```
/**
 * File: system.cpp
 *       System Module C++ implementation.
 *
 * Author: Rambod Rahmani <rambodrahmani@autistici.org>
 *         Created on 30/08/2019.
 */

#include "constants.h"
#include <libqlk.h>
#include <log.h>
#include <apic.h>

////////////////////////////////////////////////////////////////////////////////
//                                  PROCESSES                                 //
////////////////////////////////////////////////////////////////////////////////

/**
 * Maximum process priority.
 */
const natl MAX_PRIORITY = 0xfffffff;

/**
 * Minimum process priority.
 */
const natl MIN_PRIORITY = 0x0000001;

/**
 * Dummy processo priority.
 */
const natl DUMMY_PRIORITY = 0x0000000;

/**
 * Number of registers of the contest array field of the des_proc struct.
 */
const int N_REG = 16;

/**
 * Memory Virtual Address.
 */
typedef natq vaddr;

/**
 * Memory Physical Address.
 */
typedef natq faddr;

/**
 *
 */
typedef natq tab_entry;

/**
 * Process Descriptor. Each process has its own process descriptor, a system
 * stack and its own memory (which contains its code, data and user stack).
 * In order to be able to switch between processes and allow for a little
 * parallel execution we will have to take a full snapshot of the system state
 * (CPU, memory, devices etc..) in order to be able to come back to the
 * execution where it has been left.
 */
struct des_proc
{
    // hardware required
    struct __attribute__ ((packed))
    {
        natl riservato1;

        /**
         * Each process has its own system stack and the way the system stack is
         * changed moving from one process to another is up to the hardware
```

```
         * interrupt mechanism. We will place this pointer to the process system
         * stack where we know the hardware will look for it. When the process
         * is started, the CPU will save in this stack the pointer to the
         * previous stack, the content of the RFLAGS register, the previous
         * privilege level, and the address of the next instruction to be
         * executed.
         * When a process is at user level its system stack is always empty. The
         * system stack will be filled when moving to the system level and
         * emptied out when returning to user level.
         */
        vaddr system_stack;

        // 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()
    };

    // custom data
    faddr cr3;

    // process context: contains a copy of the CPU registers content
    natq context[N_REG];

    natl cpl;

/**
 * New fields must be added to the process descriptor in order to be able to
 * distinguish between master and slave process.
 */

// EXTENSION 2019-07-24

    /**
     * Pointer to the master process. Might be null if no master has been
     * defined for this process.
     */
    des_proc *bp_master;

    /**
     * Pointer to the slave process. Might be null if no slave has been defined
     * for this process.
     */
    des_proc *bp_slave;

    /**
     * Breakpoint instruction address. Meaningful only for a master process.
     */
    vaddr bp_addr;

    /**
     * Original byte contained at the address where the breakpoint has been
     * placed. Meaningful only for a master process.
     */
    natb bp_orig;

    /**
     * Slave process ID. Meaningful only for a master process.
     */
    natl bp_slave_id;

    /**
     * Process queue which can be used by the master or the slave process to
     * wait.
     */
```

```
    struct proc_elem  *bp_waiting;

// EXTENSION 2019-07-24
};

// [...]

// EXTENSION 2019-07-24

/**
 * Can be used only by master processes to be palced in the corresponding slave
 * process wait queue for them to reach the breakpoint address. Theese master
 * processes must have a bp_slave process set in their descriptor.
 */
extern "C" void c_bpwait()
{
    // retrieve calling process descriptor
    des_proc *self = des_p(esecuzione->id);

    // check if the calling process is a master process
    if (!self->bp_slave)
    {
        // if so,  print a warning log message
        flog(LOG_WARN, "Only master processes can use the bpwait() primitive.");

        // abort calling process: it must be a master process to use this
        // primitive
        c_abort_p();
    }

    // check if there is a slave process waiting
    if (!self->bp_waiting)
    {
        // if not, insert the current process in the slave process waiting queue
        self->bp_slave->bp_waiting = esecuzione;

        // schedule a new process
        schedulatore();
    }
}

/**
 * This subroutine is called in case of a breakpoint exception.
 *
 * @param  tipo    exception type (3);
 * @param  errore  exception error (0);
 * @param  p_rip   address contained in %rsp.
 */
extern "C" void c_breakpoint_exception(int tipo, natq errore, vaddr *p_rip)
{
    // retrieve calling process descriptor
    des_proc *self = des_p(esecuzione->id);

    // check if the calling process is a slave process: this breakpoint was
    // added using the bpattach() primitive
    if (!self->bp_master)
    {
        // if not, just handle the exception: the gestore_eccezioni will also
        // abort the calling process as all interrupt 3 not placed using the
        // bpattach must be aborted
        gestore_eccezioni(tipo, errore, *p_rip);

        // and return to the caller
        return;
    }

    // retrieve current rsp value
    natq* rip = reinterpret_cast<natq*>(p_rip);

    // decrease it by one
```

```
    (*rip)--;

    // insert the current slave process in the master process bp queue
    self->bp_master->bp_waiting = esecuzione;

    // check if the master process is in this slave process bp queue
    if (self->bp_waiting)
    {
        // if so, move the process in the system ready processes queue
        inserimento_lista(pronti, self->bp_waiting);

        // clear waiting master processes for this slave process
        self->bp_waiting = 0;
    }

    // schedule a new process
    schedulatore();
}

// normalmente, le parti condivise della memoria virtuale (come quelle che
// contengono il codice dei processi) sono realizzate condividendo tutto a
// partire dalle tabelle di livello 3.  Questa funzione installa nell'albero di
// traduzione del processo id una copia privata del percorso di traduzione e
// della pagina che contiene l'indirizzo v, creando, copiando e modificando
// opportunamente le tabelle di livello 3, 2 e 1 e la pagina finale.
