

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

5 giugno 2010

1. Un **mutex** è un tipo particolare di semaforo che può assumere soltanto due stati: libero oppure occupato. Inoltre, un mutex ricorda l'identità del processo che lo ha occupato. È un errore se lo stesso processo tenta di occupare nuovamente il mutex prima di averlo liberato. Inoltre, è un errore se il mutex viene liberato da un processo diverso da quello che lo aveva occupato.

Per realizzare un mutex definiamo la seguente struttura (file **sistema.cpp**):

```
struct des_mutex {  
    natl owner;  
    des_proc* waiting;  
};
```

Se il mutex è occupato, il campo **owner** contiene l'id del processo che lo ha occupato, altrimenti **owner** vale 0. Il campo **waiting** serve a realizzare una lista di processi in attesa di acquisire il **mutex**.

Le seguenti primitive, accessibili dal livello utente, operano sui **mutex**:

- **natl mutex_ini()** (già realizzata): inizializza un nuovo mutex, con i campi **owner** e **waiting** entrambi a 0, e ne restituisce l'identificatore. Se non è possibile creare un nuovo mutex restituisce 0xFFFFFFFF.
- **void mutex_wait(natl mux)**: tenta di occupare il mutex di identificatore **mux**. Se il mutex è già occupato sospende il processo in attesa che il mutex venga prima liberato. Abortisce il processo in caso di errore.
- **void mutex_signal(natl mux)**: libera il mutex di identificatore **mux**. Se qualche altro processo era in attesa, lo risveglia e gli cede il mutex (che in questo caso resta occupato). Gestisce una eventuale *preemption*. Abortisce il processo in caso di errore.

Modificare i file **sistema.cpp** e **sistema.s** in modo da realizzare le primitive mancanti.

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

```
// ( SOLUZIONE 2010-06-05 #1
carica_gateTIPO_MW           a_mutex_wait      LIV_UTENTE
carica_gateTIPO_MS           a_mutex_signal    LIV_UTENTE
//   SOLUZIONE 2010-06-05 )
// ( SOLUZIONE 2010-06-05
a_mutex_wait:
.cfi_startproc
.cfi_def_cfa_offset 40
.cfi_offset rip, -40
.cfi_offset rsp, -16
call salva_stato
call c_mutex_wait
call carica_stato
iretq
.cfi_endproc

a_mutex_signal:
.cfi_startproc
.cfi_def_cfa_offset 40
.cfi_offset rip, -40
.cfi_offset rsp, -16
call salva_stato
call c_mutex_signal
call carica_stato
iretq
.cfi_endproc
//   SOLUZIONE 2010-06-05 )
```

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

```
// ( SOLUZIONE 2010-06-05

bool mutex_valido(natl mux);
extern "C" void c_mutex_wait(natl mux)
{
    des_mutex *m;

    if (!mutex_valido(mux)) {
        flog(LOG_WARN, "mutex errato: %d", mux);
        c_abort_p();
        return;
    }

    m = &array_desm[mux];

    if (m->owner == esecuzione->id) {
        flog(LOG_WARN, "mutex_wait ricorsiva");
        c_abort_p();
        return;
    }

    if (m->owner == 0) {
        m->owner = esecuzione->id;
```

```

    } else {
        inserimento_lista(m->waiting, esecuzione);
        schedulatore();
    }
}

extern "C" void c_mutex_signal(natl mux)
{
    des_mutex *m;

    if (!mutex_valido(mux)) {
        flog(LOG_WARN, "mutex errato: %d", mux);
        c_abort_p();
        return;
    }

    m = &array_desm[mux];

    if (m->owner != esecuzione->id) {
        flog(LOG_WARN, "mutex_signal su mutex errato");
        c_abort_p();
        return;
    }

    if (m->waiting != 0) {
        des_proc *lavoro = rimozione_lista(m->waiting);
        m->owner = lavoro->id;
        // possibile preemption
        inspronti();
        inserimento_lista(pronti, lavoro);
        schedulatore();
    } else {
        m->owner = 0;
    }
}

```

// SOLUZIONE 2010-06-05)

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

26 giugno 2010

1. Definiamo un *monitor* come un oggetto che può essere posseduto da un solo processo alla volta e a cui è associata una *variabile di condizione*. Il possessore del monitor può eseguire le operazioni di *attesa* e *notifica* sulla variabile di condizione associata. L'operazione di attesa rilascia il monitor, sospende il processo fino a quando un altro processo esegue l'operazione di notifica sulla stessa variabile di condizione, quindi riacquisisce il monitor. L'operazione di notifica risveglia uno dei processi in attesa sulla variabile di condizione, se ve ne sono, altrimenti non fa niente.

Per realizzare i monitor definiamo la seguente struttura (file `sistema.cpp`):

```
struct des_monitor {  
    natl owner;  
    natl mutex;  
    natl cond;  
    int num_waiting;  
};
```

Il campo `owner` contiene l'id del processo che possiede il monitor (0 se nessun processo possiede attualmente il monitor). I campi `mutex` e `cond` sono due indici di semafori. Il campo `num_waiting` tiene conto del numero di processi in attesa sulla variabile di condizione.

Le seguenti primitive, accessibili dal livello utente, operano sui monitor (nei casi di errore, abortiscono il processo chiamante):

- `natl monitor_ini()` (già realizzata): inizializza un nuovo monitor, con i campi `owner` e `num_waiting` entrambi a 0, `mutex` con l'indice di un semaforo di valore iniziale 1, `cond` con l'indice di un semaforo di valore iniziale 0. Restituisce l'identificatore del nuovo monitor. Se non è possibile creare un nuovo monitor, restituisce 0xFFFFFFFF.
- `void monitor_enter(natl mon)` (già realizzata): tenta di impadronirsi del monitor di identificatore `mon`. Se il monitor appartiene già a qualche altro processo, sospende il processo in attesa che il monitor venga prima rilasciato. È un errore tentare di impadronirsi di un monitor che si possiede già.
- `void monitor_leave(natl mon)` (già realizzata): rilascia il monitor di identificatore `mon`. È un errore tentare di rilasciare un monitor che non si possiede.
- `void monitor_wait(natl mon)`: si pone in attesa sulla variabile di condizione associata al monitor di identificatore `mon`. È un errore tentare di porsi in attesa su una variabile di condizione associata ad un monitor che non si possiede.
- `void monitor_notify(natl mon)`: esegue una notifica sulla variabile di condizione associata al monitor di identificatore `mon`. È un errore tentare di eseguire una notifica su una variabile di condizione associata ad un monitor che non si possiede.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti.

```

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


// ( SOLUZIONE 2010-06-26
    carica_gateTIPO_MW          a_monitor_wait    LIV_UTENTE
    carica_gateTIPO_MS          a_monitor_notify LIV_UTENTE
// SOLUZIONE 2010-06-26 )
// ( SOLUZIONE 2010-06-26

a_monitor_wait:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_monitor_wait
    iretq
    .cfi_endproc

a_monitor_notify:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_monitor_notify
    iretq
    .cfi_endproc
//     SOLUZIONE 2010-06-26 )

*****


* sistema/sistema.cpp
*****


// ( SOLUZIONE 2010-06-26
extern "C" void c_monitor_wait(natl mon)
{
    if (!monitor_valido(mon)) {
        flog(LOG_WARN, "monitor non valido: %d", mon);
        abort_p();
    }

    des_monitor *p_des = &array_desm[mon];

    if (p_des->owner != esecuzione->id) {
        flog(LOG_WARN, "wait errata sul monitor: %d", mon);
        abort_p();
    }

    p_des->num_waiting++;
    p_des->owner = 0;
    sem_signal(p_des->mutex);
    sem_wait(p_des->cond);
    sem_wait(p_des->mutex);
    p_des->owner = esecuzione->id;
}

extern "C" void c_monitor_notify(natl mon)
{
    if (!monitor_valido(mon)) {

```

```
        flog(LOG_WARN, "monitor non valido: %d", mon);
        abort_p();
    }

des_monitor *p_des = &array_desm[mon];

if (p_des->owner != esecuzione->id) {
    flog(LOG_WARN, "notify errata sul monitor: %d", mon);
    abort_p();
}

if (p_des->num_waiting > 0) {
    p_des->num_waiting--;
    sem_signal(p_des->cond);
}
// SOLUZIONE 2010-06-26 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

17 luglio 2010

1. Definiamo un *monitor* come un oggetto in cui può *entrare* un solo processo alla volta, e a cui è associata una *variabile di condizione*. Il processo che si trova all'interno del monitor è detto *possessore* del monitor. Il possessore del monitor può eseguire le operazioni di *attesa* e *notifica* sulla variabile di condizione associata. L'operazione di attesa rilascia il monitor e sospende il processo fino a quando un altro processo esegue l'operazione di notifica sulla stessa variabile di condizione. L'operazione di notifica risveglia uno dei processi in attesa sulla variabile di condizione, se ve ne sono, altrimenti non fa niente. Se un processo ne risveglia un altro tramite una operazione di notifica, gli cede anche il possesso del monitor, quindi si sospende su una coda di processi “urgenti” associata al monitor. I processi i “urgenti” hanno precedenza nel riottenere il possesso del monitor, non appena questo si libera.

Per realizzare i monitor definiamo la seguente struttura (file `sistema.cpp`):

```
struct des_monitor {  
    natl owner;  
    des_proc *w_enter;  
    des_proc *w_cond;  
    des_proc *urgent;  
};
```

Il campo `owner` contiene l'id del processo che possiede il monitor (0 se nessun processo possiede attualmente il monitor). Il campo `w_cond` rappresenta la lista dei processi in attesa sulla variabile di condizione associata al monitor. Il campo `w_enter` rappresenta la lista dei processi in attesa di poter entrare nel monitor. Il campo `urgent` rappresenta la lista dei processi “urgenti”.

Le seguenti primitive, accessibili dal livello utente, operano sui monitor (nei casi di errore, abortiscono il processo chiamante):

- `natl monitor_ini()` (già realizzata): inizializza un nuovo monitor, con tutti i campi a 0. Restituisce l'identificatore del nuovo monitor. Se non è possibile creare un nuovo monitor, restituisce 0xFFFFFFFF.
- `void monitor_enter(natl mon)` (già realizzata): tenta di entrare nel monitor di identificatore `mon`. Se il monitor appartiene già a qualche altro processo, sospende il processo in attesa di potervi entrare. È un errore tentare di entrare in un monitor che si possiede già.
- `void monitor_leave(natl mon)`: esce dal monitor di identificatore `mon`. È un errore tentare di uscire da un monitor che non si possiede. La primitiva deve cedere il possesso del monitor ad uno degli eventuali processi “urgenti” o in attesa di entrare nel monitor, se ve ne sono.
- `void monitor_wait(natl mon)`: si pone in attesa sulla variable di condizione associata al monitor di identificatore `mon`. È un errore tentare di porsi in attesa su una variable di condizione associata ad un monitor che non si possiede. Prima di sospendersi, il processo deve cedere il possesso del monitor ad uno dei processi “urgenti” o in attesa di entrare nel monitor, se ve ne sono.

- `void monitor_notify(natl mon)`(già realizzata): esegue una notifica sulla variabile di condizione associata al monitor di identificatore `mon`. È un errore tentare di eseguire una notifica su una variable di condizione associata ad un monitor che non si possiede.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti.

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

```
// ( SOLUZIONE 2010-07-17
extern "C" void c_monitor_leave(natl mon)
{
    if (!monitor_valido(mon)) {
        flog(LOG_WARN, "monitor non valido: %d", mon);
        c_abort_p();
        return;
    }

    des_monitor *p_des = &array_desm[mon];

    if (p_des->owner != esecuzione->id) {
        flog(LOG_WARN, "monitor_leave errata");
        c_abort_p();
        return;
    }

    des_proc *next = 0;
    if (p_des->urgent != 0)
        next = rimozione_lista(p_des->urgent);
    else if (p_des->w_enter != 0)
        next = rimozione_lista(p_des->w_enter);
    if (next != nullptr) {
        p_des->owner = next->id;
        inspronti();
        inserimento_lista(pronti, next);
        schedulatore();
    } else {
        p_des->owner = 0;
    }
}

extern "C" void c_monitor_wait(natl mon)
{
    if (!monitor_valido(mon)) {
        flog(LOG_WARN, "monitor non valido: %d", mon);
        c_abort_p();
        return;
    }

    des_monitor *p_des = &array_desm[mon];

    if (p_des->owner != esecuzione->id) {
        flog(LOG_WARN, "monitor_wait errata");
        c_abort_p();
        return;
    }

    inserimento_lista(p_des->w_cond, esecuzione);
    des_proc *next = 0;
    if (p_des->urgent != 0)
        next = rimozione_lista(p_des->urgent);
    else if (p_des->w_enter != 0)
        next = rimozione_lista(p_des->w_enter);
    if (next != 0) {
        p_des->owner = next->id;
```

```
    inserimento_lista(pronti, next);
} else {
    p_des->owner = 0;
}
schedulatore();
}

// SOLUZIONE 2010-07-17 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

11 settembre 2010

1. Definiamo un *monitor* come un oggetto in cui può *entrare* un solo processo alla volta, e a cui è associata una *variabile di condizione*. Il processo che si trova all'interno del monitor è detto *possessore* del monitor. Il possessore del monitor può eseguire le operazioni di *attesa* e *notifica a tutti* sulla variabile di condizione associata. L'operazione di attesa fa uscire il processo dal monitor e lo sospende fino a quando un altro processo esegue l'operazione di notifica a tutti sulla stessa variabile di condizione. L'operazione di notifica a tutti risveglia tutti i processi in attesa sulla variabile di condizione, se ve ne sono, altrimenti non fa niente.

