

IPC in Minix

Sistemi operativi
Lez. 25-26

Due forme di IPC

- Minix distingue due forme diverse di meccanismi di comunicazione tra processi:
 - IPC tra processi utente
 - IPC tra processi utente e processi di sistema

IPC tra processi utente

- Si possono utilizzare i meccanismi standard di comunicazione introdotti da Unix
 - Pipe anonime
 - File
 - Named Pipe
 - Socket
 - Signal

Comunicazione tra processi user e di sistema

- Strumento principale in questo caso sono le system call
- Bisogna però tenere presente che in Minix ci sono processi di sistema che operano in user space, prevedere quindi per questi processi un meccanismo privilegiato di accesso ai servizi del Kernel

Syscall

- In Minix distinguiamo tra:
 - **System call**: tutte le chiamate di sistema POSIX effettuate a dalle applicazioni verso i server di sistema
 - **Kernel Call**: le chiamate fatte al SYSTEM_TASK esclusivamente dai server, per lo la realizzazione delle system call
 - **IPC call**: le chiamate che il SYSTEM_TASK inoltra effettivamente al kernel per l'esecuzione delle syscall

IPC in Minix

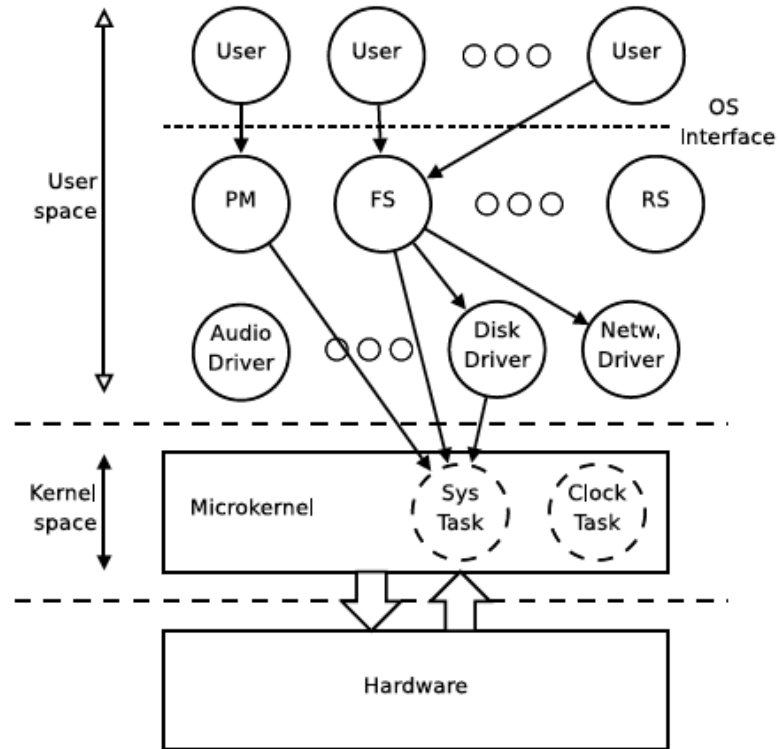


Figure 2: The MINIX 3 architecture and some IPC paths.

Syscall

- Il meccanismo attraverso cui un'applicazione richiede l'esecuzione di una syscall è lo scambio di messaggi
- Un'applicazione che vuole effettuare una system call deve mandare un messaggio al server MM o FS, che poi provvederanno a contattare gli strati inferiori del sistema
- L'unica primitiva che le applicazioni possono usare in questo contesto è **sendrec**

Messaggi in MINIX

- Il meccanismo di comunicazione adottato è il rendezvous
 - Quando un processo invia un messaggio, resta bloccato finché il processo destinatario non legge il messaggio
 - Un processo che effettua una receive si blocca sempre a meno che non ci sia già un messaggio pronto da leggere
- Il rendezvous è stato prescelto come modalità di comunicazione perché evita i problemi connessi al buffering

Primitive di comunicazione

```
#define echo      _echo
#define notify   _notify
#define sendrec  _sendrec
#define receive  _receive
#define send     _send
```

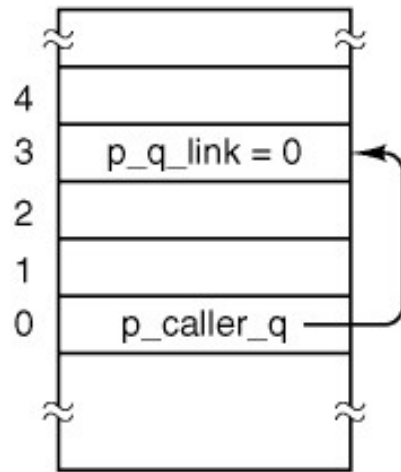
```
PROTOTYPE( int echo, (message *m_ptr) );
PROTOTYPE( int notify, (int dest) );
PROTOTYPE( int sendrec, (int src_dest, message *m_ptr) );
PROTOTYPE( int receive, (int src, message *m_ptr) );
PROTOTYPE( int send, (int dest, message *m_ptr) );
```

`send (int dest, message *m_ptr)`

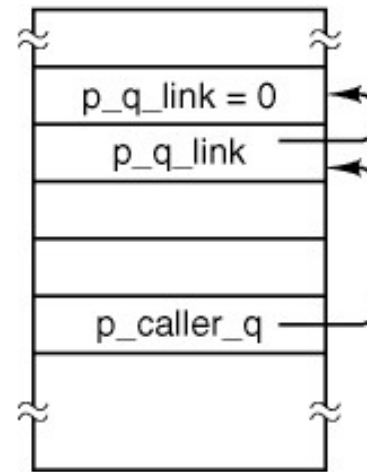
- Quando un processo effettua una SEND
 - Il kernel verifica se il destinatario è in attesa di un messaggio dal mittente (o ANY); nel PCB del destinatario sarà allora necessario disporre di
 - Un campo che indica che il destinatario è in attesa su una RECEIVE (`p_rts_flags`)
 - Un campo che indica il numero del processo da cui si aspetta un messaggio (`p_getfrom`)
 - In caso di risposta affermativa, il messaggio viene copiato da un buffer del sender ad un buffer del receiver, e il processo receiver è posto in ready

send (int dest, message *m_ptr)

