Poniżej wklejone są fragmenty kodu jądra, serwerów oraz sterownika. Kod ten przedstawia zarys mechanizmu używanego przy wywołaniach usług systemowych przez procesy użytkownika. W miejscach, w których występuje zmiana kontekstu jest to zasygnalizowane w komentarzu. Niskopoziomowa zasada działania IPC systemowego została przytoczona tylko przy pierwszym wywołaniu send/receive.

```
1: //////////
2: // test.c

3: ssize_t ret = read(fd, buf, nbytes);

4:
5: ////////////////////
6: // lib/posix/_read.c
7:
8: PUBLIC ssize_t read(fd, buffer, nbytes)

9: int fd;
10: void *buffer;
11: size_t nbytes;

12: {
13:   message m;
14:
15:   m.m1_i1 = fd;

16:   m.m1_i2 = nbytes;
17:   m.m1_p1 = (char *) buffer;

18:   return(_syscall(FS, READ, &m));

19: }
20:
21: //////////////////////
22: // lib/other/syscall.c
23:
24: PUBLIC int _syscall(who, syscallnr, msgptr)

25: int who;
26: int syscallnr;
27: register message *msgptr;

28: {
29:   int status;
30:
31:   msgptr->m_type = syscallnr;

32:   status = _sendrec(who, msgptr);
33:

34:   [...]
35:
36:   return(msgptr->m_type);
```

```
}

//////////////////////
// lib/i386/rts/_ipc.s
__sendrec:

        push    ebp
        mov     ebp, esp
        push    ebx
        mov     eax, SRC_DST(ebp)       ! eax = dest-src

        mov     ebx, MESSAGE(ebp)       ! ebx = message pointer

        mov     ecx, SENDREC            ! _sendrec(srcdest, ptr)

        int     SYSVEC                  ! trap to the kernel
        pop     ebx
        pop     ebp

        ret


/////////////////
///////////////// SWITCH
/////////////////

// kernel/proc.c

UBLIC int sys_call(call_nr, src_dst_e, m_ptr, bit_map)

int call_nr;            /* system call number and flags */
int src_dst_e;          /* src to receive from or dst to send to */

message *m_ptr;         /* pointer to message in the caller's space */
long bit_map;           /* notification event set or flags */

{
/* System calls are done by trapping to the kernel with an INT instruction.
 * The trap is caught and sys_call() is called to send or receive a message
 * (or both). The caller is always given by 'proc_ptr'.
 */


  [...]
  /* If the call is to send to a process, i.e., for SEND,
   * SENDREC or NOTIFY, verify that the caller is allowed to send to

   * the given destination.
   */
  if (call_nr == SENDREC)

  {
      if (! get_sys_bit(priv(caller_ptr)->s_ipc_sendrec,

          nr_to_id(src_dst_p))) {
          [...]
```

```
80:             return(ECALLDENIED);    /* call denied by ipc mask */
81:         }
82:     }

83:
84:     [...]
85:
86:     /* Check if the process has privileges for the requested call. Calls to the
87:      * kernel may only be SENDREC, because tasks always reply and may not block
88:      * if the caller doesn't do receive().
89:      */
90:     if (!(priv(caller_ptr)->s_trap_mask & (1 << call_nr))) {
91:       [...]
92:       return(ETRAPDENIED);        /* trap denied by mask or kernel */
93:     }
94:     if ((iskerneln(src_dst_p) && call_nr != SENDREC && call_nr != RECEIVE))
{
95:       return(ETRAPDENIED);        /* trap denied by mask or kernel */
96:     }
97:
98:     [...]
99:     if (group_size = deadlock(call_nr, caller_ptr, src_dst_p)) {
100:       return(ELOCKED);
101:     }
102:
103:     caller_ptr->p_misc_flags |= REPLY_PENDING;
104:     result = mini_send(caller_ptr, src_dst_e, m_ptr, 0);
105:     if (result != OK)
106:           return EBADCALL;
107:     result = mini_receive(caller_ptr, src_dst_e, m_ptr, 0);
108:
109:     return result;
110: }
111:
112: ////////////////
113: // kernel/proc.c
114: PRIVATE int mini_send(caller_ptr, dst_e, m_ptr, flags)
115:     register struct proc *caller_ptr;       /* who is trying to send a message?
*/
116:     int dst_e;                              /* to whom is message being sent? */
```

```
117:    message *m_ptr;                     /* pointer to message buffer */
118:    int flags;
119: {
120:    [...]