Per realizzare i monitor definiamo la seguente struttura (file `sistema.cpp`):

```
struct des_monitor {
    natl owner;
    des_proc *w_enter;
    des_proc *w_cond;
};
```

Il campo `owner` contiene l'id del processo che possiede il monitor (0 se nessun processo possiede attualmente il monitor). Il campo `w_cond` rappresenta la lista dei processi in attesa sulla variabile di condizione associata al monitor. Il campo `w_enter` rappresenta la lista dei processi in attesa di poter entrare nel monitor.

Le seguenti primitive, accessibili dal livello utente, operano sui monitor (nei casi di errore, abortiscono il processo chiamante):

- `natl monitor_ini()` (già realizzata): inizializza un nuovo monitor, con tutti i campi a 0. Restituisce l'identificatore del nuovo monitor. Se non è possibile creare un nuovo monitor, restituisce 0xFFFFFFFF.
- `void monitor_enter(natl mon)` (già realizzata): tenta di entrare nel monitor di identificatore `mon`. Se il monitor appartiene già a qualche altro processo, sospende il processo in attesa di potervi entrare. È un errore tentare di entrare in un monitor che si possiede già.
- `void monitor_leave(natl mon)` (già realizzata): esce dal monitor di identificatore `mon` e lo cede ad uno dei processi in attesa di entrare nel monitor se ve ne sono, altrimenti lo lascia libero. È un errore tentare di uscire da un monitor che non si possiede.
- `void monitor_wait(natl mon)`: esce dal monitor e si pone in attesa sulla variabile di condizione associata al monitor di identificatore `mon`. È un errore tentare di porsi in attesa su una variabile di condizione associata ad un monitor che non si possiede. Prima di sospendersi, il processo deve cedere il possesso del monitor ad uno dei processi in attesa di entrare nel monitor, se ve ne sono.
- `void monitor_notifyAll(natl mon)`: esegue l'operazione di notifica a tutti sulla variabile di condizione associata al monitor di identificatore `mon`. È un errore tentare di eseguire tale operazione su una variabile di condizione associata ad un monitor che non si possiede.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti.

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

```
// ( SOLUZIONE 2010-09-11
    carica_gateTIPO_MW          a_monitor_wait    LIV_UTENTE
    carica_gateTIPO_MS          a_monitor_notifyAll LIV_UTENTE
// SOLUZIONE 2010-09-11 )
// ( SOLUZIONE 2010-09-11

a_monitor_wait:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_monitor_wait
    call carica_stato
    iretq
    .cfi_endproc

a_monitor_notifyAll:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_monitor_notifyAll
    call carica_stato
    iretq
    .cfi_endproc
// SOLUZIONE 2010-09-11 )
```

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

```
// ( SOLUZIONE 2010-09-11

extern "C" void c_monitor_wait(natl mon)
{
    if (!monitor_valido(mon)) {
        flog(LOG_WARN, "monitor non valido: %d", mon);
        c_abort_p();
        return;
    }

    des_monitor *p_des = &array_desm[mon];

    if (p_des->owner != esecuzione->id) {
        flog(LOG_WARN, "monitor_wait errata");
        c_abort_p();
        return;
    }

    inserimento_lista(p_des->w_cond, esecuzione);
    if (p_des->w_enter != 0) {
        des_proc *work = rimozione_lista(p_des->w_enter);
        p_des->owner = work->id;
```

```

        inserimento_lista(pronti, work);
    } else {
        p_des->owner = 0;
    }
    schedulatore();
}

extern "C" void c_monitor_notifyAll(natl mon)
{
    if (!monitor_valido(mon)) {
        flog(LOG_WARN, "monitor non valido: %d", mon);
        c_abort_p();
        return;
    }

    des_monitor *p_des = &array_desm[mon];

    if (p_des->owner != esecuzione->id) {
        flog(LOG_WARN, "monitor_notifyAll errata");
        c_abort_p();
        return;
    }

    inspronti();
    while (p_des->w_cond != 0) {
        des_proc *work = rimozione_lista(p_des->w_cond);
        inserimento_lista(pronti, work);
    }
    schedulatore();
}

// SOLUZIONE 2010-09-11 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

5 febbraio 2011

1. Definiamo un *rw* come un oggetto su cui i processi possono leggere o scrivere rispettando le seguenti condizioni:

1. più processi possono leggere contemporaneamente, purchè nessun processo stia scrivendo;
2. un solo processo alla volta può scrivere.

Supporremo anche che ci sia un massimo (**MAX_RW_READERS**) al numero di processi che possono leggere contemporaneamente.

Per leggere o scrivere su un *rw*, i processi devono prima acquisire il diritto di lettura o scrittura, quindi devono rilasciare tale diritto quando hanno terminato.

Per realizzare i *rw* definiamo la seguente struttura (file **sistema.cpp**):

```
struct des_rw {
    natl readers[MAX_RW_READERS];
    natl writer;
    natl nreaders;
    des_proc* w_readers;
    des_proc* w_writers;
};
```

Il campo **readers** memorizza gli *id* degli eventuali processi che hanno acquisito il diritto di lettura e non lo hanno ancora rilasciato. Il campo **nreaders** memorizza il numero di tali processi. Il campo **writer** memorizza l'*id* dell'eventuale processo che ha acquisito il diritto di scrittura e non lo ha ancora rilasciato. La lista **w_readers** contiene i processi in attesa di acquisire il diritto di lettura. La lista **w_writers** contiene i processi in attesa di acquisire il diritto di scrittura.

Le seguenti primitive, accessibili dal livello utente, operano sui *rw* (nei casi di errore, abortiscono il processo chiamante):

- **natl rw_init()** (già realizzata): inizializza un nuovo *rw* e ne restituisce l'identificatore. Se non è possibile creare un nuovo *rw* restituisce 0xFFFFFFFF.
- **void rw_acq_write(natl rw)** (già realizzata): tenta di acquisire il diritto di scrittura sul *rw* di identificatore *rw*. Se ci sono già processi che hanno acquisito un diritto e non lo hanno ancora rilasciato, sospende il processo in attesa che le condizioni permettano l'acquisizione del diritto di scrittura.
- **void rw_acq_read(natl rw)** (già realizzata): tenta di acquisire il diritto di lettura sul *rw* di identificatore *rw*. Se c'è un processo che ha acquisito il diritto di scrittura e non lo ha ancora rilasciato, oppure se ci sono già **MAX_RW_READERS** processi che hanno acquisito il diritto di lettura e non lo hanno ancora rilasciato, sospende il processo in attesa che le condizioni permettano l'acquisizione del diritto di lettura.

- **void rw_rel_write(natl rw):** Rilascia il diritto di scrittura sul rw di identificatore **rw**. Se vi sono processi in attesa di acquisire un diritto di lettura o scrittura, lo concede dando precedenza ai lettori e cercando di concedere il diritto a più processi possibile, nel rispetto delle condizioni.
- **void rw_rel_read(natl rw):** Rilascia il diritto di lettura sul rw di identificatore **rw**. Se vi sono processi in attesa di acquisire un diritto di lettura o scrittura, lo concede dando precedenza agli scrittori e cercando di concedere il diritto a più processi possibile, nel rispetto delle condizioni.

È sempre un errore tentare di acquisire un diritto se se ne possiede già uno, o tentare di rilasciare un diritto che non si possiede.

Modificare i file **sistema.cpp** e **sistema.s** in modo da realizzare le primitive mancanti.

```

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


// ( SOLUZIONE 2011-02-05

extern "C" void c_rw_rel_write(natl rw)
{
    if (!rw_valido(rw)) {
        flog(LOG_WARN, "rw non valido: %d", rw);
        c_abort_p();
        return;
    }

    des_rw *p_des = &array_desrw[rw];

    if (p_des->writer != esecuzione->id) {
        flog(LOG_WARN, "tenta di rilasciare la scrittura su %d", rw);
        c_abort_p();
        return;
    }

    p_des->writer = 0xFFFFFFFF;
    inserimento_lista(pronti, esecuzione);
    if (p_des->w_readers) {
        while (p_des->w_readers && p_des->nreaders < MAX_RW_READERS) {
            des_proc* work = rimozione_lista(p_des->w_readers);
            p_des->readers[p_des->nreaders++] = work->id;
            inserimento_lista(pronti, work);
        }
    } else if (p_des->w_writers) {
        des_proc *work = rimozione_lista(p_des->w_writers);
        p_des->writer = work->id;
        inserimento_lista(pronti, work);
    }
    schedulatore();
}

extern "C" void c_rw_rel_read(natl rw)
{
    if (!rw_valido(rw)) {
        flog(LOG_WARN, "rw non valido: %d", rw);
        c_abort_p();
        return;
    }

    des_rw *p_des = &array_desrw[rw];

    natl pos = rw_find_reader(p_des, esecuzione->id);
    if (pos == 0xFFFFFFFF) {
        flog(LOG_WARN, "tenta di rilasciare la lettura su %d", rw);
        c_abort_p();
        return;
    }

    inserimento_lista(pronti, esecuzione);
    if (p_des->nreaders == 1 && p_des->w_writers) {
        des_proc* work = rimozione_lista(p_des->w_writers);
        p_des->writer = work->id;
        p_des->nreaders = 0;
    }
}

```

```
    inserimento_lista(pronti, work);
} else if (p_des->w_readers) {
    des_proc* work = rimozione_lista(p_des->w_readers);
    p_des->readers[pos] = work->id;
    inserimento_lista(pronti, work);
} else {
    p_des->nreaders--;
    p_des->readers[pos] = 0xFFFFFFFF;
}
schedulatore();
}

// SOLUZIONE 2011-02-05 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

22 giugno 2011

1. Vogliamo aggiungere al nucleo il meccanismo delle interruzioni inter-processo. Un qualunque processo può inviare una interruzione ad un altro processo di cui conosce l'identificatore. L'interruzione setta un flag nel descrittore del processo destinatario. Ogni processo può esaminare lo stato del proprio flag con una opportuna primitiva. Se il processo destinatario è bloccato su un semaforo l'operazione di interruzione provvede anche a risvegliarlo.

A tale scopo aggiungiamo i seguenti campi al descrittore di ogni processo:

```
bool interrupted;
natl blocked;
```

Il campo `interrupted` è il flag di interruzione, posto a `false` alla creazione del processo. Il campo `blocked` è l'indice dell'eventuale semaforo su cui il processo è bloccato (`0xFFFFFFFF` se il processo non è bloccato su un semaforo).

Aggiungiamo inoltre le seguenti primitive:

- `bool sem_wait2(natl sem)`: Opera su normali semafori. Si comporta come una normale `sem_wait`, ma se il processo si deve bloccare ed è stato interrotto (flag `interrupted` settato) restetta il flag, restituisce `false` e termina. Quando termina normalmente (perché il contatore non era 0 oppure perché il processo è stato risvegliato da una `sem_signal2`) restituisce `true`. Aggiorna opportunamente i campi nel descrittore di processo.
- `void sem_signal2(natl sem)` (già realizzata): Opera su normali semafori. Si comporta come una normale `sem_signal`, ma aggiorna opportunamente i campi nel descrittore di processo e completa il funzionamento della `sem_wait2`.
- `bool interrupt(natl id)`: Se `id` non corrisponde ad un processo esistente restituisce `false` e termina. In tutti gli altri casi restituisce `true`. Se il processo era già stato interrotto non fa altro. Se il processo non era stato interrotto e non è bloccato su un semaforo si limita a settare l'opportuno flag. Infine, se il processo non era stato interrotto ed è bloccato su un semaforo lo sblocca. Aggiorna opportunamente i campi del descrittore di processo e del semaforo e gestisce eventuali *preemption*.
- `bool interrupted()` (già realizzata): restituisce il valore del flag `interrupt` del processo corrente e lo resetta.

Sono disponibili le seguenti funzioni (già realizzate):

- `des_proc* des_p(natl id)`: restituisce un puntatore al descrittore di processo del processo di identificatore `id`. Restituisce 0 se `id` non è valido.
- `proc_elem* elimina_da_lista(proc_elem*& testa, des_proc* p)`: rimuove dalla lista `testa` il `proc_elem` che punta al processo con descrittore puntato da `p`, se presente. Restituisce un puntatore al `proc_elem` rimosso, se trovato, altrimenti restituisce 0.

ATTENZIONE: una primitiva che usa `salva_stato` e `carica_stato` può restituire valori al chiamante, ma per farlo deve modificare il campo del descrittore di processo che contiene il valore del registro `%RAX`. Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti.

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

```
// ( SOLUZIONE 2011-06-22 #1
extern "C" void c_sem_wait2(natl sem)
{
    des_sem *s;

    if (!sem_valido(sem)) {
        flog(LOG_WARN, "semaforo errato: %d", sem);
        c_abort_p();
        return;
    }

    s = &array_dess[sem];
    if (s->counter <= 0 && esecuzione->interrupted) {
        esecuzione->interrupted = false;
        esecuzione->contesto[I_RAX] = (natq)false;
        return;
    }
    s->counter--;

    if (s->counter < 0) {
        esecuzione->blocked = sem;
        inserimento_lista(s->pointer, esecuzione);
        schedulatore();
    } else
        esecuzione->contesto[I_RAX] = (natq)true;
}

extern "C" void c_interrupt(natl id)
{
    if (id >= MAX_PROC_ID) {
        flog(LOG_WARN, "id non valido: %d", id);
        c_abort_p();
        return;
    }

    des_proc *dp = des_p(id);

    if (!dp) {
        esecuzione->contesto[I_RAX] = (natl)false;
        return;
    }
    esecuzione->contesto[I_RAX] = (natl)true;

    if (esecuzione->interrupted)
        return;

    inspronti();
    if (dp->blocked != 0xFFFFFFFF) {
        des_sem *s = &array_dess[dp->blocked];
        elimina_da_lista(s->pointer, dp);
        s->counter++;
        dp->contesto[I_RAX] = (natq)false;
        dp->blocked = 0xFFFFFFFF;
        inserimento_lista(pronti, dp);
    } else
        dp->interrupted = true;
    schedulatore();
```

}

// SOLUZIONE 2011-06-22 #1)

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

6 luglio 2011

1. Vogliamo aggiungere al nucleo il meccanismo delle interruzioni inter-processo. Un qualunque processo può inviare una interruzione ad un altro processo di cui conosce l'identificatore. L'interruzione setta un flag nel descrittore del processo destinatario. Ogni processo può esaminare lo stato del proprio flag con una opportuna primitiva.