- Altrimenti il sender viene bloccato, e deve essere :
 - Memorizzato nel sender il numero del processo destinatario (`p_sendto`)
 - tenuta traccia dei processi bloccati su send al receiver (in caso di errore del receiver devono poter essere sbloccati)



(a)



(b)

```
receive (int src, message *m_ptr)
```

- Quando un processo P effettua una receive:
 - Verifica se il processo che deve inviare il messaggio (src) è in attesa di inviare un messaggio a P, per fare queste verifiche è necessario disporre nel PCB di src di:
 - un campo che indica se src è in attesa di inviare un messaggio (`p_rts_flags`)
 - un campo che identifica il destinatario del messaggio (`p_sendto`)
 - In caso affermativo, il messaggio è copiato dal buffer del Sender a quello di P
 - Se nessun processo è in attesa di inviare messaggi a P, P viene bloccato sino a quando un messaggio arriva, in questo caso si dovrà memorizzare nel PCB di P l'identificativo di src

PCB

```
05516 struct proc {
05517 struct stackframe_s p_reg; /* process' registers saved in stack frame */
05518 reg_t p_ldt_sel; /* selector in gdt with ldt base and limit */
05519 struct segdesc_s p_ldt[2+NR_REMOTE_SEGS]; /* CS, DS and remote segments
*/
05520
05521 proc_nr_t p_nr; /* number of this process (for fast access) */
05522 struct priv *p_priv; /* system privileges structure */
05523 char p_rts_flags; /* SENDING, RECEIVING, etc. */
05524
05525 char p_priority; /* current scheduling priority */
05526 char p_max_priority; /* maximum scheduling priority */
05527 char p_ticks_left; /* number of scheduling ticks left */
05528 char p_quantum_size; /* quantum size in ticks */
05529
05530 struct mem_map p_memmap[NR_LOCAL_SEGS]; /* memory map (T, D, S) */
05531
05532 clock_t p_user_time; /* user time in ticks */
05533 clock_t p_sys_time; /* sys time in ticks */
```

PCB

```
05535 struct proc *p_nextready; /* pointer to next ready process */

05536 struct proc *p_caller_q; /* head of list of procs wishing to send */
05537 struct proc *p_q_link; /* link to next proc wishing to send */
05538 message *p_messbuf; /* pointer to passed message buffer */
05539 proc_nr_t p_getfrom; /* from whom does process want to receive? */
05540 proc_nr_t p_sendto; /* to whom does process want to send? */

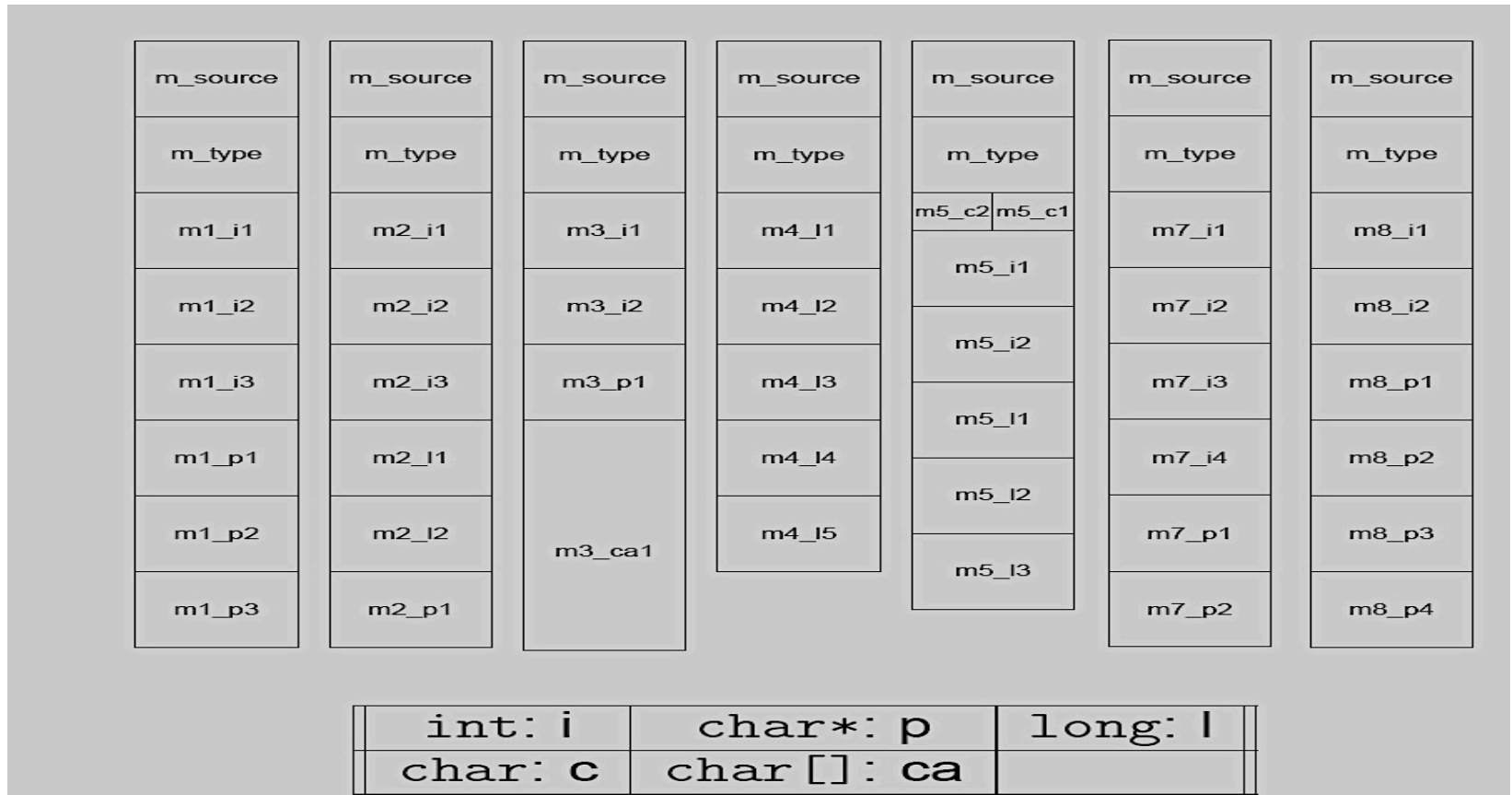
05542 sigset_t p_pending; /* bit map for pending kernel signals */
05543
05544 char p_name[P_NAME_LEN]; /* name of the process, including \0 */
05545 };
```

