_addr + NR_TASKS)[i] = rp;  /* proc ptr from number */
50     memset(rp->p_mess_sent, 0, sizeof(rp->p_mess_sent)); /* sent message counter */
51 }
52 for (sp = BEG_PRIV_ADDR, i = 0; sp < END_PRIV_ADDR; ++sp, ++i) {
53     sp->s_proc_nr = NONE;              /* initialize as free */
54     sp->s_id = i;                     /* priv structure index */
55     ppriv_addr[i] = sp;               /* priv ptr from number */
56 }
57
58 /* Set up proc table entries for processes in boot image. The stacks of the
59 * kernel tasks are initialized to an array in data space. The stacks
60 * of the servers have been added to the data segment by the monitor, so
61 * the stack pointer is set to the end of the data segment. All the
62 * processes are in low memory on the 8086. On the 386 only the kernel
63 * is in low memory, the rest is loaded in extended memory.
64 */
65

```

```

1  /* The kernel call implemented in this file:
2  *   m_type:      SYS_FORK
3  *
4  * The parameters for this kernel call are:
5  *   ml_i1:       PR_SLOT  (child's process table slot)
6  *   ml_i2:       PR_ENDPT (parent, process that forked)
7  */
8
9  #include "../system.h"
10 #include <signal.h>
11
12 #include <minix/endpoint.h>
13
14 #if USE_FORK
15
16 /*=====
17 *                                     do_fork                                     *
18 *=====*/
19 PUBLIC int do_fork(m_ptr)
20 register message *m_ptr;          /* pointer to request message */
21 {
22     /* Handle sys_fork().  PR_ENDPT has forked.  The child is PR_SLOT. */
23     #if (_MINIX_CHIP == _CHIP_INTEL)
24         reg_t old_ldt_sel;
25     #endif
26     register struct proc *rp;          /* process pointer */
27     register struct proc *rpc;         /* child process pointer */
28     struct proc *rpp;                  /* parent process pointer */
29     struct mem_map *map_ptr;           /* virtual address of map inside caller (PM) */
30     int i, gen, r;
31     int p_proc;
32
33     if(!isokendpt(m_ptr->PR_ENDPT, &p_proc))
34         return EINVAL;
35     rpp = proc_addr(p_proc);
36     rpc = proc_addr(m_ptr->PR_SLOT);
37     if (isemptyp(rpp) || ! isemptyp(rpc)) return(EINVAL);
38
39     map_ptr= (struct mem_map *) m_ptr->PR_MEM_PTR;
40
41     /* Copy parent 'proc' struct to child. And reinitialize some fields. */
42     gen = _ENDPOINT_G(rpc->p_endpoint);
43     #if (_MINIX_CHIP == _CHIP_INTEL)
44         old_ldt_sel = rpc->p_seg.p_ldt_sel;    /* backup local descriptors */
45         *rpc = *rpp;                          /* copy 'proc' struct */
46         rpc->p_seg.p_ldt_sel = old_ldt_sel;    /* restore descriptors */
47     #else
48         *rpc = *rpp;                          /* copy 'proc' struct */
49     #endif
50     if(++gen >= _ENDPOINT_MAX_GENERATION) /* increase generation */
51         gen = 1;                          /* generation number wraparound */
52     rpc->p_nr = m_ptr->PR_SLOT;              /* this was obliterated by copy */
53     rpc->p_endpoint = _ENDPOINT(gen, rpc->p_nr); /* new endpoint of slot */
54
55     rpc->p_reg.retreg = 0;                   /* child sees pid = 0 to know it is child */
56     rpc->p_user_time = 0;                    /* set all the accounting times to 0 */
57     rpc->p_sys_time = 0;
58
59     /* Because this is a copy of the parent process, message data is copied over
60     * as well. This should be reset so we have a clean slate.
61     */
62     memset(rpc->p_mess_sent, 0, sizeof(rpc->p_mess_sent));
63
64     /* Reset the number of messages sent by other processes to any previous
65     * process that used the same pid.

```