Se il processo destinatario si era sospeso sulla coda del timer con la primitiva `delay(nat1 n)`, l'operazione di interruzione, invece di settare il flag, risveglia il processo. La primitiva `delay(nat1 n)` può ora terminare per due motivi distinti: o perché sono trascorsi `n` intervalli di tempo, o perché il processo è stato interrotto. Per poter distinguere questi due casi la primitiva restituisce un valore al chiamante. Tale valore è il numero di intervalli di tempo che il processo avrebbe dovuto ancora attendere (e dunque è 0 se il processo si è risvegliato normalmente).

A tale scopo aggiungiamo i seguenti campi al descrittore di ogni processo:

```
bool interrupted;
bool sleeping;
```

Il campo `interrupted` è il flag di interruzione, posto a `false` alla creazione del processo. Il campo `sleeping` è un flag che vale `true` se il processo si trova nella coda del timer, e `false` altrimenti.

Modifichiamo inoltre la primitiva `delay` in modo che restituisca un `nat1`, e le funzioni `c_delay` e `c_driver_td` in modo che gestiscano il flag `sleeping` e il valore di ritorno della `delay` (per le modifiche a queste due funzioni si rimanda al file `sistema.cpp` fornito)

Aggiungiamo infine le seguenti primitive:

- `bool interrupt(nat1 id)`: Se `id` non corrisponde ad un processo esistente restituisce `false` e termina. In tutti gli altri casi restituisce `true`. Se il flag `interrupted` è già a `true` non fa altro. Se il flag `interrupted` è `false` e il processo non è nella coda del timer, si limita a settare l'opportuno flag. Infine, se il flag `interrupted` è `false` ed il processo si trova nella coda del timer lo risveglia. Per risveglierlo deve rimuoverlo dalla coda del timer, avendo cura di riportarla in uno stato consistente. Aggiorna opportunamente i campi del descrittore di processo e gestisce eventuali *preemption*.
- `bool interrupted()` (già realizzata): restituisce il valore del flag `interrupt` del processo corrente e lo resetta.

ATTENZIONE: una primitiva che usa `salva_stato` e `carica_stato` può restituire valori al chiamante, ma per farlo deve modificare il campo del descrittore di processo che contiene il valore del registro `%RAX`. Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare la primitiva `interrupt()`.

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

```
// ( SOLUZIONE 2011-07-06
richiesta* rimozione_lista_attesa(natl id)
{
    richiesta *prec = 0, *scorri = p_sospesi;

    natl d = 0;
    while (scorri && scorri->pp->id != id) {
        d += scorri->d_attesa;
        prec = scorri;
        scorri = scorri->p_rich;
    }
    if (!scorri)
        return 0;
    if (prec)
        prec->p_rich = scorri->p_rich;
    else
        p_sospesi = scorri->p_rich;
    richiesta *r = scorri;
    scorri = r->p_rich;
    if (scorri) {
        scorri->d_attesa += r->d_attesa;
    }
    r->d_attesa += d;
    r->p_rich = 0;
    return r;
}

extern "C" void c_interrupt(natl id)
{
    if (id >= MAX_PROC_ID) {
        flog(LOG_WARN, "id non valido: %d", id);
        c_abort_p();
        return;
    }

    des_proc *dp = des_p(id);

    if (!dp) {
        esecuzione->contesto[I_RAX] = (natq)false;
        return;
    }
    esecuzione->contesto[I_RAX] = (natq)true;

    if (dp->interrupted)
        return;

    inserimento_lista(pronti, esecuzione);
    if (dp->sleeping) {
        richiesta * r = rimozione_lista_attesa(id);
        dp->contesto[I_RAX] = r->d_attesa;
        dp->sleeping = false;
        inserimento_lista(pronti, r->pp);
        dealloca(r);
    } else
        dp->interrupted = true;
    schedulatore();
}
```

```
}
```

```
//  SOLUZIONE 2011-07-06 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

20 luglio 2011

1. Vogliamo aggiungere al nucleo il meccanismo delle interruzioni inter-processo. Un qualunque processo può inviare una interruzione ad un altro processo di cui conosce l'identificatore. L'interruzione setta un flag nel descrittore del processo destinatario. Ogni processo può esaminare lo stato del proprio flag con una opportuna primitiva.

Se il processo destinatario si era sospeso sulla coda del timer con la primitiva `delay(nat1 n)`, l'operazione di interruzione, invece di settare il flag, risveglia il processo. La primitiva `delay(nat1 n)` può ora terminare per due motivi distinti: o perché sono trascorsi `n` intervalli di tempo, o perché il processo è stato interrotto. Per poter distinguere questi due casi la primitiva restituisce un valore al chiamante. Tale valore è il numero di intervalli di tempo che il processo avrebbe dovuto ancora attendere (e dunque è 0 se il processo si è risvegliato normalmente).

A tale scopo aggiungiamo i seguenti campi al descrittore di ogni processo:

```
bool interrupted;
bool sleeping;
```

Il campo `interrupted` è il flag di interruzione, posto a `false` alla creazione del processo. Il campo `sleeping` è un flag che vale `true` se il processo si trova nella coda del timer, e `false` altrimenti.

Modifichiamo inoltre la primitiva `delay` in modo che restituisca un `nat1`, e le funzioni `c_delay` e `c_driver_td` in modo che gestiscano il flag `sleeping` e il valore di ritorno della `delay` (per le modifiche a queste due funzioni si rimanda al file `sistema.cpp` fornito)

Aggiungiamo infine le seguenti primitive:

- `bool interrupt(nat1 id)`: Se `id` non corrisponde ad un processo esistente restituisce `false` e termina. In tutti gli altri casi restituisce `true`. Se il flag `interrupted` è già a `true` non fa altro. Se il flag `interrupted` è `false` e il processo non è nella coda del timer, si limita a settare l'opportuno flag. Infine, se il flag `interrupted` è `false` ed il processo si trova nella coda del timer lo risveglia. Per risveglierlo deve rimuoverlo dalla coda del timer, avendo cura di riportarla in uno stato consistente. Aggiorna opportunamente i campi del descrittore di processo e gestisce eventuali *preemption*.
- `bool interrupted()` (già realizzata): restituisce il valore del flag `interrupt` del processo corrente e lo resetta.

ATTENZIONE: una primitiva che usa `salva_stato` e `carica_stato` può restituire valori al chiamante, ma per farlo deve modificare il campo del descrittore di processo che contiene il valore del registro `%RAX`. Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare la primitiva `interrupt()`.

```

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


// ( SOLUZIONE 2011-07-20

extern "C" void c_segnala(natl id)
{
    if (id >= MAX_PROC_ID) {
        flog(LOG_WARN, "id non valido: %d", id);
        c_abort_p();
        return;
    }

    des_proc *dest = des_p(id);

    if (!dest) {
        esecuzione->contesto[I_RAX] = (natq)false;
        return;
    }

    if (dest->livello != LIV_UTENTE) {
        flog(LOG_WARN, "errore di protezione");
        c_abort_p();
        return;
    }

    if (!dest->gest)
        esecuzione->contesto[I_RAX] = (natq)true;

    dest->pendenti++;

    if (dest->pendenti == 1) {
        salva_ritorno(id);
        forza_ritorno(id);
    }

    esecuzione->contesto[I_RAX] = (natq)true;
}

extern "C" void c_termina_gestore()
{
    if (esecuzione->pendenti == 0) {
        flog(LOG_WARN, "chiamata errata a termina_gestore()");
        c_abort_p();
        return;
    }

    esecuzione->pendenti--;
    if (esecuzione->pendenti == 0)
        ripristina_ritorno(esecuzione->id);
    else
        forza_ritorno(esecuzione->id);
}

// SOLUZIONE 2011-07-20 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

9 settembre 2011

1. Vogliamo aggiungere al nucleo il meccanismo dei *segnali*. Un qualunque processo può inviare un segnale ad un altro processo (di livello utente) di cui conosce l'identificatore. I segnali sono caratterizzati da un *tipo*, dove il tipo è un numero che va da 0 a *MAX_SEGNALI*. Ogni processo di livello utente può associare un *gestore* diverso per ciascun tipo di segnale, dove per gestore si intende una qualunque funzione `void` e senza argomenti. Definiamo per comodità il seguente tipo per i puntatori a gestore:

```
typedef void (*gestore)();
```

(Puntatore a una funzione senza argomenti che non restituisce niente). Il gestore associato ad un certo segnale verrà eseguito dal processo ogni volta che il segnale viene ricevuto. Se il processo non aveva associato alcun gestore a quel segnale, il segnale viene ignorato.

Per realizzare tale meccanismo procediamo come segue. Supponiamo che il processo *P* invii un segnale al processo utente *Q*, che vi aveva associato il gestore *g*. Mentre *P* esegue la primitiva per inviare il segnale, necessariamente *Q* si trova a livello sistema. In fondo alla pila sistema di *Q* troveremo le 5 parole quadruple che permettono a *Q* di tornare al punto del codice di livello utente che stava eseguendo prima di innalzarsi a livello sistema. Per fare in modo che *Q* esegua il gestore *g*, modifichiamo queste parole quadruple (salvandone il vecchio valore) in modo che *Q* salti alla prima istruzione di *g* la prossima volta che tornerà a livello utente. Il gestore *g*, prima di terminare, dovrà chiamare una particolare primitiva che ripristini il valore precedentemente salvato per le 5 parole quadruple, in modo che al ritorno dalla primitiva stessa *Q* ritorni al punto in cui si trovava prima di ricevere il segnale. (Per permettere un corretto ritorno, salviamo e ripristiniamo l'intero stato). Per svolgere le operazioni appena descritte aggiungiamo le seguenti funzioni di utilità:

- `void salva_ritorno(natl id)`: salva lo stato corrente del processo *id*.
- `void forza_ritorno(natl id, gestore g)`: cambia lo stato del processo *id* in modo da fargli eseguire il gestore *g*.
- `void ripristina_ritorno(natl id)`: ripristina lo stato salvato da `salva_ritorno(id)`.

Si noti che *Q* potrebbe ricevere più di un segnale prima di avere l'opportunità di eseguire un gestore, quindi dobbiamo tener conto dei segnali pendenti. Per semplificare l'implementazione stabiliamo che ci possa essere al più un segnale pendente per ogni tipo di segnale (se viene ricevuto un segnale di un tipo che era già pendente il nuovo segnale viene ignorato). Stabiliamo inoltre una priorità tra i segnali, data dall'ordine numerico del loro tipo: 0 è la priorità massima e *MAX_SEGNALI* – 1 la minima. All'invio di un segnale eseguiamo le operazioni di salvataggio solo se non c'erano già altri segnali pendenti, e forziamo il nostro gestore solo se il nostro segnale è quello a priorità massima; al termine di un gestore bisogna controllare se ci sono altri segnali pendenti (nel qual caso andrà eseguito il gestore a priorità massima tra questi) oppure no (nel qual caso bisogna ripristinare lo stato salvato).

Aggiungiamo i seguenti campi al descrittore di ogni processo:

```

gestore gest[MAX_SEGNALI];
bool    pendenti[MAX_SEGNALI];
//    salva_contesto

```

Il campo **gest** è il gestore associato al segnale (0 se nessun gestore è stato associato). Il campo **pendenti** tiene traccia dei segnali ricevuti e non ancora gestiti. Il campo **salva_contesto** contiene lo stato salvato da **salva_ritorno()**.

Aggiungiamo infine le seguenti primitive (in caso di errore abortiscono il processo):

- **void gestisci(nat1 signo, gestore gest)** (già realizzata): associa il gestore **gest** al segnale di tipo **signo**. Azzera eventuali segnali pendenti di quel tipo. (**gest** può essere 0).
- **bool segnala(nat1 signo, nat1 id)**: Invia un nuovo segnale di tipo **signo** al processo di identificatore **id**. Se tale processo non esiste o non ha associato un gestore, non fa altro. Se il processo **id** non esiste restituisce **false**, altrimenti restituisce **true**. È un errore se il processo di identificatore **id** non è un processo di livello utente. È un errore se **signo** non è minore di *MAX_SEGNALI*.
- **void termina_gestore(nat1 signo)**: Primitiva che deve essere chiamata dai gestori di segnali di tipo **signo** prima di terminare. È un errore se la primitiva viene chiamata senza che ci siano segnali di tipo **signo** pendenti.

Modificare i file **sistema.cpp** e **sistema.s** in modo da realizzare le primitive mancanti.

