# 2024-06-28 - Checkpoint processi

Aggiungiamo al nucleo il meccanismo dei *checkpoint*.

Il checkpoint di un processo è la foto del suo stato in un certo istante, dato dal contenuto dei registri del processore e contenuto di tutta la sua memoria privata. Nel nostro caso la memoria privata comprende la pila sistema e la pila utente, ma per evitare di dover rivelare il contenuto della pila sistema, permetteremo i checkpoint solo negli istanti in cui un processo sta per ritornare a livello utente (e dunque la sua pila sistema è vuota).

Nel meccanismo che vogliamo realizzare, un processo (master) si prepara a ricevere i checkpoint da un altro processo (slave) tramite la primitiva `cp_prep(pid)`. La primitiva restituisce al master l'indirizzo a cui verrà salvato lo stato della memoria privata (pila utente, per quanto detto sopra) dello slave ad ogni checkpoint. Da quel momento in poi, ogni volta che lo slave sta per tornare a livello utente si dovrà sincronizzare con il master per trasferigli un nuovo checkpoint. Il master si può mettere in attesa del prossimo checkpoint tramite la primitiva `cp_get(regs)`, dove `regs` è un puntatore ad un vettore di 16 `natq`, destinato a ricevere il contenuto dei registri dello slave. Dopo il trasferimento del checkpoint entrambi i processi ritornano pronti. La relazione di master/slave prosegue fino a quando uno dei due processi non termina. Ogni processo può essere al più master o slave di un altro processo.

Per realizzare il meccanismo appena descritto introduciamo la seguente struttura dati:

```
struct des_cp {
    /// id del processo master
    natl master;
    /// id del processo slave
    natl slave;
    /// vettore regs che il master ha passato a cp_reg() (se master_waiting è true)
    natq *cp_regs_master;
    /// true se lo slave è in attesa di trasferire il checkpoint
    bool slave_waiting;
    /// true se il master è in attesa di trasferire il checkpoint
    bool master_waiting;
};
```

La struttura `des_cp` descrive lo stato di una coppia master/slave. Il campo `master` è l'id del processo master; il campo `slave` è l'id del processo slave; i campi `slave_waiting` e `master_waiting` valgono true se il corrispondente processo è in attesa di trasferire/ricevere un nuovo checkpoint; se il processo master è in attesa, il campo `cp_regs_master` contiene il campo `regs` che il master ha passato a `cp_get()`.

Aggiungiamo anche il seguente campo ai descrittori di processo:

```
struct des_proc {
    ...
    /// puntatore al descrittore di checkpoint (non nullo solo per i processi master e slave
    des_cp *cp;
};

void distruggi_processo(des_proc* p)
{
    paddr root_tab = p->cr3;
    if (des_cp *cp = p->cp)
    {
        des_proc *master = proc_table[cp->master],
             *slave  = proc_table[cp->slave];
        if (p == slave && cp->master_waiting) {
            master->contesto[I_RAX] = false;
            inserimento_lista(pronti, master);
        } else if (p == master) {
            if (slave && cp->slave_waiting)
                inserimento_lista(pronti, slave);
            unmap(p->cr3, ini_utn_p, ini_utn_p + DIM_USR_STACK,
                    [] (vaddr, paddr p, int) { rilascia_frame(p); });
        }
        master->cp = slave->cp = nullptr;
        delete cp;
    }
    ...
}
```

Nei processi che si trovano in relazione master/slave, questo campo punta ad un'istanza di `des_cp` (condivisa da entrambi i processi). Per i processi che non sono né master, né slave, questo campo è nullo.

Aggiungiamo infine le seguenti primitive:

- `void* cp_prep(natl pid)` (già realizzata): registra il processo chiamante come master del processo di identificatore `pid`, che diventa slave. Il processo chiamante non deve essere già master e lo slave deve essere un processo di livello utente diverso dal chiamante; restituisce `nullptr` in caso di fallimento (il processo chiamante è slave, il processo `pid` non esiste o è già slave o master, qualche allocazione di memoria è fallita), altrimenti restituisce l'indirizzo di una zona di memoria (nel master) pronta a contenere lo stato della memoria privata dello slave.
- `bool cp_get(natq* regs)`: Il processo chiamante deve essere master. Il master si pone in attesa del prossimo checkpoint da parte dello slave. Quando la primitiva ritorna, `regs` contiene lo stato dei registri, mentre lo stato della memoria privata si trova all'indirizzo precedentemente ricevuto tramite `cp_prep()`. La primitiva restituisce `false` se il processo slave è

terminato prima di poter inviare un checkpoint.

