Disassemble

Name Mangling

Per via dell'overloading nei file oggetto del codice in C++, le label delle funzioni assumono identificatori diversi da quelli assegnati dal programmatore. Questi identificatori sono composti sia dall'identificatore scelto dal programmatore che da eventuali classi di cui sono membri, che dai parametri in ingresso alla funzione. Questa pratica si chiama **Mangling**.

Per verificare la correttezza, il comando `c++filt` traduce i nomi da basso livello ad alto livello:

```
$ c++filt _Z7scambiaRiRi
scambia(int &, int& )
```

In generale, le funzioni semplici in C++ seguono questa forma:

`_Z<K><nome><tipo_parametri>`

Per esempio:

`int somma(int a, int b)` diventa `_Z5sommaii` - `void stampa(char c)` diventa `_Z6stampac`

Corrispondenza dei tipi

I tipi built-in seguono la seguente corrispondenza:

Assembly	C++
v	void
w	wchar_t
b	bool
c	char
h	unsigned char
s	short
t	unsigned short
i	int
j	unsigned int
l	long
m	unsigned long
f	float
d	double

Corrispondenza tipi utente

Per i tipi utente la corrispondenza funziona come segue:

`<N><NOME>`

Dove `N` è il numero di caratteri di `<NOME>` e `<NOME>` è il nome del tipo. Ad esempio:

- `5Punto`
- `2st`
- `9struttura`
- `4elem`

Tipi Composti

In caso di riferimenti, puntatori o attributi `const`, si aggiungono le seguenti corrispondenze:

Tipo	Puntatore	Riferimento	Costante
Corrispon.	P	R	K

Nomi ripetuti

Se un tipo utente viene riferito più volte (ad esempio `somma(st, st)`), anziché riscriverlo si usa:

- `S_` per riferirsi al primo tipo utente incontrato nella stringa.
- `S0_` per riferirsi al secondo.
- `S1_` al terzo e così via.

Esempi:

- `somma(st, st)` diventa `_Z5somma2stS_`
- `somma(punto, punto, st, st*, punto*)` diventa `_Z5somma5puntoS_2stPS0_PS_`

Classi

Funzioni membro Nelle funzioni membro viene sempre passato il puntatore `this` come primo argomento implicito, quindi per quanto riguarda le funzioni membro delle classi questo si troverà sempre nel registro `RDI`.
Sia `M` il numero di caratteri del nome della classe e `K` il numero di caratteri del nome della funzione membro:

`_ZN<M><NOME_CLASSE>K<NOME_FUNZIONE_MEMBRO>E<STRINGA_DI_PARAMETRI>`

Costruttore Il costruttore è indicato con `C1` e viene inserito dopo il nome della classe prima del carattere `E`:

`_ZN<M><NOME_CLASSE>C1E<STRINGA_DI_PARAMETRI>`

- Al costruttore dovrà essere passato l'indirizzo nel quale questo deve costruire la nuova classe come primo parametro, che per tutti i metodi membri di classi sarà implicito (puntatore `this`)

Distruttore Il distruttore è indicato con `D1`:

`_ZN<M><NOME_CLASSE>D1E<STRINGA_DI_PARAMETRI>`

Gestione dello stack

Parametri di una funzione

Una funzione chiamata riceve gli argomenti in ordine, nei registri:

```
RDI, RSI, RDX, RCX, R8, R9
32-bit (movl): EDI, ESI, EDX
16-bit (movw): DI, SI, DX
8-bit (movb): DIL, SIL, DL
```

Il chiamato deve copiare i parametri attuali in pila per liberare i registri, utilizzando lo spazio noto come **frame** o **record di attivazione**. Ogni funzione porrà il suo valore di ritorno (se singolo) nel registro `RAX`.

I registri `RSP, RBP, RBX, R12-R15` devono essere preservati dal chiamante e ripristinati se utilizzati.

