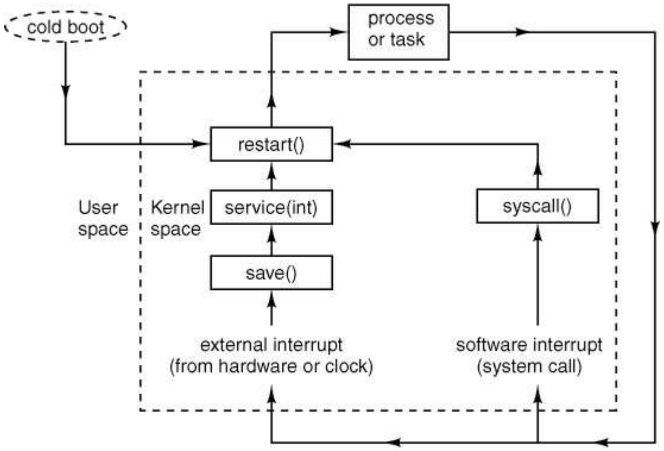
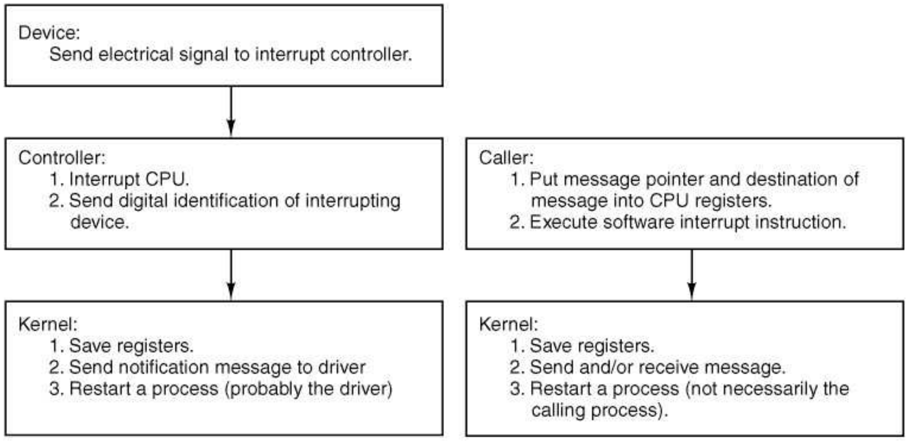




Device:
Send electrical signal to interrupt controller.

Controller:
1. Interrupt CPU.
2. Send digital identification of interrupting device.

Kernel:
1. Save registers.
2. Send notification message to driver
3. Restart a process (probably the driver)

Caller:
1. Put message pointer and destination of message into CPU registers.
2. Execute software interrupt instruction.

Kernel:
1. Save registers.
2. Send and/or receive message.
3. Restart a process (not necessarily the calling process).

(a)

(b)

La chiamata di sistema times(2) I

```
NAME
     times - get process times

SYNOPSIS
     #include <sys/types.h>
     #include <sys/times.h>
     #include <time.h>

     int times(struct tms *buffer)

DESCRIPTION

     Times returns time-accounting information  for  the  current
     process  and  for  the  terminated  child  processes  of the
     current process.  All times are in 1/CLOCKS_PER_SEC seconds.
```

La chiamata di sistema times(2) II

```
This is the structure returned by times:

     struct tms {
         clock_t tms_utime;  /* user time for this process */
         clock_t tms_stime;  /* system time for this process */
         clock_t tms_cutime; /* children's user time */
         clock_t tms_cstime; /* children's system time */
     };

The user time is the number of clock ticks used by a process
on  its  own computations.  The system time is the number of
clock ticks spent inside the kernel on behalf of a  process.
This  does not include time spent waiting for I/O to happen,
only actual CPU instruction times.

The children times are the sum  of  the  children's  process
times and their children's times.
```

La chiamata di sistema times(2) III

```
RETURN
     Times returns 0 on success, otherwise -1 with the error code
     stored into the global variable errno.

ERRORS

     The following error code may be set in errno:

     [EFAULT]       The address specified by the buffer parameter
                    is not in a valid part of the process address
                    space.
```

Un programma utente

```
#include <sys/types.h>
#include <sys/times.h>
#include <time.h>

int main() {
  struct tms buffer;
  times(&buffer);
  printf(%d\n, buffer.tms_utime)
}
```

Una syscall *sembra* un'ordinaria chiamata di lib...