121:    dst_p = _ENDPOINT_P(dst_e);
122:    dst_ptr = proc_addr(dst_p);
123:
124:    if (RTS_ISSET(dst_ptr, NO_ENDPOINT))
125:            return EDSTDIED;
126:
127:    /* Check if 'dst' is blocked waiting for this message. The destination's
128:     *    * SENDING flag may be set when its SENDREC call blocked while sending.
129:     *       */
130:    if (WILLRECEIVE(dst_ptr, caller_ptr->p_endpoint)) {
131:            /* Destination is indeed waiting for this message. */
132:            CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, dst_ptr,
133:                                    dst_ptr->p_messbuf);
134:            RTS_UNSET(dst_ptr, RECEIVING);
135:    } else {
136:            if(flags & NON_BLOCKING) {
137:                    return(ENOTREADY);
138:            }
139:
140:            /* Destination is not waiting.  Block and dequeue caller. */
141:            caller_ptr->p_messbuf = m_ptr;
142:            RTS_SET(caller_ptr, SENDING);
143:            caller_ptr->p_sendto_e = dst_e;
144:
145:            /* Process is now blocked.  Put in on the destination's queue. */
146:            [...]
147:    }
148:    return(OK);
149: }
150:
151:
152:
153: ////////////////////
154: //////////////////// SWITCH
155: ////////////////////
156: // servers/vfs/read.c
157:
158: PUBLIC int read_write(rw_flag)
```

```
159: int rw_flag;            /* READING or WRITING */
160: {
161: /* pobranie informacji o odpowiedzialnym sterowniku */
162: [...]

163: }
164:
165: PRIVATE int fs_sendrec_f(char *file, int line, endpoint_t fs_e, message
*reqm)

166: {
167: [...]
168:   for (;;) {

169:     /* Do the actual send, receive */
170:     r=sendrec(fs_e, reqm);

171:     if(r == OK) {
172:       /* Sendrec was okay */

173:       break;
174:     }
175:
176:     /* Dead driver */
177:     if (r == EDEADSRCDST || r == EDSTDIED || r == ESRCDIED) {

178:       old_driver_e = NONE;
179:       /* Find old driver by endpoint */
180:       for (vmp = &vmnt[0]; vmp < &vmnt[NR_MNTS]; ++vmp) {

181:         if (vmp->m_fs_e == fs_e) {   /* found FS */

182:           old_driver_e = vmp->m_driver_e;
183:           dmap_unmap_by_endpt(old_driver_e); /* unmap driver */

184:           break;
185:         }
186:       }
187:
188:       /* No FS ?? */

189:       if (old_driver_e == NONE)
190:         panic(__FILE__, "VFSdead_driver: couldn't find FS\n", fs_e);

191:
192:       /* Wait for a new driver. */
193:       for (;;) {

194:         [...]
195:         r = receive(RS_PROC_NR, &m);

196:       }
```

```
197:     }
198:
199: }
200:
201: ////////////////
202: //////////////// SWITCH
203: ////////////////
204: // lib/driver.c
205: PUBLIC void driver_task(dp)
206:    struct driver *dp;  /* Device dependent entry points. */
207: {
208:    [...]
209:    s = receive(ANY, &mess);
210:    [...]
211:    dp->do_rdwt(dp, &mess, 0);
212:    return r;
213: }
214:
215: //////////////////////////
216: ////////////////////////// SWITCH
217: //////////////////////////
218: // drivers/floppy/floppy.c
219:
220: PRIVATE int f_transfer(proc_nr, opcode, pos64, iov, nr_req, safe)
221: {
222:    [...]
223:    /* Set the stepping rate and data rate */
224:    if (f_dp != prev_dp) {
225:            cmd[0] = FDC_SPECIFY;
226:            cmd[1] = f_dp->spec1;
227:            cmd[2] = SPEC2;
228:            (void) fdc_command(cmd, 3);
229:            if ((s=sys_outb(FDC_RATE, f_dp->rate)) != OK)
230:                    panic("FLOPPY","Sys_outb failed", s);
231:            prev_dp = f_dp;
232:    }
233:    [...]
234:    if (r == OK && opcode == DEV_SCATTER_S) {
235:            /* Copy the user bytes to the DMA buffer. */
236:            if(safe) {
237:                    s=sys_safecopyfrom(proc_nr, *ug, *up,
```

```
238:                                    (vir_bytes) tmp_buf,
239:                                    (phys_bytes) SECTOR_SIZE, D);
240:                 if(s != OK)
241:                         panic("FLOPPY", "sys_safecopyfrom failed", s);
242:         } else {
243:                 assert(proc_nr == SELF);
244:                 memcpy(tmp_buf, (void *) (*ug + *up), SECTOR_SIZE);
245:         }
246:     }
247:     [...]
248:     return(OK);
249: }
```