Prologo

La costruzione del **record di attivazione** è detta **prologo** e consiste nei seguenti comandi:

1. `pushq %rbp`: salva in pila il contenuto di `RBP`.
2. `movq %rsp, %rbp`: utilizza `RBP` come puntatore per i parametri attuali.
3. `subq <k>, %rsp`: alloca lo spazio necessario (multiplo di 8) per i parametri attuali.
4. Inizializzazione dello spazio in pila in modo **allineato** (multipli di 8).

Epilogo

Per ripristinare `RSP` e `RBP`, si utilizza l'**epilogo**:

```
movq %rbp, %rsp
popq %rbp
ret
```

Equivalente a:

```
leave
ret
```

Return Value

Sia `ret_type` il tipo della variabile ritornata dalla funzione in esame. I casi possibili per il valore di ritorno sono:

- `sizeof(ret_type) <= 8`: Il valore è contenuto in `RAX`.
- `8 < sizeof(ret_type) <= 16`: La parte bassa è in `RAX` e la parte alta in `RDI`.
- `sizeof(ret_type) > 16`: Il valore è allocato nello stack del chiamante e il puntatore è in `RAX`.

Esercizio 1 senza costruttore

```
// file cc.h
#include <iostream>
using namespace std;
struct st {
    char vv1[4];
    long vv2[4];
};
class cl {
    st s;
public:
    cl(char v[]);
    void elab1(int d, st& ss);
};

//file es1.cpp
#include "cc.h"
void cl::elab1(int d, st& ss)
{
    for (int i = 0; i < 4; i++) {
        if (d <= ss.vv2[i])
            s.vv1[i] -= ss.vv1[i];
        s.vv2[i] = d + i;
    }
}
```

```
//es1.s
.set this, -8
.set d, -16
.set ss, -24
.set i, -28

.set s_vv1, 0
.set s_vv2, 8
```

```
.global _ZN2cl5elab1EiR2st
_ZN2cl5elab1EiR2st:

    pushq %rbp
    movq %rsp, %rbp
    subq $24, %rsp

    movq %rdi, this(%rbp)
    movl %esi, d(%rbp)
    movq %rdx, ss(%rbp)

    movl $0, i(%rbp)

ciclo_for:
    movslq i(%rbp), %rcx
    cmpq $4, %rcx
    jge fine_for

    movq ss(%rbp), %rsi
    movq this(%rbp), %rdi

    movq s_vv2(%rsi, %rcx, 8), %rax
    movslq d(%rbp), %rdx
    cmpq %rdx, %rax
    jl dopo_if

    movb s_vv1(%rdi, %rcx, 1), %al
    subb s_vv1(%rsi, %rcx, 1), %al
    movb %al, s_vv1(%rdi, %rcx, 1)


dopo_if:
    movslq d(%rbp), %rax
    movslq i(%rbp), %rdx
    addq %rdx, %rax
    movq %rax, s_vv2(%rdi, %rcx, 8)


    incl i(%rbp)
    jmp ciclo_for

fine_for:
    leave
    ret
```

Esercizio 1 con costruttore

```
// la struct era stata passata per valore ma in realtà
// verrà passata per puntatore
.set this, -8
.set s1, -16
.set cla, -56
.set i, -64

.global _ZN2cl5elab1E3st1
_ZN2cl5elab1E3st1:

    pushq %rbp
    movq %rsp, %rbp
    subq $64,%rsp

    movq %rdi, this(%rbp)
    movq %rsi, s1(%rbp)

    // procedura inizializzazione
    leaq cla(%rbp), %rdi
    mov $'k', %rsi
    leaq s1(%rbp), %rdx # senza riferimento serve leaq

    call _ZN2clC1EcR3st1

    movl $0, i(%rbp)

ciclo_for:
    movslq i(%rbp), %rcx
    cmpq $4, %rcx
    jge fine_for

    movq this(%rbp), %rdi
    leaq s1(%rbp), %rsi # senza riferimento serve leaq

    movb (%rdi, %rcx, 1), %al
    cmpb (%rsi, %rcx, 1), %al
    jg dopo_if

    # cla.s.vc[i]
    leaq cla(%rbp), %rsi
    movb (%rsi, %rcx, 1), %al
    movb %al, (%rdi, %rcx, 1)

    movq 8(%rsi, %rcx, 8), %rax
```

```
    movq %rcx, %rdx
    subq %rax, %rdx
    movq %rdx, 8(%rdi, %rcx, 8)

dopo_if:
    incl i(%rbp)
    jmp ciclo_for


fine_for:
    leave
    ret
```