p_rts_flags

```
00062 /* Bits for the runtime flags. A process is runnable iff p_rts_flags == 0. */

00063 #define SLOT_FREE 0x01 /* process slot is free */
00064 #define NO_MAP 0x02 /* keeps unmapped forked child from running */
00065 #define SENDING 0x04 /* process blocked trying to send */
00066 #define RECEIVING 0x08 /* process blocked trying to receive */
00067 #define SIGNALLED 0x10 /* set when new kernel signal arrives */
00068 #define SIG_PENDING 0x20 /* unready while signal being processed */
00069 #define P_STOP 0x40 /* set when process is being traced */
00070 #define NO_PRIV 0x80 /* keep forked system process from running */
00071 #define NO_PRIORITY 0x100 /* process has been stopped */
00072 #define NO_ENDPOINT 0x200 /* process cannot send or receive messages */
00073
00074 /* Misc flags */
00075 #define REPLY_PENDING 0x01 /* reply to IPC_REQUEST is pending */
00076 #define MF_VM 0x08 /* process uses VM */
```

I messaggi in Minix



Dichiarazione messaggi

```
/* =====*
*   Types relating to messages.           *
*=====*
#define M1 1
#define M3 3
#define M4 4
#define M3_STRING 14

typedef struct {int m1i1, m1i2, m1i3; char *m1p1, *m1p2, *m1p3;}mess_1;
typedef struct {int m2i1, m2i2, m2i3; long m2l1, m2l2; char *m2p1;} mess_2;
typedef struct {int m3i1, m3i2; char *m3p1; char m3ca1[M3_STRING];} mess_3;
typedef struct {long m4l1, m4l2, m4l3, m4l4, m4l5;} mess_4;
typedef struct {short m5c1, m5c2; int m5i1, m5i2; long m5l1, m5l2, m5l3;}
mess_5;
typedef struct {int m7i1, m7i2, m7i3, m7i4; char *m7p1, *m7p2;} mess_7;
typedef struct {int m8i1, m8i2; char *m8p1, *m8p2, *m8p3, *m8p4;} mess_8;
```

Dichiarazione

```
03020 typedef struct {
03021     int m_source;           /* who sent the message */
03022     int m_type;            /* what kind of message is it */
03023     union {
03024         mess_1 m_m1;
03025         mess_2 m_m2;
03026         mess_3 m_m3;
03027         mess_4 m_m4;
03028         mess_5 m_m5;
03029         mess_7 m_m7;
03030         mess_8 m_m8;
03031     } m_u;
03032 } message;
03033
03034 /* The following defines provide names for useful members. */
03035 #define m1_i1    m_u.m_m1.m1i1
03036 #define m1_i2    m_u.m_m1.m1i2
03037 #define m1_i3    m_u.m_m1.m1i3
03038 #define m1_p1    m_u.m_m1.m1p1
03039 #define m1_p2    m_u.m_m1.m1p2
03040 #define m1_p3    m_u.m_m1.m1p3
03041 .....
03049 #define m3_i1    m_u.m_m3.m3i1
03050 #define m3_i2    m_u.m_m3.m3i2
03051 #define m3_p1    m_u.m_m3.m3p1
03052 #define m3_ca1   m_u.m_m3.m3ca1
```

`m1_i3` ⇒ msg type #1, int-field #3

`m3_ca1` ⇒ msg type #3, character-array-field #1

```
message msg;
msg.m_u.m_m1.m1i3 = 3;
msg.m1_i3 = 3;
```

Priv

```
struct priv {
00029   proc_nr_t s_proc_nr;           /* number of associated process */
00030   sys_id_t s_id;                 /* index of this system structure */
00031   short s_flags;                 /* PREEMTIBLE, BILLABLE, etc. */
00032
00033   short s_trap_mask;             /* allowed system call traps */
00034   sys_map_t s_ipc_from;          /* allowed callers to receive from */
00035   sys_map_t s_ipc_to;           /* allowed destination processes */
00036   long s_call_mask;             /* allowed kernel calls */
00037
00038   sys_map_t s_notify_pending;    /* bit map with pending notifications */
00039   irq_id_t s_int_pending;        /* pending hardware interrupts */
00040   sigset_t s_sig_pending;        /* pending signals */
00041
00042   timer_t s_alarm_timer;         /* synchronous alarm timer */
00043   struct far_mem s_farmem[NR_REMOTE_SEGS]; /* remote memory map */
00044   reg_t *s_stack_guard;         /* stack guard word for kernel tasks */
}
```

Alcune Costanti

src/lib/i386/rts/_sendrec.s

```
0065002 .define __send, __receive, __sendrec
0065003
0065004 ! See ../h/com.h for C definitions
0065005 SEND = 1
0065006 RECEIVE = 2
0065007 BOTH = 3
0065008 SYSVEC = 33
0065009
0065010 SRCDEST = 8
0065011 MESSAGE = 12
0065012
```

__send

```
0065016 ! _send(), _receive(), all save ebp, but destroy eax and
        ecx.
0065017 .define __send, __receive, __sendrec
0065018 .sect .text
0065019 __send:
0065020     push     ebp
0065021     mov      ebp, esp
0065022     push     ebx
0065023     mov      eax, SRCDEST(ebp)      ! eax = dest-src
0065024     mov      ebx, MESSAGE(ebp)     ! ebx = message pointer
0065025     mov      ecx, SEND              ! _send(dest, ptr)
0065026     int      SYSVEC                ! trap to the kernel
0065027     pop      ebx
0065028     pop      ebp
0065029     ret
```