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

// (SOLUZIONE 2011-09-09

```
natl maxpending(bool pendenti[])
{
    natl signo = 0;
    while (signo < MAX_SEGNALI && !pendenti[signo])
        signo++;
    return signo;
}

extern "C" void c_segnala(natl signo, natl id)
{
    if (signo >= MAX_SEGNALI) {
        flog(LOG_WARN, "segnale errato: %d", signo);
        c_abort_p();
        return;
    }
    des_proc *dest = des_p(id);

    if (!dest) {
        esecuzione->contesto[I_RAX] = (natq)false;
        return;
    }

    if (dest->livello != LIV_UTENTE) {
        flog(LOG_WARN, "errore di protezione");
        c_abort_p();
        return;
    }

    if (!dest->gest[signo]) {
        esecuzione->contesto[I_RAX] = (natq)true;
        return;
    }

    natl maxsig = maxpending(dest->pendenti);
    dest->pendenti[signo] = true;

    if (maxsig == MAX_SEGNALI)
        salva_ritorno(id);

    if (maxsig > signo)
        forza_ritorno(id, dest->gest[signo]);

    esecuzione->contesto[I_RAX] = (natq)true;
}

extern "C" void c_termina_gestore(natl signo)
{
    if (signo >= MAX_SEGNALI || !esecuzione->pendenti[signo]) {
        flog(LOG_WARN, "chiamata errata a termina_gestore()");
        c_abort_p();
        return;
    }

    esecuzione->pendenti[signo] = false;
```

```
natl maxsig = maxpending(esecuzione->pendenti);

if (maxsig == MAX_SEGNALI)
    ripristina_ritorno(esecuzione->id);
else
    forza_ritorno(esecuzione->id, esecuzione->gest[maxsig]);
}

// SOLUZIONE 2011-09-09 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

12 gennaio 2012

1. Vogliamo aggiungere al nucleo il meccanismo dei *segnali*. Un qualunque processo può inviare un segnale ad un altro processo (di livello utente) di cui conosce l'identificatore. I segnali sono caratterizzati da un *tipo*, dove il tipo è un numero che va da 0 a *MAX_SEGNALI*. Ogni processo di livello utente può associare un *gestore* diverso per ciascun tipo di segnale diverso da 0, dove per gestore si intende una qualunque funzione **void** e senza argomenti. Definiamo per comodità il seguente tipo per i puntatori a gestore:

```
typedef void (*gestore)();
```

(Puntatore a una funzione senza argomenti e che non restituisce niente). Il gestore associato ad un certo segnale verrà eseguito dal processo ogni volta che il segnale viene ricevuto. Se il processo non aveva associato alcun gestore a quel segnale, il segnale viene ignorato.

I segnali possono essere temporaneamente e selettivamente mascherati. Se un processo riceve un segnale mascherato, questo resta pendente e verrà gestito non appena sarà stato smascherato.

Il segnale numero 0 è speciale: non è possibile associarvi un gestore e non può essere mascherato. Se un processo richiede tali operazioni il nucleo le ignora.

Per realizzare tale meccanismo disponiamo delle seguenti funzioni di utilità:

- **void salva_ritorno(nat1 id)**: salva lo stato corrente del processo **id**.
- **void forza_ritorno(nat1 id, gestore g)**: cambia lo stato del processo **id** in modo da fargli eseguire il gestore **g**.
- **void ripristina_ritorno(nat1 id)**: ripristina lo stato salvato da **salva_ritorno(id)**.

Si noti che un processo potrebbe ricevere più di un segnale prima di avere l'opportunità di eseguire un gestore, quindi dobbiamo tener conto dei segnali pendenti. Per semplificare l'implementazione stabiliamo che ci possa essere al più un segnale pendente per ogni tipo di segnale (se viene ricevuto un segnale di un tipo che era già pendente il nuovo segnale viene ignorato). Stabiliamo inoltre una priorità tra i segnali, data dall'ordine numerico del loro tipo: 0 è la priorità massima e *MAX_SEGNALI* – 1 la minima. All'invio di un segnale eseguiamo le operazioni di salvataggio solo se non c'erano già altri segnali pendenti, e forziamo il nostro gestore solo se il nostro segnale è quello a priorità massima; al termine di un gestore bisogna controllare se ci sono altri segnali pendenti (nel qual caso andrà eseguito il gestore a priorità massima tra questi) oppure no (nel qual caso bisogna ripristinare lo stato salvato).

Aggiungiamo i seguenti campi al descrittore di ogni processo:

```
gestore gest[MAX_SEGNALI];
bool    pendenti[MAX_SEGNALI];
bool    mascherati[MAX_SEGNALI];
//    salva_contesto
```

Il campo `gest` è il gestore associato al segnale (0 se nessun gestore è stato associato). Il campo `pendenti` conta il numero di segnali ricevuti e non ancora gestiti. Il campo `mascherati` tiene conto di quali segnali sono attualmente mascherati. Il campo `salva_contesto` contiene lo stato salvato da `salva_ritorno()`.

Aggiungiamo infine le seguenti primitive (in caso di errore abortiscono il processo):

- `void gestisci(nat1 signo, gestore gest)` (da realizzare): associa il gestore `gest` al segnale di tipo `signo`, se permesso. Azzera eventuali segnali pendenti di quel tipo. (`gest` può essere 0).
- `bool segnala(nat1 signo, nat1 id)` (già realizzata): Invia un nuovo segnale di tipo `signo` al processo di identificatore `id`. Se tale processo non esiste, non ha associato un gestore o `signo` è 0, non fa altro. Se il processo `id` non esiste restituisce `false`, altrimenti restituisce `true`. È un errore se il processo di identificatore `id` non è un processo di livello utente. È un errore se `signo` non è minore di `MAX_SEGNALI`.
- `void termina_gestore(nat1 signo)` (già realizzata): Primitiva che deve essere chiamata dai gestori di segnali di tipo `signo` prima di terminare. È un errore se la primitiva viene chiamata senza che ci siano segnali di tipo `signo` pendenti.
- `void maschera(nat1 signo)` (da realizzare): Maschera il segnale `signo` per il processo che la invoca, se permesso.
- `void smaschera(nat1 signo)` (da realizzare): Smaschera il segnale `signo` per il processo che la invoca, se permesso. (Attenzione: se il segnale appena smascherato era pendente deve essere gestito).

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti.

```

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


// ( SOLUZIONE 2012-01-12

extern "C" void c_gestisci(natl signo, gestore g)
{
    if (signo >= MAX_SEGNALI) {
        flog(LOG_WARN, "segnale errato: %d", signo);
        c_abort_p();
        return;
    }
    if (signo == 0)
        return;

    esecuzione->gest[signo] = g;
    esecuzione->pendenti[signo] = false;
}

extern "C" void c_maschera(natl signo) {
    if (signo >= MAX_SEGNALI) {
        flog(LOG_WARN, "segnale errato: %d", signo);
        c_abort_p();
        return;
    }
    if (signo == 0)
        return;

    esecuzione->mascherati[signo] = true;
}

extern "C" void c_smaschera(natl signo) {
    if (signo >= MAX_SEGNALI) {
        flog(LOG_WARN, "segnale errato: %d", signo);
        c_abort_p();
        return;
    }
    if (signo == 0)
        return;

    if (esecuzione->mascherati[signo]) {
        natl maxsig = maxpending(esecuzione);
        esecuzione->mascherati[signo] = false;
        if (esecuzione->pendenti[signo]) {
            if (maxsig == MAX_SEGNALI)
                salva_ritorno(esecuzione->id);
            if (signo < maxsig)
                forza_ritorno(esecuzione->id, esecuzione-
>gest[signo]);
        }
    }
}

// SOLUZIONE 2012-01-12 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

1 febbraio 2012

1. Vogliamo aggiungere la schedulazione di tipo *Round Robin* al nucleo. In questo tipo di schedulazione ogni processo può restare continuamente in esecuzione soltanto per certo tempo massimo, dopo di che deve cedere il processore ad un altro processo (il primo in coda pronti). Nel cedere il processore, il processo si reinserisce in coda pronti come ultimo. Poichè nel nucleo abbiamo anche le priorità, prevediamo che il processo si inserisca come ultimo tra quelli che hanno la stessa priorità (ma prima di quelli con priorità inferiore). In questo modo i processi di uguale priorità vengono eseguiti a rotazione.

Per realizzare questo tipo di schedulazione aggiungiamo al descrittore di processo un campo `int quanto`, che rappresenta il numero di tick del timer che il processo ha ancora a disposizione prima di dover cedere il processore. Il numero massimo di tick per i quali ciascun processo può restare in modo continuato in esecuzione è contenuto nella costante `MAX_QUANTO`. Ogni volta che il driver del timer va in esecuzione, decrementa il campo `quanto` del processo attualmente in esecuzione e provvede ad eseguire le azioni necessarie ad implementare quanto sopra illustrato.

È possibile utilizzare la funzione `des_proc* des_p(nat1 id)` che restituisce il puntatore al descrittore del processo il cui id è passato come argomento (restituisce 0 se il processo non esiste).

Aggiungiamo infine le seguenti primitive:

- `void abilita_rr()` (da realizzare): abilita la schedulazione Round Robin. Il driver del timer deve eseguire quanto sopra specificato solo se la schedulazione Round Robin è abilitata.
- `void disabilita_rr()` (da realizzare): Disabilita la schedulazione round robin. Se questa era abilitata provvede anche a riportare a `MAX_QUANTO` i campi `quanto` di tutti i processi esistenti nel sistema.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive e il codice mancante.

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

```
// ( SOLUZIONE 2012-02-01 #1
    carica_gateTIPO_A_RR  a_abilita_rr      LIV_UTENTE
    carica_gateTIPO_D_RR  a_disabilita_rr   LIV_UTENTE
//  SOLUZIONE 2012-02-01 )
// ( SOLUZIONE 2012-02-01 #1

a_abilita_rr:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_abilita_rr
    iretq
    .cfi_endproc

a_disabilita_rr:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_disabilita_rr
    iretq
    .cfi_endproc
//  SOLUZIONE 2012-02-01 #1 )
```

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

```
// ( SOLUZIONE 2012-02-01 #1
bool rr_abilitato = false;

extern "C" void c_abilita_rr()
{
    rr_abilitato = true;
}

extern "C" void c_disabilita_rr()
{
    if (!rr_abilitato)
        return;

    rr_abilitato = false;

    for (natl id = MIN_PROC_ID; id <= MAX_PROC_ID; id++) {
        des_proc *p = des_p(id);
        if (p) {
            p->quanto = MAX_QUANTO;
        }
    }
}
//  SOLUZIONE 2012-02-01 )
// ( SOLUZIONE 2012-02-01 #2
    if (rr_abilitato) {
        if (--esecuzione->quanto <= 0) {
```

```
    esecuzione->quanto = MAX_QUANTO;
    inserimento_lista(pronti, esecuzione);
    schedulatore();
}
}
// SOLUZIONE 2012-02-01 )
// ( SOLUZIONE 2012-02-01 #3
p->quanto = MAX_QUANTO;
// SOLUZIONE 2012-02-01 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

15 febbraio 2012

1. Vogliamo aggiungere la schedulazione di tipo *Round Robin* al nucleo. In questo tipo di schedulazione ogni processo può restare continuamente in esecuzione soltanto per certo tempo massimo, dopo di che deve cedere il processore ad un altro processo (il primo in coda pronti). Nel cedere il processore, il processo si reinserisce in coda pronti come ultimo.

Per realizzare questo tipo di schedulazione aggiungiamo al descrittore di processo un campo `int quanto`, che rappresenta il numero di tick del timer che il processo ha ancora a disposizione prima di dover cedere il processore. Il numero massimo di tick per i quali ciascun processo può restare in modo continuato in esecuzione è contenuto nella costante `MAX_QUANTO`. Ogni volta che il driver del timer va in esecuzione, decrementa il campo `quanto` del processo attualmente in esecuzione e provvede ad eseguire le azioni necessarie ad implementare quanto sopra illustrato.

È possibile utilizzare la funzione `des_proc* des_p(nat1 id)` che restituisce il puntatore al descrittore del processo il cui id è passato come argomento (restituisce 0 se il processo non esiste).

La schedulazione Round Robin può essere abilitata o disabilitata dinamicamente: quando è disabilitata la schduulazione avviene in base all apriorità dei processi; quando è abilitat le priorità dei processi devono essere ignorate.

Aggiungiamo infine le seguenti primitive:

- `void abilita_rr()` (da realizzare): abilita la schedulazione Round Robin. Il driver del timer deve eseguire quanto sopra specificato solo se la schedulazione Round Robin è abilitata.
- `void disabilita_rr()` (da realizzare): Disabilita la schedulazione round robin. Se questa era abilitata provvede anche a riportare a `MAX_QUANTO` i campi `quanto` di tutti i processi esistenti nel sistema.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive e il codice mancante.

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

```
// ( SOLUZIONE 2012-02-01 #1
    carica_gateTIPO_A_RR  a_abilita_rr      LIV_UTENTE
    carica_gateTIPO_D_RR  a_disabilita_rr   LIV_UTENTE
//  SOLUZIONE 2012-02-01 )
// ( SOLUZIONE 2012-02-01 #2
    .extern c_abilita_rr
a_abilita_rr:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_abilita_rr
    iretq
    .cfi_endproc

    .extern c_disabilita_rr
a_disabilita_rr:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_disabilita_rr
    iretq
    .cfi_endproc
//  SOLUZIONE 2012-02-01 )
```

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

```
// ( SOLUZIONE 2012-02-01 #1
bool rr_abilitato = false;

extern "C" void c_abilita_rr()
{
    rr_abilitato = true;
}

extern "C" void c_disabilita_rr()
{
    if (!rr_abilitato)
        return;

    rr_abilitato = false;
    des_proc* tmp = nullptr;
    while (pronti) {
        des_proc *lavoro = rimozione_lista(pronti);
        inserimento_lista(tmp, lavoro);
    }
    inserimento_lista(tmp, esecuzione);
    pronti = tmp;
    schedulatore();
}

void inserimento_fondo_lista(des_proc* &testa, des_proc* elem)
```

```
{  
    des_proc** pscorri = &testa;  
    while (*pscorri)  
        pscorri = &((*pscorri)->puntatore);  
    *pscorri = elem;  
    elem->puntatore = nullptr;  
}  
  
// SOLUZIONE 2012-02-01 )  
// ( SOLUZIONE 2012-02-01 #2  
if (rr_abilitato) {  
    if (--esecuzione->quanto <= 0) {  
        esecuzione->quanto = MAX_QUANTO;  
        inserimento_fondo_lista(pronti, esecuzione);  
        schedulatore();  
    }  
}  
  
// SOLUZIONE 2012-02-01 )  
// ( SOLUZIONE 2012-02-01 #3  
p->quanto = MAX_QUANTO;  
// SOLUZIONE 2012-02-01 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

7 giugno 2012

1. Vogliamo aggiungere al nucleo un meccanismo di scambio di messaggi. In questo meccanismo un processo può inviare un messaggio `msg` ad un altro processo di cui conosce l'identificatore `id`, usando una primitiva `send(id, msg)`. Assumiamo che il messaggio sia un `nat1`. Più processi possono inviare messaggi contemporaneamente ad uno stesso processo. Un processo può mettersi in ascolto di messaggi a lui indirizzati usando una primitiva `nat1 receive()`. La primitiva blocca il processo che la invoca fino a quando qualche altro processo non gli invia un messaggio; quindi la primitiva restituisce uno dei messaggi ricevuti. Supporremo inoltre che anche la primitiva `send` blocchi il processo che invia fino a quando il processo destinatario non è pronto a ricevere messaggio inviato.