Nucleo

5 quadword in pila nelle int

`stack = trasforma(p->cr3, fin_utn_p - DIM_USR_STACK);`
- `stack[-5]` = int_cast(f): `rip (instruction pointer) //pila[0]`
-stack[-4]= `SEL_CODICE_SISTEMA` oppure `SEL_CODICE_UTENTE`: cs cpl, privilegio //pila[1]
- `stack[-3]`: if (flags) da inizializzare con I bit da attivare in or fisico tra loro `|` //pila[2]
- `stack[-2]` = fin_utn_p - sizeof(natq): `srsp (stack) //pila[3]`
-stack[-1]= `SEL_DATI_UTENTE`: (segmentazione) //pila[4]

Conversioni

- da `void*` a `vaddr`:
  `int_cast<vaddr>()`
- da `vaddr` a `void*`
  `voidptr_cast(v)`
- da intero a puntatore non void
  `ptr_cast<t1, t2>(uint_var) t1: tipo degli oggetti puntati t2: tipo intero`

Prova Memoria Virtuale

Testo prova:
Aggiungiamo una nuova zona "utente/cow" alla memoria virtuale dei nuovi processi utente. Il contenuto di questa zona e inizializzato all'avvio del sistema e ogni processo ne possiede una copia privata. Invece di copiare l'intera zona ogni volta che creiamo un nuovo processo, adottiamo la tecnica del copy on write (abbreviato in cow): tutti i processi condividono inizialmente la stessa zona, ma in solo lettura, e copiamo le singole pagine se e solo se un processo tenta di scrivervi

```
/*****************************************
* sistema/sistema.cpp
*****************************************/


// ( SOLUZIONE 2023-06-07


    cow_root = alloca_tab();
    if (!cow_root)
        return false;
    vaddr v = map(cow_root, ini_utn_w, ini_utn_w + DIM_USR_COW, BIT_US,
            [](vaddr v) {
                paddr f = alloca_frame();
                memset(voidptr_cast(f), 0, DIM_PAGINA);
                return f;
            });
    if (v != ini_utn_w + DIM_USR_COW) {
        unmap(cow_root, ini_utn_w, v,
            [](vaddr, paddr p, int) {
                rilascia_frame(p);
            });
        return false;
    }
    return true;
//   SOLUZIONE 2023-06-07 )
// ( SOLUZIONE 2023-06-07

    if (esecuzione->livello != LIV_UTENTE || v < ini_utn_w || v >= ini_utn_w + DIM_USR_COW)
        return false;

    vaddr b = base(v, 0);
    for (tab_iter it(esecuzione->cr3, b, DIM_PAGINA); it; it.next()) {
        tab_entry& e = it.get_e();
        if (!(e & BIT_P))
            fpanic("indirizzo cow %lx non mappato", b);
        if (e & BIT_RW)
            continue;
        paddr new_frame;
        paddr old_frame = extr_IND_FISICO(e);
        if (it.get_l() > 1) {
            new_frame = alloca_tab();
            if (!new_frame)
                return false;
            copy_des(old_frame, new_frame, 0, 512);
        } else {
```

```
            new_frame = alloca_frame();
            if (!new_frame)
                return false;
            memcpy(reinterpret_cast<void*>(new_frame),
                   reinterpret_cast<void*>(old_frame), DIM_PAGINA);
            invalida_entrata_TLB(it.get_v());
        }
        set_IND_FISICO(e, new_frame);
        e |= BIT_RW;
    }
    return true;
//   SOLUZIONE 2023-06-07 )
```