include/times.h

```
#ifndef _CLOCK_T
#define _CLOCK_T
typedef long clock_t;           /* unit for system accounting */
#endif

struct tms {
  clock_t tms_utime;
  clock_t tms_stime;
  clock_t tms_cutime;
  clock_t tms_cstime;
};

_PROTOTYPE( clock_t times, (struct tms *_buffer) );
```

...ma in realtà coinvolge il kernel

lib/posix/_times.c

```
#include <sys/times.h>
#include <time.h>

PUBLIC clock_t times(buf)
struct tms *buf;
{
  message m;

  if (_syscall(MM, TIMES, &m) < 0) return( (clock_t) -1);
  buf->tms_utime = m.m4_l1;
  buf->tms_stime = m.m4_l2;
  buf->tms_cutime = m.m4_l3;
  buf->tms_cstime = m.m4_l4;
  return(m.m4_l5);
}
```

La chiamata di sistema si espande...

lib/other/syscall.c

```
PUBLIC int _syscall(who, syscallnr, msgptr)
int who;
int syscallnr;
register message *msgptr;
{
  int status;

  msgptr->m_type = syscallnr;
  status = _sendrec(who, msgptr);
  if (status != 0) {
    /* 'sendrec' itself failed. */
    msgptr->m_type = status;
  }
  if (msgptr->m_type < 0) {
    errno = -msgptr->m_type;
    return(-1);
  }
  return(msgptr->m_type);
}
```

La sendrec è implementata in assembler

```
__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
```

Il punto di ingresso è _s_call I

kernel/mpx386.c

```
_s_call:
        cld                     ! set direction flag to a known value
        ... save registers on stack ...
                                ! now set up parameters for sys_call()
        push    ebx             ! pointer to user message
        push    eax             ! src/dest
        push    ecx             ! SEND/RECEIVE/BOTH
        call    _sys_call       ! sys_call(function, src_dest, m_ptr)
                                ! caller is now explicitly in proc_ptr
        mov     AXREG(esi), eax ! sys_call MUST PRESERVE si

! Fall into code to restart proc/task running.

_restart:

! Restart the current process or the next process if it is set.
```

Il punto di ingresso è _s_call II

```
_restart:
! Restart the current process or the next process if it is set.
        cmp     (_next_ptr), 0          ! see if another process is schedulec
        jz      0f
        mov     eax, (_next_ptr)
        mov     (_proc_ptr), eax        ! schedule new process
        mov     (_next_ptr), 0
0:      mov     esp, (_proc_ptr)        ! will assume P_STACKBASE == 0
        lldt    P_LDT_SEL(esp)          ! enable process' segment descriptors
        lea     eax, P_STACKTOP(esp)    ! arrange for next interrupt
        mov     (_tss+TSS3_S_SP0), eax  ! to save state in process table
restart1:
        decb    (_k_reenter)
    o16 pop     gs
    o16 pop     fs
    o16 pop     es
    o16 pop     ds
        popad
        add     esp, 4          ! skip return adr
        iretd                   ! continue process
```

Il punto di ingresso è _s_call III

_syscall in kernel/proc.c I

```
PUBLIC int sys_call(call_nr, src_dst, m_ptr)
int call_nr;                    /* system call number and flags */
int src_dst;                    /* src to receive from or dst to send to */
message *m_ptr;                 /* pointer to message in the caller's space *
{
/* 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'.
 */
  register struct proc *caller_ptr = proc_ptr;  /* get pointer to caller */
  int function = call_nr & SYSCALL_FUNC;        /* get system call function *
  unsigned flags = call_nr & SYSCALL_FLAGS;     /* get flags */
  int mask_entry;                               /* bit to check in send mask
  int group_size;                               /* used for deadlock check */
  int result;                                   /* the system call's result *
  vir_clicks vlo, vhi;          /* virtual clicks containing message to send
```

_syscall in kernel/proc.c II

```
  /* Check if the process has privileges for the requested call. Calls to the
   * kernel may only be SENDREC, because tasks always reply and may not block
   * if the caller doesn't do receive().
   */
  if (! (priv(caller_ptr)->s_trap_mask & (1 « function)) ||
          (iskerneln(src_dst) && function != SENDREC
           && function != RECEIVE)) {
#if DEBUG_ENABLE_IPC_WARNINGS
      kprintf(sys_call: trap %d not allowed, caller %d, src_dst %d\n,
          function, proc_nr(caller_ptr), src_dst);
#endif
      return(ETRAPDENIED);              /* trap denied by mask or kernel */
  }
```