```
66      */
67      for (rp = BEG_PROC_ADDR, i = rpc->p_nr + NR_TASKS; rp < END_PROC_ADDR; ++rp)
68          rp->p_mess_sent[i] = 0;
69
70      /* Parent and child have to share the quantum that the forked process had,
71       * so that queued processes do not have to wait longer because of the fork.
72       * If the time left is odd, the child gets an extra tick.
73       */
74      rpc->p_ticks_left = (rpc->p_ticks_left + 1) / 2;
75      rpp->p_ticks_left = rpp->p_ticks_left / 2;
76
77      /* If the parent is a privileged process, take away the privileges from the
78       * child process and inhibit it from running by setting the NO_PRIV flag.
79       * The caller should explicitly set the new privileges before executing.
80       */
81      if (priv(rpp)->s_flags & SYS_PROC) {
82          rpc->p_priv = priv_addr(USER_PRIV_ID);
83          rpc->p_rts_flags |= NO_PRIV;
84      }
85
86      /* Calculate endpoint identifier, so caller knows what it is. */
87      m_ptr->PR_ENDPT = rpc->p_endpoint;
88
89      /* Install new map */
90      r = newmap(rpc, map_ptr);
91
92      /* Only one in group should have SIGNALED, child doesn't inherit tracing. */
93      RTS_LOCK_UNSET(rpc, (SIGNALED | SIG_PENDING | P_STOP));
94      sigemptyset(&rpc->p_pending);
95
96      return r;
97  }
98
99  #endif /* USE_FORK */
100
101
```

```

261         /* The function number is magically converted to flags. */
262         if ((xp->p_rts_flags ^ (function << 2)) & SENDING) {
263             return(0); /* not a deadlock */
264         }
265     }
266     return(group_size); /* deadlock found */
267 }
268 }
269 return(0); /* not a deadlock */
270 }
271
272 /*=====
273      *                               mini_send                               *
274      *=====*/
275 PRIVATE int mini_send(caller_ptr, dst_e, m_ptr, flags)
276 register struct proc *caller_ptr; /* who is trying to send a message? */
277 int dst_e; /* to whom is message being sent? */
278 message *m_ptr; /* pointer to message buffer */
279 unsigned flags; /* system call flags */
280 {
281     /* Send a message from 'caller_ptr' to 'dst'. If 'dst' is blocked waiting
282      * for this message, copy the message to it and unblock 'dst'. If 'dst' is
283      * not waiting at all, or is waiting for another source, queue 'caller_ptr'.
284      */
285     register struct proc *dst_ptr;
286     register struct proc **xpp;
287     int dst_p;
288
289     dst_p = _ENDPOINT_P(dst_e);
290     dst_ptr = proc_addr(dst_p);
291
292     if (RTS_ISSET(dst_ptr, NO_ENDPOINT)) return EDSTDIED;
293
294     /* Check if 'dst' is blocked waiting for this message. The destination's
295      * SENDING flag may be set when its SENDREC call blocked while sending.
296      */
297     if ( (RTS_ISSET(dst_ptr, RECEIVING) && !RTS_ISSET(dst_ptr, SENDING)) &&
298         (dst_ptr->p_getfrom_e == ANY
299          || dst_ptr->p_getfrom_e == caller_ptr->p_endpoint)) {
300         /* Destination is indeed waiting for this message. */
301         CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, dst_ptr,
302                 dst_ptr->p_messbuf);
303         RTS_UNSET(dst_ptr, RECEIVING);
304     } else if ( ! (flags & NON_BLOCKING)) {
305         /* Destination is not waiting. Block and dequeue caller. */
306         caller_ptr->p_messbuf = m_ptr;
307         RTS_SET(caller_ptr, SENDING);
308         caller_ptr->p_sendto_e = dst_e;
309
310         /* Process is now blocked. Put in on the destination's queue. */
311         xpp = &dst_ptr->p_caller_q; /* find end of list */
312         while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;
313         *xpp = caller_ptr; /* add caller to end */
314         caller_ptr->p_q_link = NIL_PROC; /* mark new end of list */
315     } else {
316         return(ENOTREADY);
317     }
318
319     /* Increment the counter keeping track of where messages are sent. */
320     ++(caller_ptr->p_mess_sent[dst_ptr->p_nr + NR_TASKS]);
321
322     return(OK);
323 }
324
325 /*=====

```