Prova I/O

Testo Prova:
Il modulo I/O del nucleo contiene le primitive readhd n() e writehd n() che permettono di leggere o scrivere blocchi dell'hard disk. Vogliamo velocizzare le due primitive introducendo una buffer cache che mantenga in memoria i blocchi letti piu di recente, in modo che eventuali letture di blocchi che si trovino nella buffer cache possano essere realizzate con una semplice copia da memoria a memoria, invece che con una costosa operazione di I/O. Per quanto riguarda le scritture adottiamo una politica write-back/write-allocate. Per il rimpiazzamento adottiamo la politica LRU (Least Recently Used): se la cache e piena e deve essere letto un blocco non in cache, si rimpiazza il blocco a cui non si accede da piu tempo (nota: per acesso ad un blocco si intende una qualunque readhd n() o writehd n() che lo ha coinvolto).

```
/*****************************************
 io/io.cpp
*****************************************/
// campo aggiunto a buf_des:
// ( SOLUZIONE 2023-02-01

    bool dirty;

//   SOLUZIONE 2023-02-01 )

// ( SOLUZIONE 2023-02-01

    extern "C" void c_readhd_n(natb vetti[], natl primo, natb quanti)
    {
        des_ata* d = &hard_disk;

        if (!access(vetti, quanti * DIM_BLOCK, true)) {
            flog(LOG_WARN, "readhd_n: parametri non validi: %p, %d", vetti, quanti);
```

```
            abort_p();
        }

        if (!quanti)
            return;

        // primitiva read prima dell'aggiunta di bufcache:
        //sem_wait(d->mutex);
        //starthd_in(d, vetti, primo, quanti);
        //sem_wait(d->sincr);
        //sem_signal(d->mutex);


        sem_wait(d->mutex);
        for (natl i = 0; i < quanti; i++) {
            // cerchiamo il blocco nella buffercache. Se non lo
            // troviamo rimpiazziamo l'lru
            buf_des *b = bufcache_search(primo + i);
            if (!b) {
                b = &d->bufcache[d->lru];
                if (b->dirty) {
                    starthd_out(d, b->buf, b->block, 1);
                    sem_wait(d->sincr);
                }
                starthd_in(d, b->buf, primo + i, 1);
                sem_wait(d->sincr);
                b->block = primo + i;
                b->full = true;
                b->dirty = false;
            }
            memcpy(vetti + i * DIM_BLOCK, b->buf, DIM_BLOCK);
            // ora b è l'mru
            bufcache_promote(b);
        }
        sem_signal(d->mutex);
    }

//   SOLUZIONE 2023-02-01 )


// ( SOLUZIONE 2023-02-01

        extern "C" void c_writehd_n(natb vetto[], natl primo, natb quanti)
        {
            des_ata* d = &hard_disk;
```

```
            if (!access(vetto, quanti * DIM_BLOCK, false)) {
                flog(LOG_WARN, "writehd_n: parametri non validi: %p, %d", vetto, quanti);
                abort_p();
            }

            if (!quanti)
                return;

        // primitiva write prima dell'aggiunta di bufcache:
        //sem_wait(d->mutex);
        //starthd_out(d, vetto, primo, quanti);
        //sem_wait(d->sincr);
        //sem_signal(d->mutex);

        sem_wait(d->mutex);
        for (natl i = 0; i < quanti; i++) {
            // politica write-back: scriviamo solo in buffercache
            // non c'è bisogno di caricare il blocco dall'hard disk,
            // in quanto dobbiamo sovrascriverlo interamente.
            buf_des *b = bufcache_search(primo + i);
            if (!b) {
                b = &d->bufcache[d->lru];
                if (b->dirty) {
                    starthd_out(d, b->buf, b->block, 1);
                    sem_wait(d->sincr);
                }
                b->block = primo + i;
                b->full = true;
            }
            memcpy(b->buf, vetto + i * DIM_BLOCK, DIM_BLOCK);
            b->dirty = true;
            // c'è stato un accesso al buffer e dobbiamo promuoverlo
            bufcache_promote(b);
        }
        sem_signal(d->mutex);
    }

//   SOLUZIONE 2023-02-01 )
// ( SOLUZIONE 2023-02-01

extern "C" void c_synchd()
{
    des_ata *d = &hard_disk;

    sem_wait(d->mutex);
    for (int i = 0; i < MAX_BUF_DES; i++) {
```

```
        buf_des *b = &d->bufcache[i];
        if (b->dirty) {
            starthd_out(d, b->buf, b->block, 1);
            sem_wait(d->sincr);
            b->dirty = false;
        }
    }
    sem_signal(d->mutex);
}
//   SOLUZIONE 2023-02-01 )


/****************************************
* io/io.s
****************************************/

// ( SOLUZIONE 2023-02-01

        fill_io_gate    IO_TIPO_SHD a_synchd

//   SOLUZIONE 2023-02-01 )
// ( SOLUZIONE 2023-02-01

        .extern c_synchd
    a_synchd:
        .cfi_startproc
        .cfi_def_cfa_offset 40
        .cfi_offset rip, -40
        .cfi_offset rsp, -16
        call c_synchd
        iretq
        .cfi_endproc

//   SOLUZIONE 2023-02-01 )
```