bool duplica(natl id, vaddr v)
{
        // per ogni livello a partire dal 3 andando a scendere...
        for (int i = 3; i >= 0; i--) {
                // prendiamo l'entrata della tabella di livello superiore
                // che punta all'entita' che stiamo considerando
                tab_entry& e = get_des(id, i + 1, v);

                // allochiamo un frame per la copia
                des_frame *df_dst = alloca_frame(id, i, v);
                if (!df_dst) {
                        flog(LOG_WARN, "memoria esaurita");
                        return false;
                }
                // riempiamo i campi del descrittore di frame
                // (per consistenza e debugging)
                df_dst->processo = id;
                df_dst->residente = true;
                df_dst->livello = i;
                df_dst->ind_virtuale = v;
                df_dst->ind_massa = 0;
                df_dst->contatore = 0;

                // copiamo l'entita' vecchia nel nuovo frame
                faddr src = extr_IND_FISICO(e);
                faddr dst = indirizzo_frame(df_dst);
                memcpy(reinterpret_cast<void *>(dst), reinterpret_cast<void *>(src), 4096
);
                // facciamo in modo che la tabella di livello superiore punti alla copia
                set_IND_FISICO(e, dst);
        }
        return true;
}

/**
 * Attaches a breakpoint in the processes having the given id at the instruction
 * having the given address.
 *
 * @param  id   slave destination process id;
 * @param  rip  instruction address.
 */
extern "C" void c_bpattach(natl id, vaddr rip)
{
    // retrieve calling process descriptor (the master one)
    des_proc *self = des_p(esecuzione->id);
```

```
    // retrieve slave process id
    des_proc *dest = des_p(id);

    // check if the calling process (master) and the slave process are the same
    // process
    if (dest == self)
    {
        // if so, print a warning log message
        flog(LOG_WARN, "A master process can not call the bpattach() on itself.");

        // abort calling process
        c_abort_p();

        // just return to the caller
        return;
    }

    // chec if the given instruction address belongs to the user process shared
    // memory area
    if (rip < ini_utn_c || rip >= fin_utn_c)
    {
        // if not, print a warning log message
        flog(LOG_WARN, "rip %p out of bounds [%p, %p)", rip, ini_utn_p, fin_utn_p);

        // abort calling process
        c_abort_p();

        // return to the caller
        return;
    }

    // set return value
    self->contesto[I_RAX] = false;

    // check if the slave process is valid, if it does not have a master process
    // already set nor a slave process
    if (!dest || dest->bp_master || dest->bp_slave)
    {
        // otherwise, just return to the caller
        return;
    }

    // set slave process for the calling master process
    self->bp_slave = dest;

    // no slave processes waiting in the master bp queue
    self->bp_waiting = 0;

    // set slave process id for the calling master process
    self->bp_slave_id = id;

    // set master process for the destination slave process
    dest->bp_master = self;

    // no master processes waiting in the slave bp queue
    dest->bp_waiting = 0;

    // update return value
    self->contesto[I_RAX] = true;

// SOLUTION 2019-07-24

    // create private memory area for the given instruction address
    if (!duplica(id, rip))
    {
        // in case of error, just return to the caller
        return;
    }
```

```
    // retrieve byte pointed by the given address
    natb *bytes = reinterpret_cast<natb*>(trasforma(id, rip));

    // set breakpoint address
    self->bp_addr = rip;

    // save original instruction byte
    self->bp_orig = *bytes;

    // replace addressed byte with the int3 opcode byte
    *bytes = 0xCC;

//   SOLUZIONE 2019-07-24 )
}
//   ESAME 2019-07-24 )

// ( SOLUZIONE 2019-07-24

/**
 * Undoes the operation performed by the bpattach at memory level.
 */
void deduplica(natl id, vaddr v)
{
        for (int i = 0; i <= 3; i++) {
                tab_entry e = get_des(id, i + 1, v);
                faddr dst = extr_IND_FISICO(e);
                des_frame *pf_dst = descrittore_frame(dst);
                rilascia_frame(pf_dst);
        }
        tab_entry& e_slave = get_des(id, 4, v);
        tab_entry e_master = get_des(esecuzione->id, 4, v);
        e_slave = e_master;
}

/**
 * Removes the breakpoint inserted by the master process and reschedules the
 * slave process.
 */
extern "C" void c_bpdetach()
{
    // retrieve calling process descriptor
    des_proc *self = des_p(esecuzione->id);

    // destination (slave) process descriptor
    des_proc *dest;

    // check if the calling process is a master process (actually has a slave
    // process)
    if (!self->bp_slave)
    {
        // if not, print a warning log message
        flog(LOG_WARN, "Only master processes can use the bpdetach() primitive.");

        // abort calling process
        c_abort_p();

        // just return to the caller
        return;
    }

    // set retrieved slave process as destination process
    dest = self->bp_slave;

    // undo the duplication operation performed by the bpattach()
    deduplica(self->bp_slave_id, self->bp_addr);

    // check if there is a slave process in the master process bp queue
    if (self->bp_waiting)
    {
        // insert the calling master process in the system ready processes list
```

```
        inspronti();

        // insert the slave process in the system ready processes list
        inserimento_lista(pronti, self->bp_waiting);

        // clear any waiting slave processes
        self->bp_waiting = 0;

        // schedule a new process
        schedulatore();
        }

    // no more waiting slave process for the master process
    self->bp_slave = 0;

    // no more breakpoint address for the master process
    self->bp_addr = 0;

    // no more original byte for the master process
    self->bp_orig = 0;

    // no more slave process ID for the master process
    self->bp_slave_id = 0;

    // no more waiting master process for the slave process
    dest->bp_master = 0;
}

// SOLUTION 2019-07-24
```