__receive

```
0065031  __receive:
0065032  push    ebp
0065033  mov     ebp, esp
0065034  push    ebx
0065035  mov     eax, SRCDEST(ebp)    ! eax = dest-src
0065036  mov     ebx, MESSAGE(ebp)    ! ebx = message
    pointer
0065037  mov     ecx, RECEIVE         ! __receive(src, ptr)
0065038  int     SYSVEC               ! trap to the kernel
0065039  pop     ebx
0065040  pop     ebp
0065041  ret
```

Contenuto IDT

```

    { hwint00, VECTOR( 0), INTR_PRIVILEGE },
00115  { hwint01, VECTOR( 1), INTR_PRIVILEGE },
00116  { hwint02, VECTOR( 2), INTR_PRIVILEGE },
00117  { hwint03, VECTOR( 3), INTR_PRIVILEGE },
00118  { hwint04, VECTOR( 4), INTR_PRIVILEGE },
00119  { hwint05, VECTOR( 5), INTR_PRIVILEGE },
00120  { hwint06, VECTOR( 6), INTR_PRIVILEGE },
00121  { hwint07, VECTOR( 7), INTR_PRIVILEGE },
00122  { hwint08, VECTOR( 8), INTR_PRIVILEGE },
00123  { hwint09, VECTOR( 9), INTR_PRIVILEGE },
00124  { hwint10, VECTOR(10), INTR_PRIVILEGE },
00125  { hwint11, VECTOR(11), INTR_PRIVILEGE },
00126  { hwint12, VECTOR(12), INTR_PRIVILEGE },
00127  { hwint13, VECTOR(13), INTR_PRIVILEGE },
00128  { hwint14, VECTOR(14), INTR_PRIVILEGE },
00129  { hwint15, VECTOR(15), INTR_PRIVILEGE },
00130 #if _WORD_SIZE == 2
00131     { p_s_call, SYS_VECTOR, USER_PRIVILEGE }, /* 286 system call
*/
00132 #else
00133     { s_call, SYS386_VECTOR, USER_PRIVILEGE }, /* 386 system call
*/
00134 #endif
00135     { level0_call, LEVEL0_VECTOR, TASK_PRIVILEGE },
00136     };
```

```

_s_call:
351  _p_s_call:
352      cld                ! set direction flag to a known value
353      sub                esp, 6*4    ! skip RETADR, eax, ecx, edx, ebx, est
354      push               ebp        ! stack already points into proc table
355      push               esi
356      push               edi
357      o16 push          ds
358      o16 push          es
359      o16 push          fs
360      o16 push          gs
361      mov                dx, ss
362      mov                ds, dx
363      mov                es, dx
364      incb              (_k_reenter)
365      mov                esi, esp    ! assumes P_STACKBASE == 0
366      mov                esp, k_stktop
367      xor                ebp, ebp    ! for stacktrace
368                                ! end of inline save
369                                ! now set up parameters for sys_call()
370      push               ebx        ! pointer to user message
371      push               eax        ! src/dest
372      push               ecx        ! SEND/RECEIVE/BOTH
373      call              _sys_call   ! sys_call(function, src_dest, m_ptr)
374                                ! caller is now explicitly in proc_ptr
375      mov                AXREG(esi), eax ! sys_call MUST PRESERVE si

```

```

!*=====
316 !*                save                *
317 !*=====
318 ! Save for protected mode.
319 ! This is much simpler than for 8086 mode, because the stack already points
320 ! into the process table, or has already been switched to the kernel stack.
321
322         .align 16
323 save:
324         cld                ! set direction flag to a known value
325         pushad             ! save "general" registers
326         o16 push ds        ! save ds
327         o16 push es        ! save es
328         o16 push fs        ! save fs
329         o16 push gs        ! save gs
330         mov dx, ss         ! ss is kernel data segment
331         mov ds, dx         ! load rest of kernel segments
332         mov es, dx         ! kernel does not use fs, gs
333         mov eax, esp       ! prepare to return
334         incb (_k_reenter)  ! from -1 if not reentering
335         jnz set_restart1   ! stack is already kernel stack
336         mov esp, k_stktop
337         push _restart      ! build return address for int handler
338         xor ebp, ebp       ! for stacktrace
339         jmp RETADR-P_STACKBASE(eax)
340
341         .align 4
342 set_restart1:
343         push restart1
344         jmp RETADR-P_STACKBASE(eax)

```

send/receive

- A seguito dell'interrupt software generato sia da send che da receive il sistema chiama come risposta all'interrupt la routine `_s_call`
- la routine `_s_call` a sua volta chiama la routine `sys_call` che a sua volta chiama le routine `mini_send` o `mini_receive`
- al termine dell'esecuzione di queste procedure il controllo torna all'istruzione successiva alla chiamata di `_sys_call` e conseguentemente alla procedura `restart()` il cui codice è adiacente a quello di `_s_call`

File di riferimento: proc.c

```
00091 /*=====*
00092 * sys_call *
00093 *=====*/
00094 PUBLIC int sys_call(call_nr, src_dst_e, m_ptr, bit_map)
00095 int call_nr; /* system call number and flags */
00096 int src_dst_e; /* src to receive from or dst to send to */
00097 message *m_ptr; /* pointer to message in the caller's space */
00098 long bit_map; /* notification event set or flags */
00099 {
00100 /* System calls are done by trapping to the kernel with an INT instruction.
00101 * The trap is caught and sys_call() is called to send or receive a message
00102 * (or both). The caller is always given by 'proc_ptr'.
00103 */
```

sys_call

- La procedura `sys_call` effettua una serie di verifiche preliminari quali:
 - Verificare che gli identificativi dei processi sender/receiver passati come parametri sono corretti
 - Verificare che il processo chiamante abbia i necessari privilegi per eseguire la system call richiesta
 - i processi utenti possono solo richiedere servizi al kernel attraverso SENDREC, al fine di evitare che il kernel resti bloccato da un processo utente che non esegue la receive a seguito di una send
 - Verificare che il sender abbia i permessi necessari per dialogare con il receiver
 - Verificare che il puntatore al messaggio sia valido

```

switch(function) {
00198 case SENDREC:
001 /* A flag is set so that notifications cannot interrupt SENDREC. */
00200 caller_ptr->p_misc_flags |= REPLY_PENDING;
00201 /* fall through */
00202 case SEND:
00203 result = mini_send(caller_ptr, src_dst_e, m_ptr, flags);
00204 if (function == SEND || result != OK) {
00205 break; /* done, or SEND failed */
00206 } /* fall through for SENDREC */
00207 case RECEIVE:
00208 if (function == RECEIVE)
00209 caller_ptr->p_misc_flags &= ~REPLY_PENDING;
00210 result = mini_receive(caller_ptr, src_dst_e, m_ptr, flags);
00211 break;
00212 case NOTIFY:
00213 result = mini_notify(caller_ptr, src_dst);
00214 break;
00215 case ECHO:
00216 CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, caller_ptr, m_ptr);
00217 result = OK;
00218 break;
00219 default:
00220 result = EBADCALL; /* illegal system call */