Prova breakpoint

Testo prova: Vogliamo fornire ai processi la possibilit'a di scoprire se l'esecuzione di un altro processo passa da una certa istruzione. Per far questo forniamo una primitiva breakpoint(vaddr rip) che installa un breakpoint (istruzione int3, codice operativo 0xCC) all'indirizzo rip, quindi blocca il processo chiamante, sia P1. Quando (e se) un altro processo P2 arriva a quell'indirizzo, il processo P1 deve essere risvegliato. Si noti che il processo P2 non si blocca e deve proseguire la sua esecuzione indisturbato (salvo che potrebbe dover cedere il processore a P1 per via della preemption)

```
/****************************************
* sistema/sistema.s
****************************************/

// ( SOLUZIONE 2019-06-12
    movq $3, %rdi
    movq $0, %rsi
    movq %rsp, %rdx
    call c_breakpoint_exception
//   SOLUZIONE 2019-06-12 )


/****************************************
* sistema/sistema.cpp
****************************************/

// ( SOLUZIONE 2019-06-12
struct b_info {
    des_proc* waiting;
    natq rip;
    natb orig;
} b_info;

extern "C" void c_breakpoint(vaddr rip)
{
    if (b_info.waiting) {
        esecuzione->contesto[I_RAX] = 0xFFFFFFFF;
        return;
    }

    if (rip < ini_utn_c || rip >= fin_utn_c) {
        flog(LOG_WARN, "rip %lx out of bounds [%lx, %lx)", rip, ini_utn_p, fin_utn_p);
        c_abort_p();
        return;
    }

    natb *bytes = ptr_cast<natb>(trasforma(esecuzione->cr3, rip));
    b_info.rip = rip;
    b_info.orig = *bytes;
    *bytes = 0xCC;
    b_info.waiting = esecuzione;
    schedulatore();
}


extern "C" void c_breakpoint_exception(int tipo, natq errore, vaddr* p_saved_rip)
```

```
{
    if (!b_info.waiting || *p_saved_rip != b_info.rip + 1) {
        gestore_eccezioni(tipo, errore, *p_saved_rip);
        return;
    }
    natb *bytes = ptr_cast<natb>(trasforma(esecuzione->cr3, b_info.rip));
    *bytes = b_info.orig;
    (*p_saved_rip)--;
    b_info.waiting->contesto[I_RAX] = esecuzione->id;
    inspronti();
    inserimento_lista(pronti, b_info.waiting);
    b_info.waiting = 0;
    schedulatore();
}
//   SOLUZIONE 2019-06-12 )
```

