```
// sistema.cpp
//
#include "costanti.h"
#include "libce.h"

//////////////////////////////////////////////////////////////////////////////
//                      PROCESSI                                            //
//////////////////////////////////////////////////////////////////////////////
const natl MAX_PRIORITY = 0xfffffff;
const natl MIN_PRIORITY = 0x0000001;
const natl DUMMY_PRIORITY = 0x0000000;
const int N_REG = 16;   // numero di registri nel campo contesto


// EXTENSION 2017-01-18

/**
 * Available broadcaste roles.
 */
enum broadcast_role
{
    B_NONE,         // no role is assigned when the process is created
    B_BROADCASTER,  // broadcaster (can use the broadcast() primitive)
    B_LISTENER      // listener (can use the listen() primitive)
};

// EXTENSION 2017-01-18

// si veda in PAGINAZIONE per il significato di questi typedef
typedef natq vaddr;
typedef natq faddr;
typedef natq tab_entry;

// descrittore di processo
struct des_proc {
        // parte richiesta dall'hardware
        struct __attribute__ ((packed)) {
                natl riservato1;
                vaddr punt_nucleo;
                // due quad  a disposizione (puntatori alle pile ring 1 e 2)
                natq disp1[2];
                natq riservato2;
                //entry della IST, non usata
                natq disp2[7];
                natq riservato3;
                natw riservato4;
                natw iomap_base; // si veda crea_processo()
        };
        //finiti i campi obbligatori
        faddr cr3;
        natq contesto[N_REG];
        natl cpl;

// EXTENSION 2017-01-18

    // process broadcast role
        broadcast_role b_reg;

// EXTENSION 2017-01-18

// SOLUTION 2017-01-18

    // process last retrieved broadcast message id
    natl b_id;

// SOLUTION 2017-01-18
};

// [...]
```

```
/**
 * Broadcast descriptor struct.
 */
struct broadcast
{
    // true if the broadcaster is registered
    bool broadcaster_registered;

// ( SOLUZIONE 2017-01-18

    // last broadcast message id
    natl last_id;

    // sent broadcast messages array
    natl msg[MAX_BROADCAST];

    // registered listeners array
    proc_elem *listeners;

//   SOLUZIONE 2017-01-18 )
};

/**
 * System global broadcast descriptor.
 */
broadcast global_broadcast;

/**
 * Initializes the global broadcast descriptor.
 */
void broadcast_init()
{
    // no initial broadcaster registered
    global_broadcast.broadcaster_registered = false;
// ( SOLUZIONE 2017-01-18

    // no broadcast messages registered at initialization
    global_broadcast.last_id = 0;

    // no initial listeners registered
    global_broadcast.listeners = 0;

//   SOLUZIONE 2017-01-18 )
}

// ( SOLUZIONE 2016-09-20

/**
 * Registers a process to the global broadcast with the specified broadcast
 * role. If the given role is not valid (!B_BROADCASTER and !B_LISTENER) or
 * if the given process is already registered, or if there is already a
 * broadcaster registered the current process must be aborted.
 *
 * @param  role  the of the process being registered (either B_BROADCASTER or
 *               B_LISTENER).
 */
extern "C" void c_reg(enum broadcast_role role)
{
    // retrieve calling process descriptor
    struct des_proc *p = des_p(esecuzione->id);

    // retrieve global broadcast descriptor
    struct broadcast *b = &global_broadcast;

    // check if the given broadcast role is valid: B_NONE is invalid
    if (role != B_BROADCASTER && role != B_LISTENER)
    {
        // print warning log message
        flog(LOG_WARN, "Invalid broadcast role: %d", role);
```

```
        // abort calling process
        c_abort_p();

        // just return
        return;
        }

    // check if the process is already registered to the global broadcast
    if (p->b_reg != B_NONE)
    {
        // print warning log message
        flog(LOG_WARN, "Process already registered as %s",
             (p->b_reg == B_BROADCASTER ? "broadcaster." : "listener."));

        // abort current process under execution
        c_abort_p();

        // just return
        return;
        }

    // check if the given role is broadcaster
    if (role == B_BROADCASTER)
    {
        // check if there is already a registered broadcaster
        if (b->broadcaster_registered)
        {
            // if so, print a warning log message
            flog(LOG_WARN, "Broadcaster already registered.");

            // abort current process under execution
            c_abort_p();

            // just return
            return;
        }

        // set broadcaster registered to true
        b->broadcaster_registered = true;
    }

    // update process broadcast role
    p->b_reg = role;
}

/**
 * Called by listener processes to retrieve the next broadcast message. If the
 * process has already received all the broadcast messages it must be placed in
 * the wait queue for the next broadcast message. If the process is not a
 * registered listener it must be aborted.
 */
extern "C" void c_listen()
{
    // retrieve calling process descriptor
    struct des_proc *p = des_p(esecuzione->id);

    // retrieve global broadcast descriptor
    struct broadcast *b = &global_broadcast;

    // check if the current process is not a registered listener
    if (p->b_reg != B_LISTENER)
    {
        // if so, print a warning log message
        flog(LOG_WARN, "Process not registered as listener.");

        // abort calling process
        c_abort_p();

        // just return to the caller
        return;
```

```
    }

    // check if there are broadcast messages to be retrieved
    if (p->b_id < b->last_id)
    {
        // if so, retrieve they nex broadcast message
        p->contesto[I_RAX] = b->msg[p->b_id];

        // increase last retrieved broadcast message id
        p->b_id++;

        // just return to the caller
        return;
    }

    // otherwise, insert the current process in the listeners processes queue:
    // it will have to wait until another broadcast message is sent by the
    // broadcaster process in which case it will receive the broadcast message
    // and be placed in the system ready processes queue and eventually
    // rescheduled
    inserimento_lista(b->listeners, esecuzione);

    // schedule a new process
    schedulatore();
}

/**
 * Sends the given broadcast message. It must check if the calling process is
 * registered as broadcaster and if the maximum number of broadcast messages is
 * not exceeded. If both conditions are not met the calling processes is
 * aborted.