```

mini_send

- Sia S il processo che effettua la send, le operazioni svolte per effettuare questa istruzione sono:
 - Verifica che processo dst sia in attesa di un messaggio da parte di S ed in questo caso consegna il messaggio e sblocca dst
 - altrimenti, S viene messo in attesa su una lista indirizzata dal puntatore iniziale `p_caller_q` del PCB di dest
 - Nel campo `p_sendto` del PCB di S viene inserito il valore di dst

Enqueue e dequeue

```
PRIVATE void enqueue(rp)
```

```
00503 register struct proc *rp; /* this process is now runnable */
00504 {
00505 /* Add 'rp' to one of the queues of runnable processes. This
      function is responsible for inserting a process into one of the
      scheduling queues. The mechanism is implemented here. The actual
      scheduling policy is defined in sched() and pick_proc(). */
```

```
PRIVATE void dequeue(rp)
```

```
00549 register struct proc *rp;
/* this process is no longer runnable */
00550 {
00551 /* A process must be removed from the scheduling queues, for
      example, because it has blocked. If the currently active process
      is removed, a new process is picked to run by calling pick_proc
      (). */
```

mini_send

```
07588*/=====*
07589 * mini_send *
07590 *=====*/
07591 PRIVATE int mini_send(caller_ptr, dst, m_ptr, flags)
07592 register struct proc *caller_ptr; /* who is trying to
    send a message? */
07593 int dst; /* to whom is message being sent? */
07594 message *m_ptr; /* pointer to message buffer */
07595 unsigned flags; /* system call flags */

07601 register struct proc *dst_ptr = proc_addr(dst);
07602 register struct proc **xpp;
07603 register struct proc *xp;
```

mini_send

```
/* Send a message from 'caller_ptr' to 'dst'. If 'dst' is blocked waiting
00207 * for this message, copy the message to it and unblock 'dst'. If 'dst' is
00208 * not waiting at all, or is waiting for another source, queue 'caller_ptr'.
00209 */
00210 register struct proc *dst_ptr = proc_addr(dst);
00211 register struct proc **xpp;
00212 register struct proc *xp;
00213
00214 /* Check for deadlock by 'caller_ptr' and 'dst' sending to each other. */
00215 xp = dst_ptr;
00216 while (xp->p_rts_flags & SENDING) { /* check while sending */
00217     xp = proc_addr(xp->p_sendto); /* get xp's destination */
00218     if (xp == caller_ptr) return(ELOCKED); /* deadlock if cyclic */
00219 }
```

Destination è pronta

```
/* Check if 'dst' is blocked waiting for this message. The destination's
 * SENDING flag may be set when its SENDREC call blocked while sending.
 */
if ( (dst_ptr->p_rts_flags & (RECEIVING | SENDING)) == RECEIVING &&
    (dst_ptr->p_getfrom == ANY || dst_ptr->p_getfrom == caller_ptr->p_nr))
{
    /* Destination is indeed waiting for this message.*/
    CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, dst_ptr,
            dst_ptr->p_messbuf);

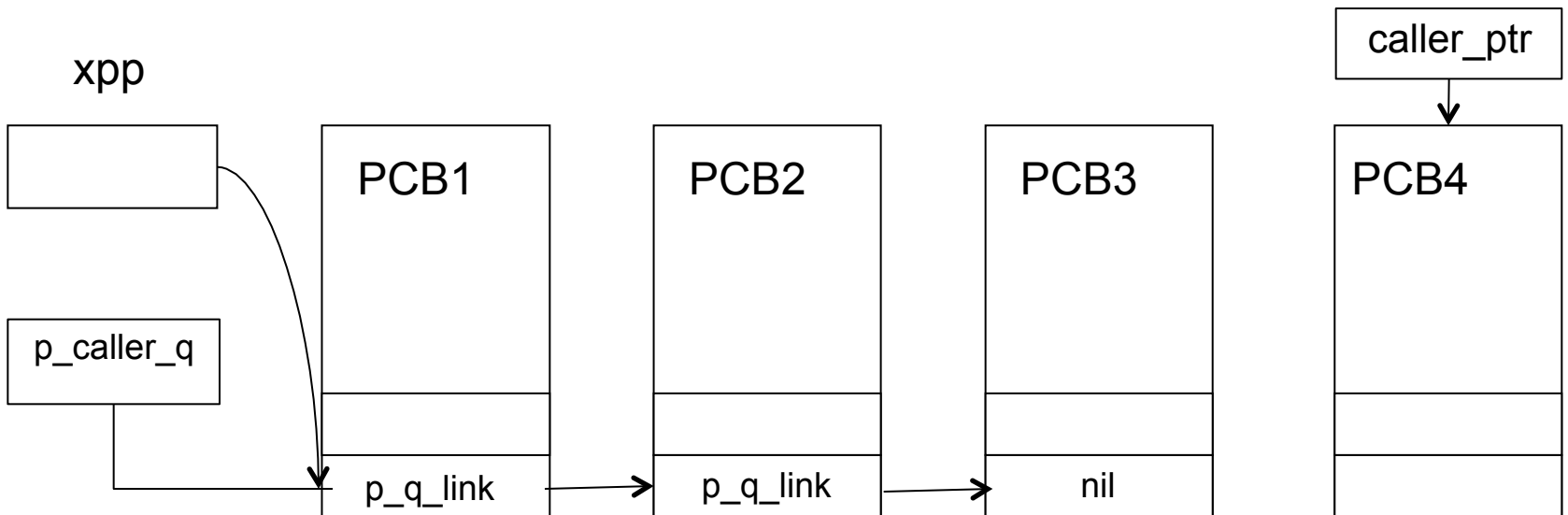
    /* SE IL PROCESSO ERA SOSPESO SOLO IN RECEIVING RISVEGLIALO */
    if ((dst_ptr->p_rts_flags &= ~RECEIVING) == 0) enqueue(dst_ptr);
}
```