Prova cacheHDD

Testo prova: Vogliamo aggiungere al nucleo delle primitive che ci permettano di leggere e scrivere dall'hard disk come se questo fosse un unico array di byte, invece che di blocchi. In particolare, il blocco 0 conterra `i byte 1 nell'intervallo [0, 512), il blocco 1 i byte nell'intervallo [512, 1024), e cosı` via. Supponiamo che l'utente chieda di leggere n byte a partire dal byte b. Le primitive dovranno internamente accedere ai blocchi che contengono tutto l'intervallo [b, b + n) e poi restituire o modificare solo i byte richiesti dall'utente. Per rendere efficienti queste operazioni le nuove primitive usano una buffer cache che mantiene in memoria i blocchi acceduti piu `di recente, in modo che eventuali letture di byte contenuti in blocchi che si trovino nella buffer cache possano essere realizzate con una semplice copia da memoria a memoria, senza ulteriori operazioni di I/O. Per quanto riguarda le scritture adottiamo una politica write-back/write-allocate. Per il rimpiazzamento adottiamo la politica LRU (Least Recently Used): se la cache` e piena e deve essere caricato un blocco non in cache, si rimpiazza il blocco a cui non si accede da pi‘u tempo.

```
/*****************************************
* io/io.cpp
*****************************************/

// ( SOLUZIONE 2023-02-15
extern "C" void c_bufread_n(natb *vetti, natl first_byte, natl nbytes)
{
    if (!access(vetti, nbytes, true)) {
        flog(LOG_WARN, "bufread_n: parametri non validi: %p, %d", vetti, nbytes);
        abort_p();
```

```
    }

    des_ata *d = &hard_disk;

    natl block = first_byte / DIM_BLOCK;
    natl offset = first_byte % DIM_BLOCK;
    natl toread = DIM_BLOCK - offset < nbytes ? DIM_BLOCK - offset : nbytes;
    sem_wait(d->mutex);
    // meccanismo di copiatura buffer
    while (nbytes > 0) {
        buf_des *b = bufcache_search(block);
        if (!b) {
            b = &d->bufcache[d->lru];
            if (b->dirty) {
                starthd_out(d, b->buf, b->block, 1);
                sem_wait(d->sincr);
            }
            starthd_in(d, b->buf, block, 1);
            sem_wait(d->sincr);
            b->block = block;
            b->full = true;
            b->dirty = false;
        }
        memcpy(vetti, b->buf + offset, toread);
        bufcache_promote(b);
        vetti += toread;
        nbytes -= toread;
        offset = 0;
        toread = DIM_BLOCK < nbytes ? DIM_BLOCK : nbytes;
        block++;
    }
    sem_signal(d->mutex);
}

extern "C" void c_bufwrite_n(natb *vetto, natq first_byte, natq nbytes)
{
    if (!access(vetto, nbytes, false)) {
        flog(LOG_WARN, "bufwrite_n: parametri non validi: %p, %lu", vetto, nbytes);
        abort_p();
    }

    des_ata *d = &hard_disk;

    natl block = first_byte / DIM_BLOCK;
    natl offset = first_byte % DIM_BLOCK;
    natl towrite = DIM_BLOCK - offset < nbytes ? DIM_BLOCK - offset : nbytes;
```

```
    sem_wait(d->mutex);
    while (nbytes > 0) {
        buf_des *b = bufcache_search(block);
        if (!b) {
            b = &d->bufcache[d->lru];
            if (b->dirty) {
                starthd_out(d, b->buf, b->block, 1);
                sem_wait(d->sincr);
            }
            if (towrite < DIM_BLOCK) {
                starthd_in(d, b->buf, block, 1);
                sem_wait(d->sincr);
            }
            b->block = block;
            b->full = true;
        }
        memcpy(b->buf + offset, vetto, towrite);
        b->dirty = true;
        bufcache_promote(b);
        vetto += towrite;
        nbytes -= towrite;
        offset = 0;
        towrite = DIM_BLOCK < nbytes ? DIM_BLOCK : nbytes;
        block++;
    }
    sem_signal(d->mutex);
}
//   SOLUZIONE 2023-02-15 )


****************************************
* io/io.s
****************************************

// ( SOLUZIONE 2023-02-15
    fill_io_gate    IO_TIPO_BRD a_bufread_n
    fill_io_gate    IO_TIPO_BWR a_bufwrite_n
//   SOLUZIONE 2023-02-15 )
// ( SOLUZIONE 2023-02-15
    .extern c_bufread_n
a_bufread_n:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_bufread_n
```

```
    iretq
    .cfi_endproc

    .extern c_bufwrite_n
a_bufwrite_n:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_bufwrite_n
    iretq
    .cfi_endproc
//   SOLUZIONE 2023-02-15 )
```