_syscall in kernel/proc.c III

```
  /* Require a valid source and/ or destination process, unless echoing. */
  if (src_dst != ANY && function != ECHO) {
      if (! isokprocn(src_dst)) {
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf(sys_call: invalid src_dst, src_dst %d, caller %d\n,
              src_dst, proc_nr(caller_ptr));
#endif
          return(EBADSRCDST);           /* invalid process number */
      }
      if (isemptyn(src_dst)) {
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf(sys_call: dead src_dst; trap %d, from %d, to %d\n,
              function, proc_nr(caller_ptr), src_dst);
#endif
          return(EDEADSRCDST);
      }
  }
```

_syscall in kernel/proc.c IV

```
  /* If the call involves a message buffer, i.e., for SEND, RECEIVE, SENDREC,
   * or ECHO, check the message pointer. This check allows a message to be
   * anywhere in data or stack or gap. It will have to be made more elaborate
   * for machines which don't have the gap mapped.
   */
  if (function & CHECK_PTR) {
      vlo = (vir_bytes) m_ptr >> CLICK_SHIFT;
      vhi = ((vir_bytes) m_ptr + MESS_SIZE - 1) >> CLICK_SHIFT;
      if (vlo < caller_ptr->p_memmap[D].mem_vir || vlo > vhi ||
              vhi >= caller_ptr->p_memmap[S].mem_vir +
              caller_ptr->p_memmap[S].mem_len) {
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf(sys_call: invalid message pointer, trap %d, caller %d\n,
                function, proc_nr(caller_ptr));
#endif
          return(EFAULT);               /* invalid message pointer */
      }
  }
```

_syscall in kernel/proc.c V

```
  /* 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 (function & CHECK_DST) {
      if (! get_sys_bit(priv(caller_ptr)->s_ipc_to, nr_to_id(src_dst))) {
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf(sys_call: ipc mask denied trap %d from %d to %d\n,
                function, proc_nr(caller_ptr), src_dst);
#endif
          return(ECALLDENIED);          /* call denied by ipc mask */
      }
  }

  /* Check for a possible deadlock for blocking SEND(REC) and RECEIVE. */
  if (function & CHECK_DEADLOCK) {
      if (group_size = deadlock(function, caller_ptr, src_dst)) {
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf(sys_call: trap %d from %d to %d deadlocked, group size %d\
              function, proc_nr(caller_ptr), src_dst, group_size);
#endif
```

_syscall in kernel/proc.c VI

```
          return(ELOCKED);
      }
  }
```

_syscall in kernel/proc.c VII

```
  switch(function) {
  case SENDREC:
      /* A flag is set so that notifications cannot interrupt SENDREC. */
      priv(caller_ptr)->s_flags |= SENDREC_BUSY;
      /* fall through */
  case SEND:
      result = mini_send(caller_ptr, src_dst, m_ptr, flags);
      if (function == SEND || result != OK) {
          break;                                /* done, or SEND failed */
      }                                         /* fall through for SENDREC *
  case RECEIVE:
      if (function == RECEIVE)
          priv(caller_ptr)->s_flags &= ~SENDREC_BUSY;
      result = mini_receive(caller_ptr, src_dst, m_ptr, flags);
      break;
  case NOTIFY:
      result = mini_notify(caller_ptr, src_dst);
      break;
  case ECHO:
      CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, caller_ptr, m_ptr);
```

_syscall in kernel/proc.c VIII

```
   result = OK;
   break;
 default:
   result = EBADCALL;                        /* illegal system call */
 }

 /* Now, return the result of the system call to the caller. */
 return(result);
}
```

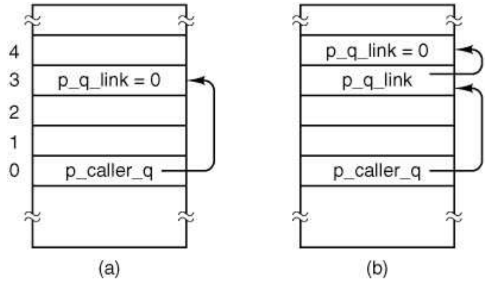
mini_send in kernel/proc.c I

```
PRIVATE int mini_send(caller_ptr, dst, m_ptr, flags)
register struct proc *caller_ptr;       /* who is trying to send a message? *
int dst;                                /* to whom is message being sent? */
message *m_ptr;                         /* pointer to message buffer */
unsigned flags;                         /* system call flags */
{
/* Send a message from 'caller_ptr' to 'dst'. If 'dst' is blocked waiting
 * for this message, copy the message to it and unblock 'dst'. If 'dst' is
 * not waiting at all, or is waiting for another source, queue 'caller_ptr'.
 */
 register struct proc *dst_ptr = proc_addr(dst);
 register struct proc **xpp;
 /* 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,
```