Destination non è pronta

```
} else if ( ! (flags & NON_BLOCKING)) {
00231     /* Destination is not waiting.  Block and dequeue caller. */
00232     caller_ptr->p_messbuf = m_ptr;
00233     if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
00234     caller_ptr->p_rts_flags |= SENDING;
00235     caller_ptr->p_sendto = dst;
00236
00237     /* Process is now blocked.  Put in on the destination's queue. */
00238     xpp = &dst_ptr->p_caller_q;           /* find end of list */
00239     while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;
00240     *xpp = caller_ptr;                   /* add caller to end */
00241     caller_ptr->p_q_link = NIL_PROC;     /* mark new end of list */
00242 } else {
00243     return(ENOTREADY);
00244 }
00245 return(OK);
00246 }
```

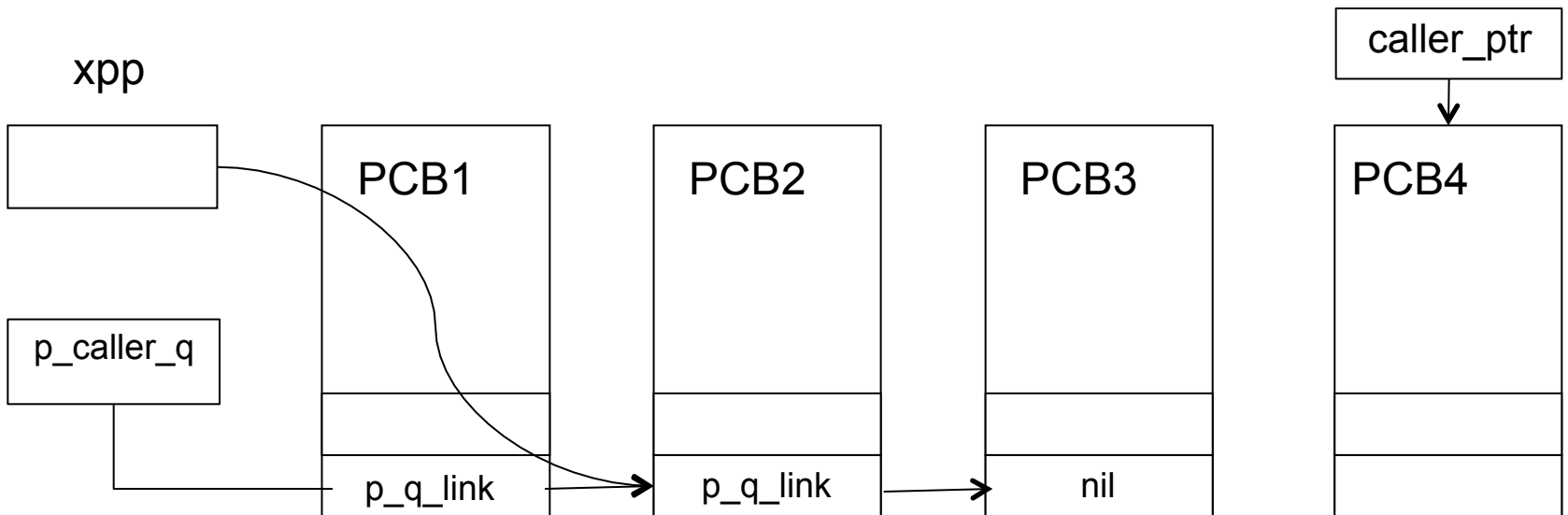
Puntatori a puntatori

```
xpp = &dst_ptr->p_caller_q;    /* find end of list */  
while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;  
*xpp = caller_ptr;           /* add caller to end */  
caller_ptr->p_q_link = NIL_PROC; /* new end of list */
```



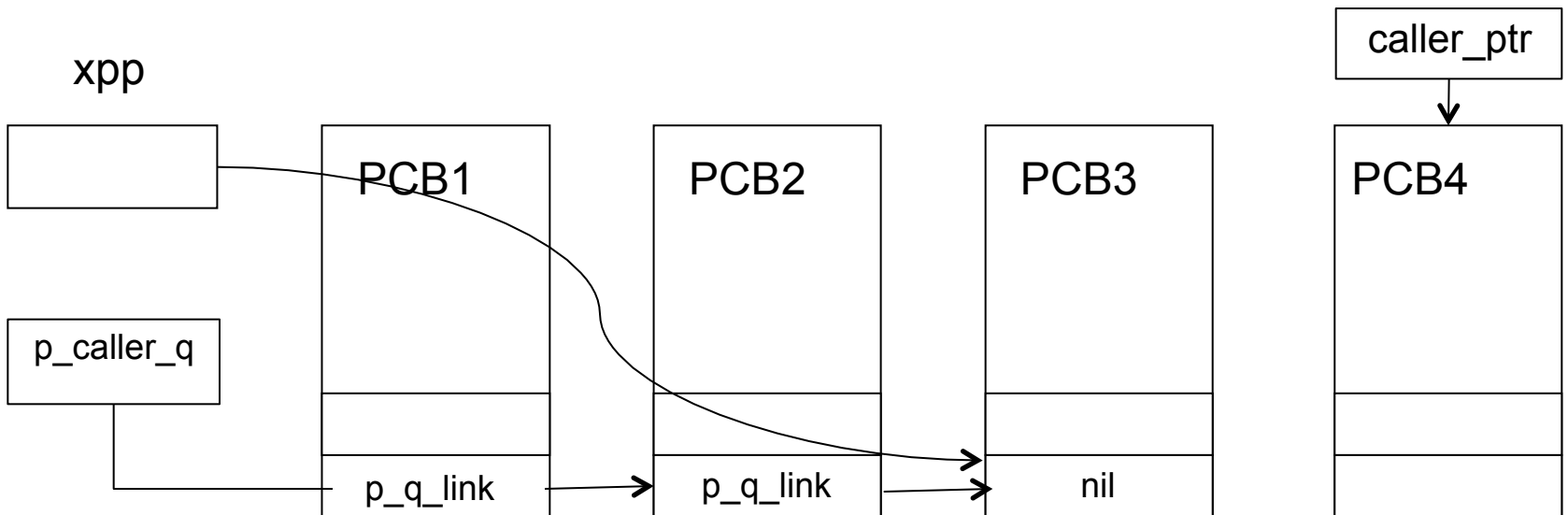
Puntatori a puntatori

```
xpp = &dst_ptr->p_caller_q;    /* find end of list */  
while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;  
*xpp = caller_ptr;          /* add caller to end */  
caller_ptr->p_q_link = NIL_PROC; /* new end of list */
```