Prova I/O generica con registri

Testo prova:
Colleghiamo al sistema delle periferiche PCI di tipo ce, con vendorID 0xedce e deviceID 0x1234. Ogni periferica ce usa 16 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione BAR0, sia b. Le periferiche ce sono semplici periferiche con un registro RBR di ingresso, tramite il quale 'e possibile leggere il prossimo byte disponibile. Se abilitata scrivendo 1 nel registro CTL, la perifica invia una richiesta di interruzione quando il registro RBR contiene un nuovo byte. La periferica non invia nuove richieste di interruzione fino a quando il registro RBR non viene letto.

```
/*****************************************
* io/io.cpp
*****************************************/

// ( SOLUZIONE 2023-07-19
void vce_buf_read(vce_buf *vb, char *vetti)
{
    sem_wait(vb->mutex);
    if (vb->n > 0) {
        *vetti = vb->buf[vb->head];
        vb->head = (vb->head + 1) % VCE_BUFSIZE;
        vb->n--;
    }
    sem_signal(vb->mutex);
}
//   SOLUZIONE 2023-07-19 )
// ( SOLUZIONE 2023-07-19
extern "C" void c_vceread_n(natl vn, char *vetti, natq quanti)
{
    if (vn >= VCE_NUM) {
```

```
        flog(LOG_WARN, "vce %d non valido", vn);
        abort_p();
    }

    if (!access(vetti, quanti, true, false)) {
        flog(LOG_WARN, "parametri non validi: %p, %lu", vetti, quanti);
        abort_p();
    }

    vce_des *v = &ce.vces[vn];
    sem_wait(v->mutex);
    for (natl i = 0; i < quanti; i++) {
        sem_wait(v->sync);
        vce_buf_read(&v->buf, vetti);
        vetti++;
    }
    sem_signal(v->mutex);
}

extern "C" void estern_ce(natq)
{
    for (;;) {
        char c = inputb(ce.iRBR);
        sem_wait(ce.mutex);
        vce_des *v = &ce.vces[ce.active];
        sem_signal(ce.mutex);
        if (vce_buf_write(&v->buf, c))
            sem_signal(v->sync);
        wfi();
    }
}
//   SOLUZIONE 2023-07-19 )
// ( SOLUZIONE 2023-07-19
        for (natq i = 0; i < VCE_NUM; i++) {
            vce_des *v = &ce.vces[i];
            v->mutex = sem_ini(1);
            if (v->mutex == 0xFFFFFFFF) {
                flog(LOG_ERR, "semafori insufficienti");
                return false;
            }
            v->sync = sem_ini(0);
            if (v->sync == 0xFFFFFFFF) {
                flog(LOG_ERR, "semafori insufficienti");
                return false;
            }
            v->buf.mutex = sem_ini(1);
```

```
            if (v->buf.mutex == 0xFFFFFFFF) {
                flog(LOG_ERR, "semafori insufficienti");
                return false;
            }
            v->buf.head = v->buf.tail = v->buf.n = 0;
        }
        if (activate_pe(estern_ce, 0, MIN_EXT_PRIO + INTR_TIPO_CE, LIV_SISTEMA, irq) == 0xFF
            flog(LOG_ERR, "impossibile attivare processo esterno");
            return false;
        }
        outputb(1, ce.iCTL);
//   SOLUZIONE 2023-07-19 )
```