Può accadere che il processo destinatario non esista, oppure termini prima di ricevere il messaggio. In questi casi la primitiva `send` deve restituire un errore.

Per realizzare il precedente meccanismo aggiungiamo i seguenti campi al descrittore di processo:

```
des_proc* senders;
bool waiting;
nat1 msg;
```

Consideriamo il descrittore di processo di un dato processo P . Il campo `senders` è una coda di processi (ordinata per priorità) su cui si sospendono (se necessario) i processi che vogliono inviare un messaggio al processo P . Il campo `waiting` vale `true` quando il processo P stesso è sospeso in attesa della ricezione di un messaggio. Il campo `msg` è utilizzato quando P vuole inviare un messaggio ad un altro processo, ma si deve sospendere. Il suo scopo è di memorizzare il messaggio che P vuole inviare.

Aggiungiamo dunque le seguenti primitive:

- `bool send(nat1 id, nat1 msg)` (da realizzare): Invia il messaggio `msg` al processo di identificatore `id`. Ritorna quando il messaggio è stato ricevuto, restituendo `true`, oppure se non è stato possibile recapitare il messaggio, restituendo `false`.
- `nat1 receive()` (da realizzare): Si pone in attesa di un messaggio. Ritorna quando un messaggio è stato ricevuto e lo restituisce.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive e il codice mancante.

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


// ( SOLUZIONE 2012-06-07

extern "C" void c_send(natl id, natl msg)
{
    if (id >= MAX_PROC_ID) {
        flog(LOG_WARN, "id non valido: %d", id);
        c_abort_p();
        return;
    }

    des_proc *dest = proc_table[id];

    if (!dest) {
        esecuzione->contesto[I_RAX] = false;
        return;
    }

    if (!dest->waiting) {
        inserimento_lista(dest->senders, esecuzione);
        esecuzione->msg = msg;
        schedulatore();
        return;
    }

    inspronti();
    inserimento_lista(pronti, dest);
    dest->waiting = false;
    dest->contesto[I_RAX] = msg;
    esecuzione->contesto[I_RAX] = true;
    schedulatore();
}

extern "C" void c_receive()
{
    des_proc *sender;

    if (!esecuzione->senders) {
        esecuzione->waiting = true;
        schedulatore();
        return;
    }

    sender = rimozione_lista(esecuzione->senders);
    sender->contesto[I_RAX] = true;
    esecuzione->contesto[I_RAX] = sender->msg;
    inspronti();
    inserimento_lista(pronti, sender);
    schedulatore();
}

// SOLUZIONE 2012-06-07 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

27 giugno 2012

1. Vogliamo aggiungere al nucleo un meccanismo di scambio di messaggi. In questo meccanismo un processo può inviare un messaggio `msg` ad un altro processo di cui conosce l'identificatore `id`, usando una primitiva `send(id, msg)`. Assumiamo che il messaggio sia un `nat1`. Più processi possono inviare messaggi contemporaneamente ad uno stesso processo. Un processo può mettersi in ascolto di messaggi a lui indirizzati usando una primitiva `nat1 receive()`. La primitiva blocca il processo che la invoca fino a quando qualche altro processo non gli invia un messaggio; quindi la primitiva restituisce il primo dei messaggi ricevuti.

Ogni processo possiede una coda di `MAX_MSG` messaggi pendenti. La primitiva `send` si limita ad accodare un nuovo messaggio nella coda del processo destinatario e ritorna senza attendere che il destinatario prelevi il messaggio. Se la coda è piena il messaggio viene scartato e la primitiva restituisce un errore.

Può accadere che il processo destinatario non esista, oppure termini prima di ricevere il messaggio. In questi casi la primitiva `send` deve restituire un errore.

Per realizzare il precedente meccanismo aggiungiamo i seguenti campi al descrittore di processo:

```
bool waiting;
nat1 msg[MAX_MSG];
nat1 first_free;
nat1 first_unread;
nat1 n_msg;
```

Consideriamo il descrittore di un processo P . Il campo `waiting` vale `true` se il processo P stesso è sospeso in attesa di un messaggio. I campi `msg`, `first_free`, `first_unread` e `n_msg` servono a realizzare una coda circolare di messaggi in attesa di essere ricevuti.

Aggiungiamo infine le seguenti primitive:

- `nat1 send(nat1 id, nat1 msg)` (da realizzare): . Invia il messaggio `msg` al processo di identificatore `id`. Restituisce 0 se il messaggio è stato correttamente accodato, 1 il processo non esiste e 2 se la coda era piena.
- `nat1 receive()` (da realizzare): Si pone in attesa di un messaggio. Ritorna quando almeno un messaggio è stato ricevuto restituisce quello accodato da più tempo.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive e il codice mancante.

N.B. Gestire correttamente eventuali *preemption*.

```

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


// ( SOLUZIONE 2012-06-27

extern "C" void c_send(natl id, natl msg)
{
    if (id >= MAX_PROC_ID) {
        flog(LOG_WARN, "id %d non valido");
        c_abort_p();
        return;
    }

    des_proc *dst = des_p(id);

    if (!dst) {
        esecuzione->contesto[I_RAX] = 1;
        return;
    }

    if (dst->n_msg == MAX_MSG) {
        esecuzione->contesto[I_RAX] = 2;
        return;
    }

    esecuzione->contesto[I_RAX] = 0;

    inspronti();
    if (dst->waiting) {
        dst->contesto[I_RAX] = msg;
        inserimento_lista(pronti, dst);
        dst->waiting = false;
    } else {
        dst->msg[dst->first_free] = msg;
        dst->first_free = (dst->first_free + 1) % MAX_MSG;
        dst->n_msg++;
    }
    schedulatore();
}

extern "C" void c_receive()
{
    if (!esecuzione->n_msg) {
        esecuzione->waiting = esecuzione;
        schedulatore();
        return;
    }

    esecuzione->contesto[I_RAX] = esecuzione->msg[esecuzione->first_unread];
    esecuzione->first_unread = (esecuzione->first_unread + 1) % MAX_MSG;
    esecuzione->n_msg--;
}

// SOLUZIONE 2012-06-27 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

18 luglio 2012

1. Vogliamo aggiungere al nucleo un meccanismo di scambio di messaggi tramite canali. In questo meccanismo un processo può inviare un messaggio `msg` su un canale da cui un altro processo può poi prelevarlo. Assumiamo che i messaggi siano di tipo `natl`. Ogni canale ha un identificatore, di tipo `natl`. Il canale accoda i messaggi inviati e non ancora ricevuti, e ogni canale ha una coda di lunghezza massima finita. Un processo che vuole inviare un nuovo messaggio su un canale e trova la coda piena si blocca fino a quando almeno un messaggio non è stato ricevuto. Un processo che vuole ricevere un messaggio da un canale, ma trova la coda vuota, si blocca fino a quando almeno un messaggio non è stato inviato.

Per realizzare il precedente meccanismo introduciamo il seguente descrittore di canale:

```
struct des_channel {  
    des_proc *wait_w;  
    des_proc *wait_r;  
    natl *msg_buf;  
    natl size;  
    natl n_free;  
    natl first_free;  
    natl first_unread;  
};
```

Il campo `wait_w` è una coda su cui si sospendono i processi in attesa di inviare un messaggio. Il campo `wait_r` è una coda su cui si sospendono i processi in attesa di ricevere un messaggio. Il campo `msg_buf` punta ad un buffer che può contenere al massimo `size` elementi di tipo `natl`. I campi `first_free`, `first_unread` e `n_free` servono a realizzare, nel buffer puntato da `msg_buf`, una coda circolare di messaggi in attesa di essere ricevuti.

Inoltre aggiungiamo un campo `natl msg` ai descrittori di processo. Un processo che si deve bloccare a causa di una coda piena può salvare qui il messaggio che voleva inviare.

Aggiungiamo infine le seguenti primitive (in caso di errore abortiscono il processo):

- `natl channel_init(natl size)` (da realizzare): alloca un nuovo canale che può contenere al massimo `size` messaggi e ne restituisce l'identificatore. Se l'allocazione di un nuovo canale non è possibile, non alloca alcuna risorsa e restituisce `0xFFFFFFFF`.
- `void send(natl chan_id, natl msg)` (già realizzata): invia il messaggio `msg` sul canale `chan_id`. È un errore se `chan_id` non corrisponde ad un canale precedentemente allocato.
- `natl receive(natl chan_id)` (da realizzare): riceve un messaggio dal canale `chan_id`. È un errore se `chan_id` non corrisponde ad un canale precedentemente allocato.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive e il codice mancante.

N.B. Gestire correttamente eventuali *preemption*.

```

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


// ( SOLUZIONE 2012-07-18
extern "C" natl c_channel_init(natl size)
{
    if (next_channel >= MAX_CHANNELS)
        return 0xFFFFFFFF;

    natl *buf = (natl*)alloca(size * sizeof(natl));
    if (!buf)
        return 0xFFFFFFFF;
    natl id = next_channel++;
    des_channel *des_c = &array_deschan[id];
    des_c->wait_w = 0;
    des_c->wait_r = 0;
    des_c->msg_buf = buf;
    des_c->size = size;
    des_c->n_free = size;
    des_c->first_unread = 0;
    des_c->first_free = 0;

    return id;
}

extern "C" void c_channel_receive(natl id)
{
    if (id >= next_channel) {
        flog(LOG_WARN, "channel id errato: %d", id);
        c_abort_p();
        return;
    }

    des_channel *des_c = &array_deschan[id];

    if (des_c->n_free == des_c->size) {
        inserimento_lista(des_c->wait_r, esecuzione);
        schedulatore();
        return;
    }
    natl msg = des_c->msg_buf[des_c->first_unread];
    des_c->first_unread = (des_c->first_unread + 1) % des_c->size;
    des_c->n_free++;
    des_proc *self = des_p(esecuzione->id);
    self->contesto[I_RAX] = msg;
    if (des_c->wait_w) {
        des_proc *p = rimozione_lista(des_c->wait_w);
        msg = p->msg;
        des_c->msg_buf[des_c->first_free] = msg;
        des_c->first_free = (des_c->first_free + 1) % des_c->size;
        des_c->n_free--;
        inspronti();
        inserimento_lista(pronti, p);
        schedulatore();
    }
}

//      SOLUZIONE 2012-07-18 )

```


Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

13 settembre 2012

- Vogliamo aggiungere al nucleo un meccanismo di scambio di messaggi tramite canali. In questo meccanismo un processo può inviare un messaggio `msg` su un canale da cui un altro processo può poi prelevarlo. Assumiamo che i messaggi siano di tipo `nat1`. Ogni canale ha un identificatore, di tipo `nat1`. Il canale accoda i messaggi inviati e non ancora ricevuti, e ogni canale ha una coda di lunghezza massima finita. Un processo che vuole inviare un nuovo messaggio su un canale e trova la coda piena si blocca fino a quando almeno un messaggio non è stato ricevuto. Un processo che vuole ricevere un messaggio da un canale, ma trova la coda vuota, si blocca fino a quando almeno un messaggio non è stato inviato.

Inoltre, un processo può porsi in attesa da due canali contemporaneamente. In questo caso il processo si blocca solo se non ci sono messaggi su entrambi i canali e si risveglia non appena c'è un messaggio su almeno uno dei canali. Se c'è un messaggio su entrambi i canali, il processo legge dal canale specificato per primo.

Per realizzare il precedente meccanismo introduciamo il seguente descrittore di canale:

```
struct des_channel {  
    des_proc *wait_w;  
    des_proc *wait_r;  
    natl *msg_buf;  
    natl size;  
    natl n_free;  
    natl first_free;  
    natl first_unread;  
};
```

Il campo `wait_w` è una coda su cui si sospendono i processi in attesa di inviare un messaggio. Il campo `wait_r` è una coda su cui si sospendono i processi in attesa di ricevere un messaggio. Il campo `msg_buf` punta ad un buffer che può contenere al massimo `size` elementi di tipo `nat1`. I campi `first_free`, `first_unread` e `n_free` servono a realizzare, nel buffer puntato da `msg_buf`, una coda circolare di messaggi in attesa di essere ricevuti.

Inoltre aggiungiamo un campo `nat1 msg` ai descrittori di processo. Un processo che si deve bloccare a causa di una coda piena può salvare qui il messaggio che voleva inviare.

Aggiungiamo infine le seguenti primitive (in caso di errore abortiscono il processo):

- `natl channel_init(natl size)` (già realizzata): alloca un nuovo canale che può contenere al massimo `size` messaggi e ne restituisce l'identificatore. Se l'allocazione di un nuovo canale non è possibile, non alloca alcuna risorsa e restituisce `0xFFFFFFFF`.
- `void channel_send(natl chan_id, natl msg)` (da realizzare): invia il messaggio `msg` sul canale `chan_id`. **Attenzione:** il ricevitore potrebbe essere in attesa anche su un altro canale e quindi può essere necessario eliminarlo da ogni coda. È un errore se `chan_id` non corrisponde ad un canale precedentemente allocato.

- `natl channel_receive(natl chan_id)` (già realizzata): riceve un messaggio dal canale `chan_id`. È un errore se `chan_id` non corrisponde ad un canale precedentemente allocato.
- `natl channel_receive2(natl chan_id1, natl chan_id2)` (da realizzare): riceve un messaggio dai canali `chan_id1` e `chan_id2`. È un errore se `chan_id1` o `chan_id2` non corrispondono ad un canale precedentemente allocato.

Si assuma che su ogni canale vi possa essere un solo processo alla volta in attesa di ricevere.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive e il codice mancante.

N.B. Gestire correttamente eventuali *preemption*.