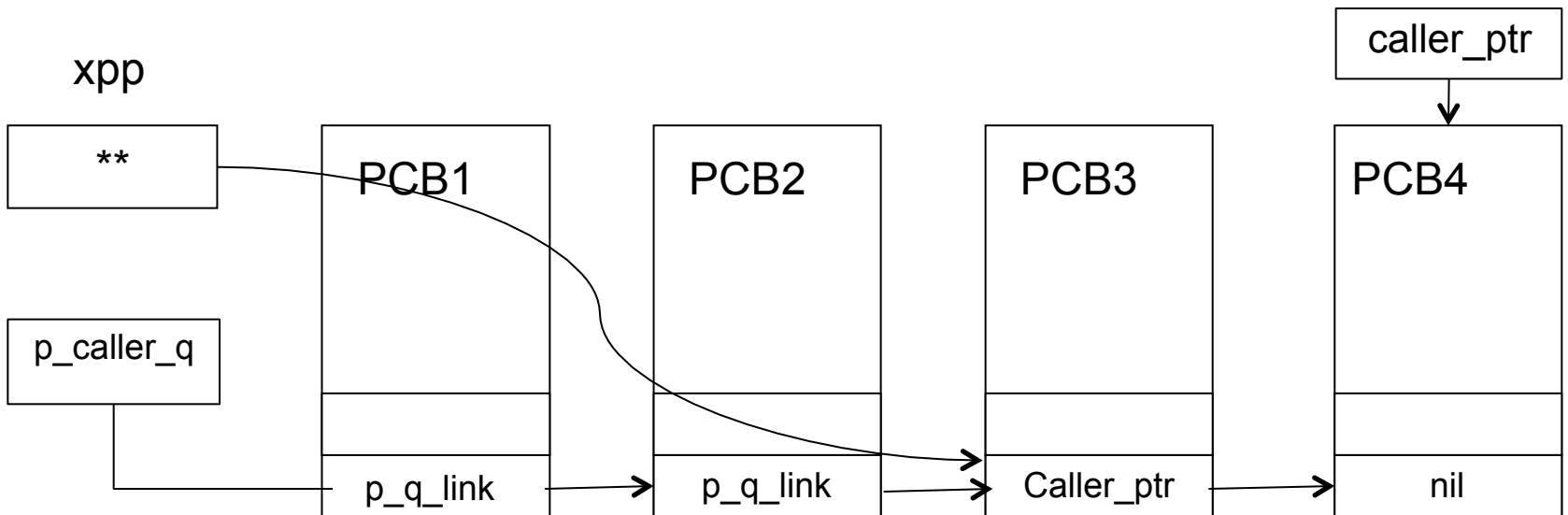
Puntatori a puntatori

```
xpp = &dst_ptr->p_caller_q;    /* find end of list */  
while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;  
*xpp = caller_ptr;           /* add caller to end */  
caller_ptr->p_q_link = NIL_PROC; /* new end of list */
```



Puntatori a puntatori

```
xpp = &dst_ptr->p_caller_q;    /* find end of list */  
while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;  
*xpp = caller_ptr;           /* add caller to end */  
caller_ptr->p_q_link = NIL_PROC; /* new end of list */
```



mini_receive

- Sia R il processo che effettua la `receive(src, msg)`, le operazioni svolte per effettuare questa istruzione sono:
 - Verifica se ci sono notifiche da consegnare
 - Verifica se processo src è in attesa di consegnare il messaggio, in questo caso consegna il messaggio a R e sblocca src
 - Altrimenti: R viene messo in attesa
 - Nel campo `p_getfrom` del PCB di R viene inserito il valore di src

mini_receive

```
PRIVATE int mini_receive(caller_ptr, src, m_ptr, flags)
00252 register struct proc *caller_ptr;          /* process trying to get message */
00253 int src;                                   /* which message source is wanted */
00254 message *m_ptr;                            /* pointer to message buffer */
00255 unsigned flags;                           /* system call flags */
00256 {
00257 /* A process or task wants to get a message.  If a message is already queued,
00258  * acquire it and deblock the sender.  If no message from the desired source
00259  * is available block the caller, unless the flags don't allow blocking.
00260  */
```

Cosegna notifiche

```
00358  /* Check to see if a message from desired source is already available.
00359  * The caller's SENDING flag may be set if SENDREC couldn't send. If it is
00360  * set, the process should be blocked.
      */

00362  if (!(caller_ptr->p_rts_flags & SENDING)) {
00363
364  /* Check if there are pending notifications, except for SENDREC. */

00365      if (! (caller_ptr->p_misc_flags & REPLY_PENDING)) {
```

Verifica se src è in attesa

```
00297  /* Check caller queue. Use pointer pointers to keep code simple. */
00298  xpp = &caller_ptr->p_caller_q;
00299  while (*xpp != NIL_PROC) {
00300      if (src == ANY || src == proc_nr(*xpp)) {
00301          /* Found acceptable message. Copy it and update status. */
00302          CopyMess((*xpp)->p_nr, *xpp, (*xpp)->p_messbuf, caller_ptr, m_ptr);
00303          if (((*xpp)->p_rts_flags & ~SENDING) == 0) enqueue(*xpp);
00304          *xpp = (*xpp)->p_q_link;          /* remove from queue */
00305          return(OK);                      /* report success */
00306      }
00307      xpp = &(*xpp)->p_q_link;          /* proceed to next */
00308  }
```

src non è pronto/caller_tr è bloccato

```
/* No suitable message is available or the caller couldn't send in
SENDREC.
Block the process trying to receive, unless the flags tell otherwise.
*/
00314     if ( ! (flags & NON_BLOCKING)) {
00315         caller_ptr->p_getfrom = src;
00316         caller_ptr->p_messbuf = m_ptr;
00317         if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
00318         caller_ptr->p_rts_flags |= RECEIVING;
00319         return(OK);
00320     } else {
00321         return(ENOTREADY);
00322     }
```