Prova con manipolazione della pila

Testo prova: Aggiungiamo al nucleo il meccanismo dei checkpoint. Il checkpoint di un processo e 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 e vuota). 1 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 verra 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 dovra sincronizzare con il master per trasferigli un nuovo checkpoint. Il master si puo mettere in attesa del prossimo checkpoint tramite la primitiva cp_get(regs), dove regse 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 pu'o essere al piu master o slave di un altro processo.

```
/*****************************************
* sistema/sistema.cpp
*****************************************/

// ( SOLUZIONE 2024-06-28
    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();
//   SOLUZIONE 2024-06-28 )
// ( SOLUZIONE 2024-06-28
    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();
//   SOLUZIONE 2024-06-28 )
```

```
## funzioni liste:
    // Nodo della lista
    struct Node {
        int data;
        Node* next;
    };

    // Funzione per creare un nuovo nodo
    Node* createNode(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = nullptr;
        return newNode;
    }

    // Funzione per estrarre uno specifico elemento
    bool extractElement(Node*& head, int value) {
        if (!head) return false; // Lista vuota
```

```
    // Caso in cui l'elemento da rimuovere è il primo
    if (head->data == value) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return true;
    }

    // Ricerca dell'elemento da rimuovere
    Node* current = head;
    while (current->next && current->next->data != value) {
        current = current->next;
    }

    if (current->next) {
        Node* temp = current->next;
        current->next = temp->next;
        delete temp;
        return true;
    }

    return false; // Elemento non trovato
}

// Funzione per inserire un elemento in modo ordinato
void insertOrdered(Node*& head, int value) {
    Node* newNode = createNode(value);

    // Caso in cui la lista è vuota o il nuovo elemento è minore del primo elemento
    if (!head || value < head->data) {
        newNode->next = head;
        head = newNode;
        return;
    }

    // Inserimento nel posto corretto
    Node* current = head;
    while (current->next && current->next->data < value) {
        current = current->next;
    }

    newNode->next = current->next;
    current->next = newNode;
}

// Funzione per estrarre l'ultimo elemento della lista
```

```cpp
bool extractTail(Node*& head) {
    if (!head) return false; // Lista vuota

    // Caso in cui la lista ha un solo elemento
    if (!head->next) {
        delete head;
        head = nullptr;
        return true;
    }

    // Ricerca del penultimo nodo
    Node* current = head;
    while (current->next->next) {
        current = current->next;
    }

    delete current->next;
    current->next = nullptr;
    return true;
}

// Funzione per stampare la lista
void printList(Node* head) {
    Node* current = head;
    while (current) {
        std::cout << current->data << " -> ";
        current = current->next;
    }
    std::cout << "null\n";
}

// Funzione principale per testare le operazioni
int main() {
    Node* head = nullptr;

    // Inserimento ordinato
    insertOrdered(head, 10);
    insertOrdered(head, 5);
    insertOrdered(head, 15);
    insertOrdered(head, 12);
    std::cout << "Lista dopo inserimenti ordinati:\n";
    printList(head);

    // Estrazione di un elemento specifico
    std::cout << "\nEstraggo elemento 10:\n";
    if (extractElement(head, 10)) {
```

```
        std::cout << "Elemento 10 rimosso.\n";
    } else {
        std::cout << "Elemento 10 non trovato.\n";
    }
    printList(head);

    // Estrazione in coda
    std::cout << "\nEstraggo l'ultimo elemento:\n";
    if (extractTail(head)) {
        std::cout << "Ultimo elemento rimosso.\n";
    } else {
        std::cout << "Lista vuota, niente da rimuovere.\n";
    }
    printList(head);

    return 0;
}
```

Dispensa scritta da: Pietro Balestri e Matteo Spallazzi