```

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

// ( SOLUZIONE 2012-09-13

void rimozione_tutti_canali(des_proc* elem)
{
    for (unsigned i = 0; i < next_channel; i++) {
        des_channel *des_c = &array_deschan[i];

        elimina_da_lista(des_c->wait_r, elem);
    }
}

extern "C" void c_channel_send(natl id, natl msg)
{
    if (id >= next_channel) {
        flog(LOG_WARN, "channel id errato: %d", id);
        c_abort_p();
        return;
    }

    des_channel *des_c = &array_deschan[id];

    if (!des_c->n_free) {
        inserimento_lista(des_c->wait_w, esecuzione);
        schedulatore();
        return;
    }
    inserimento_lista(pronti, esecuzione);
    if (des_c->wait_r) {
        des_proc *reader = rimozione_lista(des_c->wait_r);
        rimozione_tutti_canali(reader);
        reader->contesto[I_RAX] = msg;
        inserimento_lista(pronti, reader);
    } else {
        des_c->msg_buf[des_c->first_free] = msg;
        des_c->first_free = (des_c->first_free + 1) % des_c->size;
        des_c->n_free--;
    }
    schedulatore();
}

extern "C" void c_channel_receive2(natl id1, natl id2)
{
    if (id1 >= next_channel) {
        flog(LOG_WARN, "channel id1 errato: %d", id1);
        c_abort_p();
        return;
    }
    if (id2 >= next_channel) {
        flog(LOG_WARN, "channel id2 errato: %d", id2);
        c_abort_p();
        return;
    }

    des_channel *des_c1 = &array_deschan[id1];
    des_channel *des_c2 = &array_deschan[id2];
}

```

```

        if (des_c1->n_free == des_c1->size && des_c2->n_free == des_c2-
>size) {
            inserimento_lista(des_c1->wait_r, esecuzione);
            inserimento_lista(des_c2->wait_r, esecuzione);
            schedulatore();
            return;
        }
        des_channel *des_c = des_c1;
        if (des_c->n_free == des_c->size)
            des_c = des_c2;
        natl msg = des_c->msg_buf[des_c->first_unread];
        des_c->first_unread = (des_c->first_unread + 1) % des_c->size;
        des_c->n_free++;
        esecuzione->contesto[I_RAX] = msg;
        if (des_c->wait_w) {
            des_proc *p = rimozione_lista(des_c->wait_w);
            msg = p->msg;
            des_c->msg_buf[des_c->first_free] = msg;
            des_c->first_free = (des_c->first_free + 1) % des_c->size;
            des_c->n_free--;
            inspronti();
            inserimento_lista(pronti, p);
            schedulatore();
        }
    }
}

// SOLUZIONE 2012-09-13 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

30 gennaio 2013

- Vogliamo modificare il nucleo in modo che un processo, alla fine della propria esecuzione, possa comunicare un risultato al processo che lo ha creato. Assumiamo che tale dato sia di tipo `natl`. A tal fine modifichiamo la primitiva `void terminate_p()` in `void terminate_p(natl result)` e aggiungiamo una primitiva `natl join()`. Un processo figlio passa il risultato da comunicare al processo padre come parametro della `terminate_p()`. La nuova primitiva `join()` deve essere usata da un processo padre quando vuole recuperare il risultato di uno dei suoi figli terminati. Se nessun figlio è ancora terminato, la primitiva sospende il processo padre in attesa che almeno un figlio termini. (Se il processo non ha figli viene abortito).

Per realizzare tale meccanismo introduciamo i seguenti tipi:

```
enum status_t { RUNNING, TERMINATED };
struct child {
    status_t status;
    natl result;
    natl father;
    child* next;
};
```

La struttura `child` descrive lo stato di un processo figlio. Il campo `status` ci dice se il processo è già terminato oppure no. Il campo `result` contiene il risultato del processo, ed è significativo solo se il processo è già terminato. Il campo `father` contiene l'id del processo padre (se non ancora terminato, altrimenti contiene `0xFFFFFFFF`) e il campo `child` serve a costruire una lista dei processi figli di uno stesso processo.

Aggiungiamo tre campi al descrittore di processo:

```
child* me;
child* children;
bool waiting;
```

Il campo `children` punta al primo elemento della lista (eventualmente vuota) dei figli di questo processo. Questo processo è a sua volta un figlio, e il campo `me` punta al proprio descrittore `child` nella lista dei figli del padre. Tale struttura viene allocata, inizializzata e inserita in lista alla creazione del processo figlio. Quando un processo vuole attendere che uno dei suoi figli termini pone a `true` il campo `waiting`. **Attenzione:** per motivi tecnici alcuni processi non hanno un padre (l'id del padre è `0xFFFFFFFF`). In quel caso la struttura `child` non deve essere inserita in lista.

La struttura `child` di un dato processo figlio deve essere deallocata quando il padre ha letto il risultato di quel processo. Per assicurarsi che le strutture vengano deallocate in tutti casi, adottiamo le seguenti regole seguite da ogni processo alla propria terminazione:

- il padre dealloca tutte le strutture `child` dei processi figli terminati e pone a `0xFFFFFFFF` il campo `father` delle altre. `father`;

2. il figlio dealloca la propria struttura `child` se il padre non esiste oppure è già terminato.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive e il codice mancante.

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


// ( SOLUZIONE 2013-01-30 #1
    p->me = new child;
    p->me->status = RUNNING;
    p->me->father = esecuzione;
    p->me->result = 0;
    p->me->next = esecuzione->children;
    esecuzione->children = p->me;
    p->children = nullptr;
//  SOLUZIONE 2013-01-30 )
// ( SOLUZIONE 2013-01-30
extern "C" void c_join() {
    if (!esecuzione->children) {
        flog(LOG_WARN, "join senza figli");
        c_abort_p();
        return;
    }
    for (child** c = &esecuzione->children; *c; c = &(*c)->next) {
        child* w = *c;
        if (w->status == TERMINATED) {
            esecuzione->conto[I_RAX] = w->result;
            *c = w->next;
            delete w;
            return;
        }
    }
    esecuzione->waiting = true;
    schedulatore();
}
//  SOLUZIONE 2013-01-30 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

12 giugno 2013

1. Un mutex con *priority inheritance* (*pim*) è un semaforo di mutua esclusione che tiene conto delle priorità dei processi per evitare il fenomeno dell'*inversione di priorità*: se un processo ad altra priorità P_1 è bloccato in attesa di acquisire la mutua esclusione su una risorsa posseduta da un processo a bassa priorità P_2 , non vogliamo che processi a priorità intermedia tra P_1 e P_2 interrompano P_2 , perché questo allungherebbe ingiustamente il tempo di attesa di P_1 . I pim risolvono questo problema facendo in modo che il processo che possiede la mutua esclusione innalzi la propria priorità, ponendola uguale alla maggiore tra le priorità dei processi in attesa sulla stessa risorsa e la sua. L'innalzamento di priorità è temporaneo: quando il processo libera la risorsa, ritorna alla sua priorità originaria.

Per realizzare i pim definiamo la seguente struttura (file `sistema.cpp`):

```
struct des_pim {
    natl orig_prio;
    des_proc *owner;
    des_proc *waiting;
};
```

Il campo `owner` punta al processo che possiede la mutua esclusione (0 se la risorsa è libera). Il campo `orig_prio` memorizza la priorità originaria del processo che possiede la mutua esclusione. Il campo `waiting` è una lista di processi in attesa di acquisire la mutua esclusione.

Le seguenti primitive, accessibili dal livello utente, operano sui pim. In caso di errore abortiscono il processo.

- `natl pim_init()` (già realizzata): inizializza un nuovo pim e ne restituisce l'identificatore. Se non è possibile creare un nuovo pim restituisce 0xFFFFFFFF.
- `void pim_wait(natl pim)` (da realizzare): tenta di acquisire la mutua esclusione sul pim di identificatore `pim`.
- `void pim_signal(natl pim)` (da realizzare): rilascia la mutua esclusione sul pim di identificatore `pim`.

Nota: Si assume che i processi non acquisiscano mai più di una mutua esclusione per volta.

Attenzione: poichè i pim cambiano dinamicamente la priorità di processi mentre questi possono trovarsi già in qualche coda, succede che le code dei processi non sono più ordinate per priorità. È dunque necessario modificare la funzione `rimozione_lista()` in modo che non si limiti ad estrarre dalla testa, ma cerchi il processo a maggiore priorità tra tutti quelli in lista.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti. Gestire correttamente la preemption.

```

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


// ( SOLUZIONE 2013-06-12
extern "C" void c_pim_wait(natl pim)
{
    if (!pim_valido(pim)) {
        flog(LOG_WARN, "pim non valido: %d", pim);
        c_abort_p();
        return;
    }

    des_pim* d = &array_despim[pim];

    if (d->owner) {
        if (d->owner->precedenza < esecuzione->precedenza)
            d->owner->precedenza = esecuzione->precedenza;
        inserimento_lista(d->waiting, esecuzione);
        schedulatore();
    } else {
        d->owner = esecuzione;
        d->orig_prio = esecuzione->precedenza;
    }
}

extern "C" void c_pim_signal(natl pim)
{
    if (!pim_valido(pim)) {
        flog(LOG_WARN, "pim non valido: %d", pim);
        c_abort_p();
        return;
    }

    des_pim* d = &array_despim[pim];

    esecuzione->precedenza = d->orig_prio;
    inserimento_lista(pronti, esecuzione);
    if (d->waiting) {
        des_proc *work = rimozione_lista(d->waiting);
        d->owner = work;
        d->orig_prio = work->precedenza;
        inserimento_lista(pronti, work);
    } else {
        d->owner = nullptr;
        d->orig_prio = 0;
    }
    schedulatore();
}
// SOLUZIONE 2013-06-12 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

2 luglio 2013

1. Un mutex con *priority inheritance* (*pim*) è un semaforo di mutua esclusione che tiene conto delle priorità dei processi per evitare il fenomeno dell'*inversione di priorità*: se un processo ad altra priorità P_1 è bloccato in attesa di acquisire la mutua esclusione su una risorsa posseduta da un processo a bassa priorità P_2 , non vogliamo che processi a priorità intermedia tra P_1 e P_2 interrompano P_2 , perché questo allungherebbe ingiustamente il tempo di attesa di P_1 . I pim risolvono questo problema facendo in modo che il processo che possiede la mutua esclusione innalzi la propria priorità, ponendola uguale alla maggiore tra le priorità dei processi in attesa sulla stessa risorsa e la sua. L'innalzamento di priorità è temporaneo: quando il processo libera la risorsa, ritorna alla sua priorità originaria.

Se un processo può acquisire più di una risorsa per volta, dobbiamo tenere conto di alcune complicazioni:

- Un processo P_1 può essere in possesso di una risorsa R_1 e contemporaneamente essere in attesa di acquisire una risorsa R_2 . Se innalziamo la priorità di P_1 dobbiamo anche innalzare la priorità del processo P_2 che possiede R_2 . A sua volta P_2 potrebbe essere in attesa di una risorsa R_3 , quindi dobbiamo anche innalzare la priorità del processo P_3 che possiede R_3 , e così via.
- Se un processo che possiede più risorse ne rilascia una, non deve ritornare alla sua priorità originaria, ma alla massima priorità richiesta dalle risorse che ancora possiede.

Per realizzare i pim definiamo la seguente struttura (file `sistema.cpp`):

```
struct des_pim {  
    natl curr_prio;  
    des_proc *owner;  
    des_proc *waiting;  
    des_pim *prec;  
};
```

Il campo `curr_prio` punta alla priorità massima dei processi in attesa di acquisire la risorsa (0 se non ve ne sono). Il campo `owner` punta al processo che possiede la risorsa (0 se la risorsa è libera). Il campo `waiting` è una lista di processi in attesa di acquisire la risorsa. Il campo `prec` serve a costruire una lista delle risorse possedute dall'attuale owner del pim (0 se la risorsa è libera). Le risorse sono in lista in ordine di acquisizione, con la più recente in testa.

Aggiungiamo inoltre i seguenti campi al descrittore di processo:

```
natl orig_prio;  
des_pim *owner;  
des_pim *waiting;
```

Il campo `orig_prio` contiene la priorità originaria del processo. Il campo `owner` punta all'ultimo pim acquisito dal processo (0 se il processo non possiede nessun pim). Il campo `waiting` punta al pim su cui il processo è in attesa (0 se non è in attesa su alcun pim).

Le seguenti primitive, accessibili dal livello utente, operano sui pim. In caso di errore abortiscono il processo.

- natl `pim_init()` (inizializza un nuovo pim e ne restituisce l'identificatore. Se non è possibile creare un nuovo pim restituisce 0xFFFFFFFF.
- void `pim_wait(natl pim)` (da realizzare): tenta di acquisire la mutua esclusione sul pim di identificatore `pim`. È un errore tentare di acquisire un pim che si possiede già.
- void `pim_signal(natl pim)` (già realizzata): rilascia la mutua esclusione sul pim di identificatore `pim`. È un errore tentare di rilasciare un pim diverso dall'ultimo acquisito.

Attenzione: poichè i pim cambiano dinamicamente la priorità di processi mentre questi possono trovarsi già in qualche coda, succede che le code dei processi non sono più ordinate per priorità. È dunque necessario modificare la funzione `rimozione_lista()` in modo che non si limiti ad estrarre dalla testa, ma cerchi il processo a maggiore priorità tra tutti quelli in lista.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti. Gestire correttamente la preemption.

