```
#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 */

```

```
 65  /* These runtime flags can be tested and manipulated by these macros. */
 66
 67  #define RTS_ISSET(rp, f) (((rp)->p_rts_flags & (f)) == (f))
 68
 69
 70  /* Set flag and dequeue if the process was runnable. */
 71  #define RTS_SET(rp, f)                                                  \
 72          do {                                                            \
 73                  if(!(rp)->p_rts_flags) { dequeue(rp); }                 \
 74                  (rp)->p_rts_flags |=  (f);                              \
 75          } while(0)
 76
 77  /* Clear flag and enqueue if the process was not runnable but is now. */
 78  #define RTS_UNSET(rp, f)                                                \
 79          do {                                                            \
 80                  int rts;                                                \
 81                  rts = (rp)->p_rts_flags;                                        \
 82                  (rp)->p_rts_flags &= ~(f);                              \
 83                  if(rts && !(rp)->p_rts_flags) { enqueue(rp); }          \
 84          } while(0)
 85
 86  /* Set flag and dequeue if the process was runnable. */
 87  #define RTS_LOCK_SET(rp, f)                                             \
 88          do {                                                            \
 89                  if(!(rp)->p_rts_flags) { lock_dequeue(rp); }            \
 90                  (rp)->p_rts_flags |=  (f);                              \
 91          } while(0)
 92
 93  /* Clear flag and enqueue if the process was not runnable but is now. */
 94  #define RTS_LOCK_UNSET(rp, f)                                           \
 95          do {                                                            \
 96                  int rts;                                                \
 97                  rts = (rp)->p_rts_flags;                                        \
 98                  (rp)->p_rts_flags &= ~(f);                              \
 99                  if(rts && !(rp)->p_rts_flags) { lock_enqueue(rp); }     \
100          } while(0)
101
102  /* Set flags to this value. */
103  #define RTS_LOCK_SETFLAGS(rp, f)                                        \
104          do {                                                            \
105                  if(!(rp)->p_rts_flags && (f)) { lock_dequeue(rp); }     \
106                  (rp)->p_rts_flags = (f);                                        \
107          } while(0)
108
109  /* Misc flags */
110  #define REPLY_PENDING   0x01    /* reply to IPC_REQUEST is pending */
111  #define MF_VM           0x08    /* process uses VM */
112
113  /* Scheduling priorities for p_priority. Values must start at zero (highest
114   * priority) and increment.  Priorities of the processes in the boot image
115   * can be set in table.c. IDLE must have a queue for itself, to prevent low
116   * priority user processes to run round-robin with IDLE.
117   */
118  #define NR_SCHED_QUEUES   16    /* MUST equal minimum priority + 1 */
119  #define TASK_Q             0    /* highest, used for kernel tasks */
120  #define MAX_USER_Q         0    /* highest priority for user processes */
121  #define USER_Q             7    /* default (should correspond to nice 0) */
122  #define MIN_USER_Q        14    /* minimum priority for user processes */
123  #define IDLE_Q            15    /* lowest, only IDLE process goes here */
124
125  /* Magic process table addresses. */
126  #define BEG_PROC_ADDR (&proc[0])
127  #define BEG_USER_ADDR (&proc[NR_TASKS])
128  #define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
129
```

```
130   #define NIL_PROC          ((struct proc *) 0)
131   #define NIL_SYS_PROC      ((struct proc *) 1)
132   #define cproc_addr(n)     (&(proc + NR_TASKS)[(n)])
133   #define proc_addr(n)      (pproc_addr + NR_TASKS)[(n)]
134   #define proc_nr(p)        ((p)->p_nr)
135
136   #define isokprocn(n)      ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
137   #define isemptyn(n)       isemptyp(proc_addr(n))
138   #define isemptyp(p)       ((p)->p_rts_flags == SLOT_FREE)
139   #define iskernelp(p)      iskerneln((p)->p_nr)
140   #define iskerneln(n)      ((n) < 0)
141   #define isuserp(p)        isusern((p)->p_nr)
142   #define isusern(n)        ((n) >= 0)
143
144   /* The process table and pointers to process table slots. The pointers allow
145    * faster access because now a process entry can be found by indexing the
146    * pproc_addr array, while accessing an element i requires a multiplication
147    * with sizeof(struct proc) to determine the address.
148    */
149   EXTERN struct proc proc[NR_TASKS + NR_PROCS];   /* process table */
150   EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
151   EXTERN struct proc *rdy_head[NR_SCHED_QUEUES]; /* ptrs to ready list headers */
152   EXTERN struct proc *rdy_tail[NR_SCHED_QUEUES]; /* ptrs to ready list tails */
153
154   #endif /* PROC_H */
155
```