 *
 * @param  msg  the broadcast message to be sent.
 */
extern "C" void c_broadcast(natl msg)
{
    // retrieve current process descriptor
    struct des_proc *p = des_p(esecuzione->id);

    // retrieve global broadcast descriptor
    struct broadcast *b = &global_broadcast;

    // check if the current process is registered as broadcaster
    if (p->b_reg != B_BROADCASTER)
    {
        // if not, print a warning log message
        flog(LOG_WARN, "Broadcast message from invalid process.");

        // abort current process under execution
        c_abort_p();

        // just return
        return;
    }

    // check if the number of maximum broadcast messages has been reached
    if (b->last_id >= MAX_BROADCAST)
    {
        // if so, print a warning log message
        flog(LOG_WARN, "Too many broadcast messages.");

        // abort the current process under execution
        c_abort_p();

        // just retun
        return;
    }

    // set broadcast message
    b->msg[b->last_id] = msg;
```

```
    // increase last broadcast message id
    b->last_id++;

    // insert the current process at the top of the ready processes queue
    inspronti();

    // deliver the new broadcast message to all listeners in the wait queue:
    // theese processes have already retrieved all previous broadcast messages
    // and called the listen() primitive one more time which resulted for them
    // beign placed in the global broadcaster descriptor listeners wait queue
    while (b->listeners)
    {
        // process descriptor
        struct proc_elem *work;

        // extract top indexed listener process
        rimozione_lista(b->listeners, work);

        // retrieve process descriptor
        struct des_proc *w = des_p(work->id);

        // deliver broadcast message to the listener process
        w->contesto[I_RAX] = msg;

        // increase listener process broadcast messages last id
        w->b_id++;

        // insert the listener process in the system ready processes queue
        inserimento_lista(pronti, work);
    }

    // schedule a new process
    schedulatore();
}

// SOLUTION 2016-09-20

// [...]

/**
 * In this new implementation of the broadcast system there can be multiple
 * broadcaster processes. However, only one process can be active with the role
 * of broadcaster. When each process is destroyed we have to check if it is the
 * broadcaster process and in that case remove the broadcaster from the global
 * broadcast descriptor.
 */
void distruggi_processo(proc_elem* p)
{
        des_proc* pdes_proc = des_p(p->id);

// EXTENSION 2016-09-20

    // check if the process is a broadcaster
    if (pdes_proc->b_reg == B_BROADCASTER)
    {
        // if so, remove the global broadcast broadcaster
        global_broadcast.broadcaster_registered = false;
    }

// EXTENSION 2016-09-20

        faddr tab4 = pdes_proc->cr3;
        riassegna_tutto(p->id, tab4, I_MIO_C, N_MIO_C);
        riassegna_tutto(p->id, tab4, I_UTN_C, N_UTN_C);
        rilascia_tutto(tab4, I_UTN_P, N_UTN_P);
        ultimo_terminato = tab4;
        if (p != esecuzione) {
                distruggi_pila_precedente();
        }
```

```
        rilascia_tss(id_to_tss(p->id));
        dealloca(pdes_proc);
}

// [...]

void main_sistema(int n)
{
        natl sync_io;

        // ( caricamento delle tabelle e pagine residenti degli spazi condivisi ()
        flog(LOG_INFO, "creazione o lettura delle tabelle e pagine residenti condivise...
");
        if (!crea_spazio_condiviso())
                goto error;
        // )

        gdb_breakpoint();

        // ( inizializzazione del modulo di io
        flog(LOG_INFO, "creazione del processo main I/O...");
        sync_io = sem_ini(0);
        if (sync_io == 0xFFFFFFFF) {
                flog(LOG_ERR, "Impossibile allocare il semaforo di sincr per l'IO");
                goto error;
        }
        // occupiamo l'entrata del timer
        aggiungi_pe(ESTERN_BUSY, 2);
        if (activate_p(swap_dev.sb.io_entry, sync_io, MAX_PRIORITY, LIV_SISTEMA) == 0xFFF
FFFFF) {
                flog(LOG_ERR, "impossibile creare il processo main I/O");
                goto error;
        }
        flog(LOG_INFO, "attendo inizializzazione modulo I/O...");
        sem_wait(sync_io);
        // )

        // ( creazione del processo start_utente
        flog(LOG_INFO, "creazione del processo start_utente...");
        if (activate_p(swap_dev.sb.user_entry, 0, MAX_PRIORITY, LIV_UTENTE) == 0xFFFFFFFF
) {
                flog(LOG_ERR, "impossibile creare il processo main utente");
                goto error;
        }
        // )
        // (* attiviamo il timer
        attiva_timer(DELAY);
        flog(LOG_INFO, "attivato timer (DELAY=%d)", DELAY);
        // *)

// ( ESAME 2017-01-18

    // initialize global broadcast descriptor
        broadcast_init();

//   ESAME 2017-01-18 )

        // ( terminazione
        flog(LOG_INFO, "passo il controllo al processo utente...");
        terminate_p();
        // )
error:
        panic("Errore di inizializzazione");
}
```