```

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


// ( SOLUZIONE 2013-07-02

void promote(des_pim *d, natl prio)
{
    do {
        if (prio <= d->curr_prio)
            break;
        d->curr_prio = prio;
        if (d->owner->precedenza >= prio)
            break;
        d->owner->precedenza = prio;
        d = d->owner->waiting;
    } while (d);
}

extern "C" void c_pim_wait(natl pim)
{
    if (!pim_valido(pim)) {
        flog(LOG_WARN, "pim non valido: %d", pim);
        c_abort_p();
        return;
    }

    des_pim* d = &array_despim[pim];

    if (d->owner == esecuzione) {
        flog(LOG_WARN, "pim_wait(%d) non valida", pim);
        c_abort_p();
        return;
    }

    if (d->owner) {
        promote(d, esecuzione->precedenza);
        inserimento_lista(d->waiting, esecuzione);
        esecuzione->waiting = d;
        schedulatore();
    } else {
        d->owner = esecuzione;
        d->prec = esecuzione->owner;
        esecuzione->owner = d;
    }
}

// SOLUZIONE 2013-07-02 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

23 luglio 2013

1. Un mutex con *priority inheritance* (*pim*) è un semaforo di mutua esclusione che tiene conto delle priorità dei processi per evitare il fenomeno dell'*inversione di priorità*: se un processo ad altra priorità P_1 è bloccato in attesa di acquisire la mutua esclusione su una risorsa posseduta da un processo a bassa priorità P_2 , non vogliamo che processi a priorità intermedia tra P_1 e P_2 interrompano P_2 , perché questo allungherebbe ingiustamente il tempo di attesa di P_1 . I pim risolvono questo problema facendo in modo che il processo che possiede la mutua esclusione innalzi la propria priorità, ponendola uguale alla maggiore tra le priorità dei processi in attesa sulla stessa risorsa e la sua. L'innalzamento di priorità è temporaneo: quando il processo libera la risorsa, ritorna alla sua priorità originaria.

Per realizzare i pim definiamo la seguente struttura (file `sistema.cpp`):

```
struct des_pim {
    natl curr_prio;
    des_proc *owner;
    des_proc *waiting;
    des_pim *prec;
};
```

Il campo `curr_prio` punta alla priorità massima dei processi in attesa di acquisire la risorsa (0 se non ve ne sono). Il campo `owner` punta al processo che possiede la risorsa (0 se la risorsa è libera). Il campo `waiting` è una lista di processi in attesa di acquisire la risorsa. Il campo `prec` serve a costruire una lista delle risorse possedute dall'attuale owner del pim (0 se la risorsa è libera). Le risorse sono in lista in ordine di acquisizione, con la più recente in testa.

Aggiungiamo inoltre i seguenti campi al descrittore di processo:

```
natl orig_prio;
des_pim *owner;
des_pim *waiting;
```

Il campo `orig_prio` contiene la priorità originaria del processo. Il campo `owner` punta all'ultimo pim acquisito dal processo (0 se il processo non possiede nessun pim). Il campo `waiting` punta al pim su cui il processo è in attesa (0 se non è in attesa su alcun pim).

Le seguenti primitive, accessibili dal livello utente, operano sui pim. In caso di errore abortiscono il processo.

- `natl pim_init()` (già realizzata): inizializza un nuovo pim e ne restituisce l'identificatore. Se non è possibile creare un nuovo pim restituisce 0xFFFFFFFF.
- `void pim_wait(natl pim)` (già realizzata): tenta di acquisire la mutua esclusione sul pim di identificatore `pim`. È un errore tentare di acquisire un pim che si possiede già.

- `void pim_signal(natl pim)` (da realizzare): rilascia la mutua esclusione sul pim di identificatore `pim`. È un errore tentare di rilasciare un pim che non si possiede. Si noti che se un processo che possiede più risorse ne rilascia una, non deve ritornare alla sua priorità originaria, ma alla massima priorità richiesta dalle risorse che ancora possiede. **Attenzione:** un processo può rilasciare uno qualunque dei pim che possiede, non necessariamente l'ultimo acquisito.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti. Gestire correttamente la preemption.

```

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


// ( SOLUZIONE 2013-07-23

void demote(des_proc *p)
{
    p->precedenza = p->orig_prio;
    for (des_pim *d = p->owner; d; d = d->prec)
        if (d->curr_prio > p->precedenza)
            p->precedenza = d->curr_prio;
}

bool estrai_pim(des_pim*& h, des_pim* p)
{
    for (des_pim** ps = &h; *ps; ps = &((*ps)->prec))
        if (*ps == p) {
            *ps = p->prec;
            return true;
        }
    return false;
}

extern "C" void c_pim_signal(natl pim)
{
    if (!pim_valido(pim)) {
        flog(LOG_WARN, "pim non valido: %d", pim);
        c_abort_p();
        return;
    }

    des_pim* d = &array_despim[pim];

    if (!estrai_pim(esecuzione->owner, d)) {
        flog(LOG_WARN, "pim_signal(%d) non valida", pim);
        c_abort_p();
        return;
    }

    demote(esecuzione);
    if (d->waiting) {
        des_proc *work = rimozione_lista(d->waiting);
        d->curr_prio = maxprior(d->waiting);
        d->owner = work;
        d->prec = work->owner;
        work->owner = d;
        work->waiting = nullptr;
        inspronti();
        inserimento_lista(pronti, work);
        schedulatore();
    } else {
        d->owner = nullptr;
        d->curr_prio = 0;
        d->prec = nullptr;
    }
}
// SOLUZIONE 2013-07-23 )

```


Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

12 settembre 2013

1. Un modo per evitare il problema del *blocco critico* nell'utilizzo dei semafori (di mutua esclusione) è di fare in modo che ogni processo acquisisca in una sola operazione indivisibile tutti i semafori di cui ha bisogno. Se qualche semaforo non può essere acquisito, allora non deve esserne acquisito nessuno: il processo si deve bloccare fino a quando tutti diventano disponibili.

Per supportare questo meccanismo, anche se in forma limitata, aggiungiamo al nucleo la primitiva

```
void sem_multivait(int sem1, int sem2)
```

che acquisisce i semafori `sem1` e `sem2` in modo indivisibile: se `sem1` e `sem2` possono essere acquisiti senza bloccarsi, allora vengono acquisiti entrambi. Altrimenti i contatori non vengono modificati e il processo viene sospeso in attesa che entrambi diventino acquisibili.

Si noti che, se sia `sem1` che `sem2` non sono acquisibili, dovremmo inserire il processo che ha invocato la primitiva in due code. Poichè non possiamo farlo, adottiamo la seguente tecnica. Blocchiamo il processo in ogni caso su uno solo dei due semafori (quello non acquisibile se ve ne è uno solo, oppure uno qualunque se entrambi non sono acquisibili) e aggiungiamo il campo

```
des_sem *other_sem
```

al descrittore di processo. Utilizziamo questo campo per memorizzare il puntatore all'altro semaforo (quello su cui non stiamo bloccando il processo). Modifichiamo quindi la primitiva `sem_signal` in modo che, quando deve svegliare un processo, controlli il valore di questo campo nel suo descrittore. Se non è nullo, la `sem_signal` si deve preoccupare di completare le operazioni della `sem_multivait` per conto del processo svegliato. Questo può anche comportare che il processo debba essere sospeso nuovamente, se accade che `other_sem` non è acquisibile. Si noti che in questo caso il processo dovrà essere sospeso su `other_sem`, in quanto il semaforo corrente è diventato acquisibile per effetto della `sem_signal`.

Modificare i file `sistema.cpp` e `sistema.s` completando le parti mancanti.

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

```
// ( SOLUZIONE 2013-09-12
    des_sem *s1, *s2;

    if (!sem_valido(sem1)) {
        flog(LOG_WARN, "semaforo errato: %d", sem1);
        c_abort_p();
        return;
    }
    s1 = &array_dess[sem1];

    if (!sem_valido(sem2)) {
        flog(LOG_WARN, "semaforo errato: %d", sem2);
        c_abort_p();
        return;
    }
    s2 = &array_dess[sem2];

    if (s1->counter > 0 && s2->counter > 0) {
        s1->counter--;
        s2->counter--;
        return;
    }
    if (s1->counter <= 0) {
        inserimento_lista(s1->pointer, esecuzione);
        esecuzione->other_sem = s2;
    } else {
        inserimento_lista(s2->pointer, esecuzione);
        esecuzione->other_sem = s1;
    }
    schedulatore();
// SOLUZIONE 2013-09-12 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

13 gennaio 2014

1. Un modo per evitare il problema del *blocco critico* nell'utilizzo dei semafori (di mutua esclusione) è di fare in modo che ogni processo acquisisca in una sola operazione indivisibile tutti i semafori di cui ha bisogno. Se qualche semaforo non può essere acquisito, allora non deve esserne acquisito nessuno: il processo si deve bloccare fino a quando tutti diventano disponibili.

Per supportare questo meccanismo, anche se in forma limitata, aggiungiamo al nucleo la primitiva

```
void sem_multivait(int sem1, int sem2)
```

che acquisisce i semafori `sem1` e `sem2` in modo indivisibile: se `sem1` e `sem2` possono essere acquisiti senza bloccarsi, allora vengono acquisiti entrambi. Altrimenti i contatori non vengono modificati e il processo viene sospeso in attesa che entrambi diventino acquisibili.

Si noti che, se sia `sem1` che `sem2` non sono acquisibili, dovremmo inserire il processo che ha invocato la primitiva in due code. Poichè non possiamo farlo, adottiamo la seguente tecnica. Blocchiamo il processo in ogni caso su uno solo dei due semafori (quello non acquisibile se ve ne è uno solo, oppure uno qualunque se entrambi non sono acquisibili) e aggiungiamo il campo

```
des_sem *other_sem
```

al descrittore di processo. Utilizziamo questo campo per memorizzare il puntatore all'altro semaforo (quello su cui non stiamo bloccando il processo). Modifichiamo quindi la primitiva `sem_signal` in modo che, quando deve svegliare un processo, controlli il valore di questo campo nel suo descrittore. Se non è nullo, la `sem_signal` si deve preoccupare di completare le operazioni della `sem_multivait` per conto del processo svegliato. Questo può anche comportare che il processo debba essere sospeso nuovamente, se accade che `other_sem` non è acquisibile. Si noti che in questo caso il processo dovrà essere sospeso su `other_sem`, in quanto il semaforo corrente è diventato acquisibile per effetto della `sem_signal`.

Modificare i file `sistema.cpp` e `sistema.s` completando le parti mancanti.

Gestire correttamente eventuali *preemption*.

```
*****
* sistema/sistema.cpp
*****  
  
// ( SOLUZIONE 2014-01-13 #1  
    des_sem* s = &array_dess[sem];  
    (s->counter)++;  
  
    if (!s->pointer)  
        return;  
  
    inspronti();      // preemption  
    des_proc *lavoro = rimozione_lista(s->pointer);  
    des_sem *os = lavoro->other_sem;  
    if (!os) {  
        inserimento_lista(pronti, lavoro);  
    } else if (os->counter > 0) {  
        s->counter--;  
        os->counter--;  
        lavoro->other_sem = nullptr;  
        inserimento_lista(pronti, lavoro);  
    } else {  
        inserimento_lista(os->pointer, lavoro);  
        lavoro->other_sem = s;  
    }  
    schedulatore();  
//  SOLUZIONE 2014-01-13 )  
// ( SOLUZIONE 2014-01-13 #2  
    p->other_sem = 0;  
//  SOLUZIONE 2014-01-13 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

30 gennaio 2014

1. Un modo per evitare il problema del *blocco critico* nell'utilizzo dei semafori (di mutua esclusione) è di fare in modo che ogni processo acquisisca in una sola operazione indivisibile tutti i semafori di cui ha bisogno. Se qualche semaforo non può essere acquisito, allora non deve esserne acquisito nessuno: il processo si deve bloccare fino a quando tutti diventano disponibili.

Per supportare questo meccanismo aggiungiamo al nucleo le seguenti primitive

```
void sem_add(natl sem)
void sem_del(natl sem)
void sem_multiwait()
```

Con le prime due primitive, un processo può costruire una lista di semafori che vuole acquisire contemporaneamente. La primitiva `sem_multiwait()` provvede poi ad aquisire in modo indivisibile tutti i semafori che si trovano nella lista al momento della sua invocazione: se tutti i semafori nella lista possono essere acquisiti senza bloccarsi, allora vengono acquisiti tutti. Altrimenti i contatori non vengono modificati e il processo viene sospeso in attesa che tutti diventino acquisibili.

Si noti che, nel caso in cui non tutti i semafori siano acquisibili, dovremo comunque inserire il processo nella coda di uno solo di essi.

Al fine di realizzare questo meccanismo, aggiungiamo i seguenti campi al descrittore di processo:

```
des_sem *mysem[MAX_MULTISEM];
natl nextsem;
bool multiwait;
```

I semafori della lista richiesta dal processo si troveranno nelle prime `nextsem` posizioni dell'array `mysem`. Il campo `multiwait` ci permette di sapere se il processo si è bloccato per via di una `sem_multiwait()` (`multiwait==true`) o di una comune `sem_wait()` (`multiwait==false`). Modifichiamo quindi la primitiva `sem_signal` in modo che, quando deve svegliare un processo, controlli il valore di questo campo nel suo descrittore. Se è `true`, la `sem_signal` si deve preoccupare di completare le operazioni della `sem_multiwait` per conto del processo svegliato. Questo può anche comportare che il processo debba essere sospeso nuovamente, se accade che non tutti i semafori della lista siano acquisibili. (se `multiwait==false`, la `sem_signal()` deve comportarsi normalmente).

