```c
/* 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.
```

```
     */
    for (rp = BEG_PROC_ADDR, i = rpc->p_nr + NR_TASKS; rp < END_PROC_ADDR; ++rp)
          rp->p_mess_sent[i] = 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;

    /* If the parent is a privileged process, take away the privileges from the
     * child process and inhibit it from running by setting the NO_PRIV flag.
     * The caller should explicitely set the new privileges before executing.
     */
    if (priv(rpp)->s_flags & SYS_PROC) {
        rpc->p_priv = priv_addr(USER_PRIV_ID);
        rpc->p_rts_flags |= NO_PRIV;
    }

    /* Calculate endpoint identifier, so caller knows what it is. */
    m_ptr->PR_ENDPT = rpc->p_endpoint;

    /* Install new map */
    r = newmap(rpc, map_ptr);

    /* Only one in group should have SIGNALED, child doesn't inherit tracing. */
    RTS_LOCK_UNSET(rpc, (SIGNALED | SIG_PENDING | P_STOP));
    sigemptyset(&rpc->p_pending);

    return r;
  }

  #endif /* USE_FORK */


```