mini_send in kernel/proc.c II

```
                dst_ptr->p_messbuf);
        if ((dst_ptr->p_rts_flags &= ~RECEIVING) == 0) enqueue(dst_ptr);
 } else if ( ! (flags & NON_BLOCKING)) {
        /* Destination is not waiting.  Block and dequeue caller. */
        caller_ptr->p_messbuf = m_ptr;
        if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
        caller_ptr->p_rts_flags |= SENDING;
        caller_ptr->p_sendto = dst;

        /* Process is now blocked.  Put in on the destination's queue. */
        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;        /* mark new end of list */
 } else {
        return(ENOTREADY);
 }
 return(OK);
}
```



Il process manager I

servers/pm/main.c

```
PUBLIC int main()
{
  ...
  pm_init();                    /* initialize process manager tables */

  /* This is PM's main loop-  get work and do it, forever and forever. */
  while (TRUE) {
    get_work();             /* wait for an PM system call */
    ...
    if ((unsigned) call_nr >= NCALLS) {
          result = ENOSYS;
      } else {
          result = (*call_vec[call_nr])();
      }
    }

    /* Send the results back to the user to indicate completion. */
    if (result != SUSPEND) setreply(who, result);
```

Il process manager II

```
swap_in();              /* maybe a process can be swapped in? */

/* Send out all pending reply messages, including the answer to
 * the call just made above.  The processes must not be swapped out.
 */
for (proc_nr=0, rmp=mproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {
  /* In the meantime, the process may have been killed by a
   * signal (e.g. if a lethal pending signal was unblocked)
   * without the PM realizing it. If the slot is no longer in
   * use or just a zombie, don't try to reply.
   */
  if ((rmp->mp_flags & (REPLY | ONSWAP | IN_USE | ZOMBIE)) ==
     (REPLY | IN_USE)) {
          if ((s=send(proc_nr, &rmp->mp_reply)) != OK) {
                  panic(__FILE__,PM can't reply to, proc_nr);
          }
          rmp->mp_flags &= ~REPLY;
  }
}
```

La tabella delle syscall in servers/pm/table.c

```
_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
  no_sys,         /*  0 = unused  */
  do_pm_exit,     /*  1 = exit    */
  do_fork,        /*  2 = fork    */
  no_sys,         /*  3 = read    */
  no_sys,         /*  4 = write   */
  no_sys,         /*  5 = open    */
  no_sys,         /*  6 = close   */
  do_waitpid,     /*  7 = wait    */
...
  do_times,       /* 43 = times   */
...
  do_time,        /* 90 = gettimeofday */
};
```

L'implementazione di times(2) ... finalmente! O quasi

```
PUBLIC int do_times()
{
/* Perform the times(buffer) system call. */
  register struct mproc *rmp = mp;
  clock_t t[5];
  int s;

  if (OK != (s=sys_times(who, t)))
      panic(__FILE__,do_times couldn't get times, s);
  rmp->mp_reply.reply_t1 = t[0];                /* user time */
  rmp->mp_reply.reply_t2 = t[1];                /* system time */
  rmp->mp_reply.reply_t3 = rmp->mp_child_utime; /* child user time */
  rmp->mp_reply.reply_t4 = rmp->mp_child_stime; /* child system time */
  rmp->mp_reply.reply_t5 = t[4];                /* uptime since boot */

  return(OK);
}
```

L’implementazione di sys_times è nel system task

kernel/system/do_times.c

```
PUBLIC int do_times(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_times().  Retrieve the accounting information. */
  register struct proc *rp;
  int proc_nr;
  /* Insert the times needed by the SYS_TIMES kernel call in the message.
   * The clock's interrupt handler may run to update the user or system time
   * while in this code, but that cannot do any harm.
   */
  proc_nr = (m_ptr->T_PROC_NR == SELF) ? m_ptr->m_source : m_ptr->T_PROC_NR;
  if (isokprocn(proc_nr)) {
      rp = proc_addr(m_ptr->T_PROC_NR);
      m_ptr->T_USER_TIME   = rp->p_user_time;
      m_ptr->T_SYSTEM_TIME = rp->p_sys_time;
  }
  m_ptr->T_BOOT_TICKS = get_uptime();
  return(OK);
}
```