NOTIFY

- Primitiva di comunicazione non bloccante: il sender prosegue la sua esecuzione anche se il mittente non è in attesa del messaggio
- La notify non è comunque persa
- La prima volta che il destinatario effettua una receive, le notifiche sono consegnate prima dei messaggi ordinari
- Viene usata esclusivamente all' interno del kernel

mini-notify

```
PRIVATE int mini_notify(caller_ptr, dst)
00329 register struct proc *caller_ptr; /* sender of the notification */
00330 int dst; /* which process to notify */
00331 {
00332     register struct proc *dst_ptr = proc_addr(dst);
00333     int src_id; /* source id for late delivery */
00334     message m; /* the notification message */
00335
00336 /* Check to see if target is blocked waiting for this message. A process
00337  * can be both sending and receiving during a SENDREC system call.
00338  */
```

Destination è pronta

```
/* Destination is indeed waiting for a message. Assemble a notification
00344 * message and deliver it. Copy from pseudo-source HARDWARE, since the
00345 * message is in the kernel's address space.
00346 */
00347 BuildMess(&m, proc_nr(caller_ptr), dst_ptr);
00348 CopyMess(proc_nr(caller_ptr), proc_addr(HARDWARE), &m,
00349         dst_ptr, dst_ptr->p_messbuf);
00350 dst_ptr->p_rts_flags &= ~RECEIVING; /* deblock destination */
00351 if (dst_ptr->p_rts_flags == 0) enqueue(dst_ptr);
00352 return(OK);
00353 }
```

Destination non è pronta

```
/* Destination is not ready to receive the notification. Add it to the
00356 * bit map with pending notifications. Note the indirectness: the system id
00357 * instead of the process number is used in the pending bit map.
00358 */
00359 src_id = priv(caller_ptr)->s_id;
00360 set_sys_bit(priv(dst_ptr)->s_notify_pending, src_id);
00361 return(OK);
```

lock_notify

```
PUBLIC int lock_notify(src, dst)
00368 int src;                /* sender of the notification */
00369 int dst;                /* who is to be notified */
00370 {
00371 /* Safe gateway to mini_notify() for tasks and interrupt handlers. The sender
00372 * is explicitly given to prevent confusion where the call comes from. MINIX
00373 * kernel is not reentrant, which means to interrupts are disabled after
00374 * the first kernel entry (hardware interrupt, trap, or exception). Locking
00375 * is done by temporarily disabling interrupts.
00376 */
00377     int result;
00378
00379     /* Exception or interrupt occurred, thus already locked. */
00380     if (k_reenter >= 0) {
00381         result = mini_notify(proc_addr(src), dst);
00382     }
```

lock

```
    /* Call from task level, locking is required. */
00385     else {
00386         lock(0, "notify");
00387         result = mini_notify(proc_addr(src), dst);
00388         unlock(0);
00389     }
00390     return(result);
00391 }
00392
```

IPC TRA PROCESSI UTENTE/ SISTEMA

Signal

- Meccanismo per segnalare eventi sincroni e/o asincroni tra processi utente e di sistema
- Esiste un insieme predefinito di segnali e per ciascuno di essi sono definite le azioni di default che devono essere effettuate quando un processo riceve questi tipi di segnale

Esempi

Signal	Description	Generated by
SIGHUP	Hangup	KILL system call
SIGINT	Interrupt	TTY
SIGQUIT	Quit	TTY
SIGILL	Illegal instruction	Kernel (*)
SIGTRAP	Trace trap	Kernel (M)
SIGABRT	Abnormal termination	TTY
SIGFPE	Floating point exception	Kernel (*)
SIGKILL	Kill (cannot be caught or ignored)	KILL system call
SIGUSR1	User-defined signal # 1	Not supported
SIGSEGV	Segmentation violation	Kernel (*)
SIGUSR2	User defined signal # 2	Not supported
SIGPIPE	Write on a pipe with no one to read it	FS

Esempio

Signal	Description	Generated by
SIGALRM	Alarm clock, timeout	PM
SIGTERM	Software termination signal from kill	KILL system call
SIGCHLD	Child process terminated or stopped	PM
SIGCONT	Continue if stopped	Not supported
SIGSTOP	Stop signal	Not supported
SIGTSTP	Interactive stop signal	Not supported

- Signal definite da POSIX e MINIX 3.
- Le Signal indicate con (*) dipendono dall' hardware
- Signal con (M) non sono POSIX, sono però definite in MINIX per questioni di back compatibilità

Signal

- Un processo può però specificare:
 - Azioni alternative da eseguire quando riceve un segnale di un certo tipo e quindi specificare un signal handler per la gestione del segnale
 - Dichiarare di voler ignorare un certo tipo di segnale
- Alle suddette regole fa eccezione il segnale SIGKILL che non può essere ignorato
- Tutte le procedure di gestione di un segnale terminano con l'esecuzione della syscall sigreturn

Segnali e processi utente

- Per specificare le proprie opzioni su una signal un processo ha a disposizione due system call:
 - Sigaction: usata per specificare i segnali che si intendono ignorare, i signal handler specifici che si intendono adottare, o ripristinare le azioni di default su un segnale
 - Sigprocmask: può essere usata per bloccare un segnale per un certo periodo di tempo dopo il quale sarà il processo stesso a riabilitarlo con una sigaction

```
struct sigaction {
    __sighandler_t sa_handler; /* SIG_DFL, SIG_IGN, SIG_MESS,
                               or pointer to function */
    sigset_t sa_mask;         /* signals to be blocked during handler */
    int sa_flags;             /* special flags */
}
```