```
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 messaging_dmp, (void)                   );
15 _PROTOTYPE( void sendmask_dmp, (void)                   );
16 _PROTOTYPE( void image_dmp, (void)                      );
17 _PROTOTYPE( void irqtab_dmp, (void)                      );
18 _PROTOTYPE( void kmessages_dmp, (void)                   );
19 _PROTOTYPE( void sched_dmp, (void)                      );
20 _PROTOTYPE( void monparams_dmp, (void)                   );
21 _PROTOTYPE( void kenv_dmp, (void)                       );
22 _PROTOTYPE( void timing_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
```

```

1  /* This file contains information dump procedures. During the initialization
2  * of the Information Service 'known' function keys are registered at the TTY
3  * server in order to receive a notification if one is pressed. Here, the
4  * corresponding dump procedure is called.
5  *
6  * The entry points into this file are
7  *   handle_fkey:      handle a function key pressed notification
8  */
9
10 #include "inc.h"
11
12 /* Define hooks for the debugging dumps. This table maps function keys
13  * onto a specific dump and provides a description for it.
14  */
15 #define NHOOKS 18
16
17 struct hook_entry {
18     int key;
19     void (*function)(void);
20     char *name;
21 } hooks[NHOOKS] = {
22     { F1,   proctab_dmp, "Kernel process table" },
23     { F2,   memmap_dmp, "Process memory maps" },
24     { F3,   image_dmp, "System image" },
25     /* { F4,   privileges_dmp, "Process privileges" }, */
26     { F4,   messaging_dmp, "Messaging activity" },
27     { F5,   monparams_dmp, "Boot monitor parameters" },
28     { F6,   irqtab_dmp, "IRQ hooks and policies" },
29     { F7,   kmessages_dmp, "Kernel messages" },
30     { F9,   sched_dmp, "Scheduling queues" },
31     { F10,  kenv_dmp, "Kernel parameters" },
32     { F11,  timing_dmp, "Timing details (if enabled)" },
33     { SF1,  mproc_dmp, "Process manager process table" },
34     { SF2,  sigaction_dmp, "Signals" },
35     { SF3,  fproc_dmp, "Filesystem process table" },
36     { SF4,  dtab_dmp, "Device/Driver mapping" },
37     { SF5,  mapping_dmp, "Print key mappings" },
38     { SF6,  rproc_dmp, "Reincarnation server process table" },
39     { SF7,  holes_dmp, "Memory free list" },
40     { SF8,  data_store_dmp, "Data store contents" },
41 };
42
43 /*=====*
44  *                               handle_fkey                               *
45  *=====*/
46 #define pressed(k) ((F1<=(k)&&(k)<=F12 && bit_isset(m->FKEY_FKEYS, ((k)-F1+1)))\
47  || (SF1<=(k) && (k)<=SF12 && bit_isset(m->FKEY_SFKEYS, ((k)-SF1+1))))
48 PUBLIC int do_fkey_pressed(m)
49 message *m;                                /* notification message */
50 {
51     int s, h;
52
53     /* The notification message does not convey any information, other
54      * than that some function keys have been pressed. Ask TTY for details.
55      */
56     m->m_type = FKEY_CONTROL;
57     m->FKEY_REQUEST = FKEY_EVENTS;
58     if (OK != (s=sendrec(TTY_PROC_NR, m)))
59         report("IS", "warning, sendrec to TTY failed", s);
60
61     /* Now check which keys were pressed: F1-F12, SF1-SF12. */
62     for(h=0; h < NHOOKS; h++)
63         if(pressed(hooks[h].key))
64             hooks[h].function();
65

```