Le primitive abortiscono il processo chiamante in caso di errore e tengono conto della priorità tra i processi. Si può assumere che la pila sistema occupi una singola pagina.

```
extern "C" void c_cp_prep(natl pid)
{
    if (pid >= MAX_PROC_ID) {
        flog(LOG_WARN, "cp_prep: id non valido: %u", pid);
        c_abort_p();
        return;
    }

    esecuzione->contesto[I_RAX] = 0;

    if (esecuzione->cp) {
        if (esecuzione->cp->master == esecuzione->id) {
            flog(LOG_WARN, "c_prep: il processo e' gia' master");
            c_abort_p();
        }
        return;
    }

    des_proc *slave = proc_table[pid];
    if (slave == nullptr) {
        return;
    }

    if (slave->livello != LIV_UTENTE) {
        flog(LOG_WARN, "cp_prep: id non permesso: %u", pid);
        c_abort_p();
        return;
    }

    if (slave == esecuzione) {
        flog(LOG_WARN, "cp_prep: self-checkpoint non ammesso");
        c_abort_p();
        return;
    }

    if (slave->cp)
        return;

    vaddr beg = ini_utn_p;
    vaddr end = beg + DIM_USR_STACK;
    vaddr v = map(esecuzione->cr3, beg, end, BIT_US|BIT_RW, [] (vaddr) { return alloca_frame
```

```
    if (v != end) {
        unmap(esecuzione->cr3, beg, v, [](vaddr, paddr p, int) { rilascia_frame(p); });
        return;
    }

    des_cp *cp = new des_cp;
    if (cp == nullptr)
        return;

    cp->master = esecuzione->id;
    cp->slave = pid;
    cp->cp_regs_master = nullptr;
    cp->slave_waiting = false;
    cp->master_waiting = false;
    esecuzione->cp = cp;
    slave->cp = cp;

    esecuzione->contesto[I_RAX] = beg;
}
```

Modificare il file `sistema.cpp` in modo da realizzare le parti mancanti.

**Suggerimento**: per riconoscere quando un processo sta per tornare a livello utente ci si può ispirare alla funzione `liv_chiamante()`.

```
extern "C" void c_cp_get(natq *regs)
{
    des_cp *cp = esecuzione->cp;

    if (cp == nullptr || cp->master != esecuzione->id) {
        flog(LOG_WARN, "cp_get: processo non master");
        c_abort_p();
        return;
    }

    if (!c_access(int_cast<vaddr>(regs), (N_REG + 1) * sizeof(natq), true, true)) {
        flog(LOG_WARN, "cp_get: regs non valido");
        c_abort_p();
        return;
    }

    des_proc *slave = proc_table[cp->slave];
    if (slave == nullptr) {
        esecuzione->contesto[I_RAX] = false;
        return;
    }
```

```
    if (cp->slave_waiting) {
        esecuzione->contesto[I_RAX] = true;
        memcpy(regs, slave->contesto, sizeof(slave->contesto));
        natq *pila = ptr_cast<natq>(trasforma(slave->cr3, slave->contesto[I_RSP]));
        regs[I_RSP] = pila[3];
        regs[N_REG] = pila[0];
        for (vaddr src =  fin_utn_p - DIM_USR_STACK, dst = ini_utn_p;
               src != fin_utn_p;
               src += DIM_PAGINA, dst += DIM_PAGINA)
        {
            paddr psrc = trasforma(slave->cr3, src);
            memcpy(voidptr_cast(dst), voidptr_cast(psrc), DIM_PAGINA);
        }
        cp->slave_waiting = false;
        inspronti();
        inserimento_lista(pronti, slave);
    } else {
        cp->cp_regs_master = regs;
        cp->master_waiting = true;
    }
    schedulatore();
}

/**
 * @brief Trasferisce il checkpoint se richiesto.
 *
 * La funzione viene invocata all'inizio di carica_stato. Deve controllare se
 * si sta tentando di caricare lo stato di uno slave con ritorno a livello utente e,
 * in tal caso, sincronizzarsi con il master per trasferire il checkpoint.
 */
extern "C" void maybe_cp()
{
    des_cp *cp = esecuzione->cp;

    if (cp == nullptr || esecuzione->id != cp->slave)
        return;

    natq *pila = ptr_cast<natq>(trasforma(esecuzione->cr3, esecuzione->contesto[I_RSP]));
    if (pila[1] != SEL_CODICE_UTENTE)
        return;

    if (cp->master_waiting) {
        des_proc *master = proc_table[cp->master];
        master->contesto[I_RAX] = true;
        memcpy(cp->cp_regs_master, esecuzione->contesto, sizeof(esecuzione->contesto));
        cp->cp_regs_master[I_RSP] = pila[3];
```

```
        cp->cp_regs_master[N_REG] = pila[0];
        for (vaddr src =  fin_utn_p - DIM_USR_STACK, dst = ini_utn_p;
               src != fin_utn_p;
               src += DIM_PAGINA, dst += DIM_PAGINA)
        {
            paddr psrc = trasforma(esecuzione->cr3, src);
            paddr pdst = trasforma(master->cr3, dst);
            memcpy(voidptr_cast(pdst), voidptr_cast(psrc), DIM_PAGINA);
        }
        cp->master_waiting = false;
        inspronti();
        inserimento_lista(pronti, master);
    } else {
        cp->slave_waiting = true;
    }
    schedulatore();
}
```