```
#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 */

  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 */

  unsigned long p_mess_sent[NR_TASKS + NR_PROCS]; /* number of messages sent to other
  processes */

#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 */

```

```
/* 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 */
        memset(rp->p_mess_sent, 0, sizeof(rp->p_mess_sent)); /* sent message counter */
  }
  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 *rp;             /* process pointer */
  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;

  /* Because this is a copy of the parent process, message data is copied over
   * as well. This should be reset so we have a clean slate.
   */
  memset(rpc->p_mess_sent, 0, sizeof(rpc->p_mess_sent));

  /* Reset the number of messages sent by other processes to any previous
   * 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 explicitely 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
```

```
/* 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 18

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" }, */
        { F4,   messaging_dmp, "Messaging activity" },
        { 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)" },
        { 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();

```

```
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 (isemptyp(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 (isemptyp(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 +
431                                           NR_TASKS]);
432                   }
433           }
434           printf("\n");
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)       rrp = BEG_PROC_ADDR;
443           else                            printf("--more-- <Ù\r");
444     }
445     else {
446           rrp = oldrrp;
447           printf("--more-- >>\r");
448     }
449   #else
450     /* top-to-bottom, left-to-right paging */
451     if (rrp == END_PROC_ADDR) {
452           rrp = BEG_PROC_ADDR;
453           if (crp == END_PROC_ADDR)       crp = BEG_PROC_ADDR;
454           else                            printf("--more-- ©>\r");
```

```
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' : '-';
```