```

326
327     return str;
328 }
329
330 /*=====
331 *                               privileges_dmp                               *
332 *=====*/
333 PUBLIC void privileges_dmp()
334 {
335     register struct proc *rp;
336     static struct proc *oldrp = BEG_PROC_ADDR;
337     register struct priv *sp;
338     int r, i, n = 0;
339
340     /* First obtain a fresh copy of the current process and system table. */
341     if ((r = sys_getprivtab(priv)) != OK) {
342         report("IS", "warning: couldn't get copy of system privileges table", r);
343         return;
344     }
345     if ((r = sys_getproctab(proc)) != OK) {
346         report("IS", "warning: couldn't get copy of process table", r);
347         return;
348     }
349
350     printf("\n--nr-id-name---- -flags- -traps- grants -ipc_to-- -system calls--\n");
351
352     for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
353         if (isemptyp(rp)) continue;
354         if (++n > 23) break;
355         if (proc_nr(rp) == IDLE)         printf("(%2d) ", proc_nr(rp));
356         else if (proc_nr(rp) < 0)        printf("[%2d] ", proc_nr(rp));
357         else                             printf(" %2d ", proc_nr(rp));
358         r = -1;
359         for (sp = &priv[0]; sp < &priv[NR_SYS_PROCS]; sp++)
360             if (sp->s_proc_nr == rp->p_nr) { r++; break; }
361         if (r == -1 && ! (rp->p_rts_flags & SLOT_FREE)) {
362             sp = &priv[USER_PRIV_ID];
363         }
364         printf("(%02u) %-7.7s %s    %s %7d",
365             sp->s_id, rp->p_name,
366             s_flags_str(sp->s_flags), s_traps_str(sp->s_trap_mask),
367             sp->s_grant_entries);
368         for (i=0; i < NR_SYS_PROCS; i += BITCHUNK_BITS) {
369             printf(" %04x", get_sys_bits(sp->s_ipc_to, i));
370         }
371
372         printf(" ");
373         for (i=0; i < NR_SYS_CALLS; i += BITCHUNK_BITS) {
374             printf(" %04x", sp->s_k_call_mask[i/BITCHUNK_BITS]);
375         }
376         printf("\n");
377     }
378
379     if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
380     oldrp = rp;
381
382 }
383
384 /*=====
385 *                               messaging_dmp                               *
386 *=====*/
387 PUBLIC void messaging_dmp()
388 {
389     /* Messaging grid dump */
390

```



```

391 register struct proc *rrp, *crp;
392 static struct proc *oldrrp = BEG_PROC_ADDR, *oldcrp = BEG_PROC_ADDR;
393 int r, row, col;
394
395 /* First obtain a fresh copy of the current process table. */
396 if ((r = sys_getproctab(proc)) != OK) {
397     report("IS", "warning: couldn't get copy of process table", r);
398     return;
399 }
400
401 printf("\nNumber of messages sent from process in given row to process in given
column:\n");
402
403 /* iterate through the proc table (rows of matrix) */
404 for (rrp = oldrrp, row = 0; rrp < END_PROC_ADDR; rrp++) {
405     if (isemptytyp(rrp)) continue;
406     if (++row > 23) break;
407
408     /* iterate through the message array (columns of matrix) */
409     for (crp = oldcrp, col = 0; crp < END_PROC_ADDR; crp++) {
410         if (isemptytyp(crp)) continue;
411         if (++col > 7) break;
412         if (row == 1) {
413             if (col == 1)
414                 printf("process names|");
415             else {
416                 if (strcmp(crp->p_name, "<unset>"))
417                     printf("%10s|", crp->p_name);
418                 else
419                     printf(" <pid> %3d|", crp->p_nr);
420             }
421         }
422         else {
423             if (col == 1) {
424                 if (strcmp(rrp->p_name, "<unset>"))
425                     printf("%13s|", rrp->p_name);
426                 else
427                     printf("      <pid> %3d|", rrp->p_nr);
428             }
429             else
430                 printf("%10u|", rrp->p_mess_sent[crp->p_nr +
NR_TASKS]);
431         }
432     }
433     printf("\n");
434 }
435
436 /* handle the paging logic */
437 #define LTR_PAGING 1
438 #if LTR_PAGING
439     /* left-to-right, top-to-bottom paging */
440     if (crp == END_PROC_ADDR) {
441         crp = BEG_PROC_ADDR;
442         if (rrp == END_PROC_ADDR)
443             rrp = BEG_PROC_ADDR;
444         else
445             printf("--more-- <Û\r");
446     }
447     else {
448         rrp = oldrrp;
449         printf("--more-- >>\r");
450     }
451 #else
452     /* top-to-bottom, left-to-right paging */
453     if (rrp == END_PROC_ADDR) {
454         rrp = BEG_PROC_ADDR;
455         if (crp == END_PROC_ADDR)
456             crp = BEG_PROC_ADDR;
457         else
458             printf("--more-- @>\r");
459     }
460     else
461         printf("\n");
462 #endif
463 }
464
465 #endif
466
467 #endif
468
469 #endif
470
471 #endif
472
473 #endif
474
475 #endif
476
477 #endif
478
479 #endif
480
481 #endif
482
483 #endif
484
485 #endif
486
487 #endif
488
489 #endif
490
491 #endif
492
493 #endif
494
495 #endif
496
497 #endif
498
499 #endif
500
501 #endif
502
503 #endif
504
505 #endif
506
507 #endif
508
509 #endif
510
511 #endif
512
513 #endif
514
515 #endif
516
517 #endif
518
519 #endif
520
521 #endif
522
523 #endif
524
525 #endif
526
527 #endif
528
529 #endif
530
531 #endif
532
533 #endif
534
535 #endif
536
537 #endif
538
539 #endif
540
541 #endif
542
543 #endif
544
545 #endif
546
547 #endif
548
549 #endif
550
551 #endif
552
553 #endif
554
555 #endif
556
557 #endif
558
559 #endif
560
561 #endif
562
563 #endif
564
565 #endif
566
567 #endif
568
569 #endif
570
571 #endif
572
573 #endif
574
575 #endif
576
577 #endif
578
579 #endif
580
581 #endif
582
583 #endif
584
585 #endif
586
587 #endif
588
589 #endif
590
591 #endif
592
593 #endif
594
595 #endif
596
597 #endif
598
599 #endif
600
601 #endif
602
603 #endif
604
605 #endif
606
607 #endif
608
609 #endif
610
611 #endif
612
613 #endif
614
615 #endif
616
617 #endif
618
619 #endif
620
621 #endif
622
623 #endif
624
625 #endif
626
627 #endif
628
629 #endif
630
631 #endif
632
633 #endif
634
635 #endif
636
637 #endif
638
639 #endif
640
641 #endif
642
643 #endif
644
645 #endif
646
647 #endif
648
649 #endif
650
651 #endif
652
653 #endif
654
655 #endif
656
657 #endif
658
659 #endif
660
661 #endif
662
663 #endif
664
665 #endif
666
667 #endif
668
669 #endif
670
671 #endif
672
673 #endif
674
675 #endif
676
677 #endif
678
679 #endif
680
681 #endif
682
683 #endif
684
685 #endif
686
687 #endif
688
689 #endif
690
691 #endif
692
693 #endif
694
695 #endif
696
697 #endif
698
699 #endif
700
701 #endif
702
703 #endif
704
705 #endif
706
707 #endif
708
709 #endif
710
711 #endif
712
713 #endif
714
715 #endif
716
717 #endif
718
719 #endif
720
721 #endif
722
723 #endif
724
725 #endif
726
727 #endif
728
729 #endif
730
731 #endif
732
733 #endif
734
735 #endif
736
737 #endif
738
739 #endif
740
741 #endif
742
743 #endif
744
745 #endif
746
747 #endif
748
749 #endif
750
751 #endif
752
753 #endif
754
755 #endif
756
757 #endif
758
759 #endif
760
761 #endif
762
763 #endif
764
765 #endif
766
767 #endif
768
769 #endif
770
771 #endif
772
773 #endif
774
775 #endif
776
777 #endif
778
779 #endif
780
781 #endif
782
783 #endif
784
785 #endif
786
787 #endif
788
789 #endif
790
791 #endif
792
793 #endif
794
795 #endif
796
797 #endif
798
799 #endif
800
801 #endif
802
803 #endif
804
805 #endif
806
807 #endif
808
809 #endif
810
811 #endif
812
813 #endif
814
815 #endif
816
817 #endif
818
819 #endif
820
821 #endif
822
823 #endif
824
825 #endif
826
827 #endif
828
829 #endif
830
831 #endif
832
833 #endif
834
835 #endif
836
837 #endif
838
839 #endif
840
841 #endif
842
843 #endif
844
845 #endif
846
847 #endif
848
849 #endif
850
851 #endif
852
853 #endif
854
855 #endif
856
857 #endif
858
859 #endif
860
861 #endif
862
863 #endif
864
865 #endif
866
867 #endif
868
869 #endif
870
871 #endif
872
873 #endif
874
875 #endif
876
877 #endif
878
879 #endif
880
881 #endif
882
883 #endif
884
885 #endif
886
887 #endif
888
889 #endif
890
891 #endif
892
893 #endif
894
895 #endif
896
897 #endif
898
899 #endif
900
901 #endif
902
903 #endif
904
905 #endif
906
907 #endif
908
909 #endif
910
911 #endif
912
913 #endif
914
915 #endif
916
917 #endif
918
919 #endif
920
921 #endif
922
923 #endif
924
925 #endif
926
927 #endif
928
929 #endif
930
931 #endif
932
933 #endif
934
935 #endif
936
937 #endif
938
939 #endif
940
941 #endif
942
943 #endif
944
945 #endif
946
947 #endif
948
949 #endif
950
951 #endif
952
953 #endif
954
955 #endif
956
957 #endif
958
959 #endif
960
961 #endif
962
963 #endif
964
965 #endif
966
967 #endif
968
969 #endif
970
971 #endif
972
973 #endif
974
975 #endif
976
977 #endif
978
979 #endif
980
981 #endif
982
983 #endif
984
985 #endif
986
987 #endif
988
989 #endif
990
991 #endif
992
993 #endif
994
995 #endif
996
997 #endif
998
999 #endif
1000
1001 #endif
1002
1003 #endif
1004
1005 #endif
1006
1007 #endif
1008
1009 #endif
1010
1011 #endif
1012
1013 #endif
1014
1015 #endif
1016
1017 #endif
1018
1019 #endif
1020
1021 #endif
1022
1023 #endif
1024
1025 #endif
1026
1027 #endif
1028
1029 #endif
1030
1031 #endif
1032
1033 #endif
1034
1035 #endif
1036
1037 #endif
1038
1039 #endif
1040
1041 #endif
1042
1043 #endif
1044
1045 #endif
1046
1047 #endif
1048
1049 #endif
1050
1051 #endif
1052
1053 #endif
1054
1055 #endif
1056
1057 #endif
1058
1059 #endif
1060
1061 #endif
1062
1063 #endif
1064
1065 #endif
1066
1067 #endif
1068
1069 #endif
1070
1071 #endif
1072
1073 #endif
1074
1075 #endif
1076
1077 #endif
1078
1079 #endif
1080
1081 #endif
108
```

```

455     }
456     else {
457         crp = oldcrp;
458         printf("--more-- vv\r");
459     }
460 #endif
461     oldcrp = crp;
462     oldrrp = rrp;
463 }
464
465 /*=====
466 *                                     sendmask_dmp                                     *
467 *=====*/
468 PUBLIC void sendmask_dmp()
469 {
470     register struct proc *rp;
471     static struct proc *oldrp = BEG_PROC_ADDR;
472     int r, i, j, n = 0;
473
474     /* First obtain a fresh copy of the current process table. */
475     if ((r = sys_getproctab(proc)) != OK) {
476         report("IS", "warning: couldn't get copy of process table", r);
477         return;
478     }
479
480     printf("\n\n");
481     printf("Sendmask dump for process table. User processes (*) don't have [].");
482     printf("\n");
483     printf("The rows of bits indicate to which processes each process may send.");
484     printf("\n\n");
485
486     #if DEAD_CODE
487         printf(" ");
488         for (j=proc_nr(BEG_PROC_ADDR); j< INIT_PROC_NR+1; j++) {
489             printf("%3d", j);
490         }
491         printf(" *\n");
492
493         for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
494             if (isemptyp(rp)) continue;
495             if (++n > 20) break;
496
497             printf("%8s ", rp->p_name);
498             if (proc_nr(rp) == IDLE)         printf("(%2d ", proc_nr(rp));
499             else if (proc_nr(rp) < 0)        printf("[%2d] ", proc_nr(rp));
500             else                             printf(" %2d ", proc_nr(rp));
501
502             for (j=proc_nr(BEG_PROC_ADDR); j<INIT_PROC_NR+2; j++) {
503                 if (isallowed(rp->p_sendmask, j)) printf(" 1 ");
504                 else                             printf(" 0 ");
505             }
506             printf("\n");
507         }
508         if (rp == END_PROC_ADDR) { printf("\n"); rp = BEG_PROC_ADDR; }
509         else printf("--more--\r");
510         oldrp = rp;
511     #endif
512 }
513
514 PRIVATE char *p_rts_flags_str(int flags)
515 {
516     static char str[10];
517     str[0] = (flags & NO_PRIORITY) ? 's' : '-';
518     str[1] = (flags & SENDING) ? 'S' : '-';
519     str[2] = (flags & RECEIVING) ? 'R' : '-';

```