Modificare i file `sistema.cpp` e `sistema.s` completando le parti mancanti.

Gestire correttamente eventuali *preemption*.

```

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


// ( SOLUZIONE 2014-01-30 #1
struct des_sem* sem_first_busy(des_proc *p)
{
    struct des_sem *s;
    for (natl i = 0; i < p->nextsem; i++) {
        s = p->mysem[i];

        if (s->counter <= 0)
            return s;
    }
    return 0;
}

extern "C" void c_sem_multiwait()
{
    struct des_sem *s = sem_first_busy(esecuzione);

    if (!s) {
        for (natl i = 0; i < esecuzione->nextsem; i++) {
            s = esecuzione->mysem[i];
            s->counter--;
        }
        return;
    }
    esecuzione->multiwait = true;
    inserimento_lista(s->pointer, esecuzione);
    schedulatore();
}
// SOLUZIONE 2014-01-30 )
// ( SOLUZIONE 2014-01-30 #2
des_sem *s = &array_dess[sem];
s->counter++;

if (!s->pointer)
    return;

inspronti();      // preemption
des_proc *p = rimozione_lista(s->pointer);
if (!p->multiwait) {
    inserimento_lista(pronti, p);
} else {
    struct des_sem *os = sem_first_busy(p);
    if (!os) {
        for (natl i = 0; i < p->nextsem; i++) {
            os = p->mysem[i];
            os->counter--;
        }
        p->multiwait = false;
        inserimento_lista(pronti, p);
    } else {
        inserimento_lista(os->pointer, p);
    }
}
schedulatore();
// SOLUZIONE 2014-01-30 )

```


Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

11 giugno 2014

1. Una *barriera* è un meccanismo di sincronizzazione tra processi con la seguente proprietà: ogni processo che arriva alla barriera si deve bloccare fino a quando non sono arrivati alla barriera tutti i processi.

Vogliamo realizzare il meccanismo della barriera nel nucleo, ma solo tra i processi che si sono preventivamente registrati. Per questo scopo aggiungiamo le seguenti primitive:

- **void reg()** (da realizzare): registra il processo corrente sulla barriera. È un errore se un processo tenta di registrarsi due volte, oppure se tenta di registrarsi dopo che un qualunque processo è già giunto alla barriera.
- **void barrier()** (da realizzare): informa il sistema che il processo corrente è giunto alla barriera. È un errore se un processo invoca questa primitiva senza essersi registrato.

Le primitive abortiscono il processo chiamante in caso di errore e tengono conto della priorità tra i processi.

Modificare i file **sistema.cpp** e **sistema.s** in modo da realizzare le primitive appena descritte. Può essere necessario definire nuove strutture dati e aggiungere campi al descrittore di processo.

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

```
// ( SOLUZIONE 2014-06-11
    carica_gateTIPO_R           a_reg      LIV_UTENTE
    carica_gateTIPO_B           a_barrier  LIV_UTENTE
//  SOLUZIONE 2014-06-11 )
// ( SOLUZIONE 2014-06-11

a_reg:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_reg
    call carica_stato
    iretq
    .cfi_endproc

a_barrier:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_barrier
    call carica_stato
    iretq
    .cfi_endproc
//  SOLUZIONE 2014-06-11 )
```

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

```
// ( SOLUZIONE 2014-06-11
    bool barrier_reg;
//  SOLUZIONE 2014-06-11 )
// ( SOLUZIONE 2014-06-11
struct barrier_t {
    bool started;
    int registered;
    int nwaiting;
    des_proc* waiting;
};

barrier_t global_barrier = { false, 0, 0, nullptr };

extern "C" void c_reg()
{
    barrier_t *b = &global_barrier;

    if (b->started) {
        flog(LOG_WARN, "registrazione in ritardo");
        c_abort_p();
        return;
    }
```

```

if (esecuzione->barrier_reg) {
    flog(LOG_WARN, "doppia registrazione");
    c_abort_p();
    return;
}
b->registered++;
esecuzione->barrier_reg = true;
}

extern "C" void c_barrier()
{
    barrier_t *b = &global_barrier;

    if (!esecuzione->barrier_reg) {
        flog(LOG_WARN, "non registrato");
        c_abort_p();
        return;
    }
    b->started = true;
    b->nwaiting++;
    inserimento_lista(b->waiting, esecuzione);
    if (b->nwaiting == b->registered) {
        while (b->waiting) {
            des_proc* work = rimozione_lista(b->waiting);
            inserimento_lista(pronti, work);
        }
        b->nwaiting = 0;
    }
    schedulatore();
}
// SOLUZIONE 2014-06-11 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

2 luglio 2014

1. Una *barriera* è un meccanismo di sincronizzazione tra processi che funziona nel modo seguente: la barriera è normalmente chiusa; se un processo arriva alla barriera e la trova chiusa, si blocca; la barriera si apre solo quando sono arrivati tutti i processi, che a quel punto si sbloccano; una volta aperta e sbloccata tutti i processi, la barriera si richiude e il meccanismo si ripete.

Vogliamo realizzare il meccanismo della barriera nel nucleo, ma solo tra i processi che si sono preventivamente registrati. I processi possono registrarsi e deregistrarsi per una barriera in qualunque momento. La deregistrazione di un processo può causare l'apertura di una barriera, se tutti gli altri processi registrati erano già arrivati.

Per realizzare il meccanismo aggiungiamo le seguenti primitive:

- **void reg()** (da realizzare): registra il processo corrente sulla barriera. È un errore se un processo tenta di registrarsi due volte.
- **void dereg()** (da realizzare): deregistra il processo corrente sulla barriera. È un errore se un processo tenta di deregistrarsi senza essere registrato.
- **void barrier()** (da realizzare): fa giungere il processo corrente alla barriera. È un errore se un processo invoca questa primitiva senza essersi registrato.

Le primitive abortiscono il processo chiamante in caso di errore e tengono conto della priorità tra i processi.

Modificare i file **sistema.cpp** e **sistema.s** in modo da realizzare le primitive appena descritte. Può essere necessario definire nuove strutture dati e aggiungere campi al descrittore di processo.

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

```
// ( SOLUZIONE 2014-07-02
    carica_gateTIPO_R           a_reg      LIV_UTENTE
    carica_gateTIPO_DR          a_dereg    LIV_UTENTE
    carica_gateTIPO_B           a_barrier  LIV_UTENTE
//  SOLUZIONE 2014-07-02 )
// ( SOLUZIONE 2014-07-02

a_reg:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_reg
    call carica_stato
    iretq
    .cfi_endproc

a_dereg:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_dereg
    call carica_stato
    iretq
    .cfi_endproc

a_barrier:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_barrier
    call carica_stato
    iretq
    .cfi_endproc
//  SOLUZIONE 2014-07-02 )

*****
```

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

```
// ( SOLUZIONE 2014-07-02
    bool barrier_reg;
//  SOLUZIONE 2014-07-02 )
// ( SOLUZIONE 2014-07-02
struct barrier_t {
    int registered;
    int nwaiting;
    des_proc *waiting;
};
```

```

barrier_t global_barrier = { 0, 0, 0 };

extern "C" void c_reg()
{
    barrier_t *b = &global_barrier;

    if (esecuzione->barrier_reg) {
        flog(LOG_WARN, "reg: doppia registrazione");
        c_abort_p();
        return;
    }
    b->registered++;
    esecuzione->barrier_reg = true;
}

void check_barrier()
{
    barrier_t *b = &global_barrier;

    if (b->nwaiting == b->registered) {
        while (b->waiting) {
            des_proc *work = rimozione_lista(b->waiting);
            inserimento_lista(pronti, work);
        }
        b->nwaiting = 0;
    }
}

extern "C" void c_dereg()
{
    barrier_t *b = &global_barrier;

    if (!esecuzione->barrier_reg) {
        flog(LOG_WARN, "dereg: non registrato");
        c_abort_p();
        return;
    }
    b->registered--;
    esecuzione->barrier_reg = false;
    inserimento_lista(pronti, esecuzione);
    check_barrier();
    schedulatore();
}

extern "C" void c_barrier()
{
    barrier_t *b = &global_barrier;

    if (!esecuzione->barrier_reg) {
        flog(LOG_WARN, "non registrato");
        c_abort_p();
        return;
    }
    b->nwaiting++;
    inserimento_lista(b->waiting, esecuzione);
    check_barrier();
    schedulatore();
}
// SOLUZIONE 2014-07-02 )

```


Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

23 luglio 2014

1. Una *barriera* è un meccanismo di sincronizzazione tra processi che funziona nel modo seguente: la barriera è normalmente chiusa; se un processo arriva alla barriera e la trova chiusa, si blocca; la barriera si apre solo quando sono arrivati tutti i processi, che a quel punto si sbloccano; una volta aperta e sbloccata tutti i processi, la barriera si richiude e il meccanismo si ripete.

Vogliamo realizzare il meccanismo della barriera nel nucleo, prevedendo la possibilità che esistano più barriere distinte, ciascuna con un proprio identificatore. Le barriere sono create dinamicamente, specificando il numero di processi che vi si devono sincronizzare. L'operazione di creazione restituisce l'identificatore della nuova barriera. Un processo si sincronizza su una data barriera specificandone l'identificatore.

Prevediamo anche la possibilità di distruggere una barriera esistente. Tutti i processi eventualmente bloccati sulla barriera distrutta devono essere risvegliati.

Per rappresentare una barriera introduciamo la seguente struttura dati:

```
struct barrier {  
    natl nproc;  
    natl narrived;  
    des_proc *waiting;  
    barrier *next;  
    barrier *prec;  
    natl id;  
};
```

Dove: `nproc` è il numero di processi che devono sincronizzarsi sulla barriera; `narrived` conta i processi arrivati alla barriera dall'ultima apertura (o dalla creazione, quando la barriera non è ancora mai stata aperta); `waiting` è la coda dei processi che attendono l'apertura della barriera; `next` e `prec` sono usati per realizzare la lista di tutte le barriere esistenti; `id` è l'identificatore della barriera.

Aggiungiamo inoltre le seguenti primitive:

- `natl barrier_create(natl nproc)` (già realizzata): crea una nuova barriera che sincronizza `nproc` processi e ne restituisce l'identificatore (0xFFFFFFFF se non è stato possibile completare l'operazione).
- `bool barrier(natl id)` (da realizzare): fa giungere il processo corrente alla barriera di identificatore `id`. È un errore se tale barriera non esiste. Restituisce `true` quando termina normalmente, e `false` quando termina perché la barriera è stata distrutta mentre il processo era in attesa dell'apertura.
- `void barrier_destroy(natl id)` (da realizzare): distrugge la barriera di identificatore `id`. È un errore se tale barriera non esiste.

Le primitive abortiscono il processo chiamante in caso di errore e tengono conto della priorità tra i processi.

Modificare i file **sistema.cpp** e **sistema.s** in modo da realizzare le primitive mancanti.

```

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


// ( SOLUZIONE 2014-07-23
    carica_gateTIPO_BD           a_barrier_destroy      LIV_UTENTE
    carica_gateTIPO_B            a_barrier             LIV_UTENTE
//  SOLUZIONE 2014-07-23 )
// ( SOLUZIONE 2014-07-23

a_barrier_destroy:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_barrier_destroy
    call carica_stato
    iretq
    .cfi_endproc

a_barrier:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_barrier
    call carica_stato
    iretq
    .cfi_endproc
//  SOLUZIONE 2014-07-23 )

*****


* sistema/sistema.cpp
*****


// ( SOLUZIONE 2014-07-23
barrier_t *barrier_find(natl id)
{
    for (barrier_t *scan = barriers; scan; scan = scan->next) {
        if (scan->id == id)
            return scan;
    }
    return 0;
}

extern "C" void c_barrier(natl id)
{
    barrier_t *b = barrier_find(id);

    if (!b) {
        flog(LOG_WARN, "barriera inesistente: %d", id);
        c_abort_p();
        return;
    }

    inserimento_lista(b->waiting, esecuzione);
    b->narrived++;
}

```

```

    if (b->narrived == b->nproc) {
        while (b->waiting) {
            des_proc *work = rimozione_lista(b->waiting);
            work->contesto[I_RAX] = (natq)true;
            inserimento_lista(pronti, work);
        }
        b->narrived = 0;
    }
    schedulatore();
}

extern "C" void c_barrier_destroy(natl id)
{
    barrier_t *b = barrier_find(id);

    if (!b) {
        flog(LOG_WARN, "barriera inesistente: %d", id);
        c_abort_p();
        return;
    }

    if (b->next)
        b->next->prec = b->prec;
    if (b->prec)
        b->prec->next = b->next;
    else
        barriers = b->next;
    b->next = b->prec = nullptr;

    inserimento_lista(pronti, esecuzione);
    while (b->waiting) {
        des_proc *work = rimozione_lista(b->waiting);
        work->contesto[I_RAX] = (natq)false;
        inserimento_lista(pronti, work);
    }
    schedulatore();
    dealloca(b);
}
// SOLUZIONE 2014-07-23 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

17 settembre 2014

1. Una *barriera* è un meccanismo di sincronizzazione tra processi che funziona nel modo seguente: la barriera è normalmente chiusa; se un processo arriva alla barriera si blocca; la barriera si apre solo quando sono arrivati tutti i processi registrati su quella barriera. Quando la barriera si apre tutti i processi si sbloccano, la barriera si richiude (istantaneamente) e il meccanismo si ripete.

Vogliamo realizzare il meccanismo della barriera nel nucleo, prevedendo la possibilità che esistano più barriere distinte, identificate con un numero da 0 a *MAX_BARRIERS* – 1. In ogni istante, ogni processo è registrato su al più una barriera e può sincronizzarsi solo sulla barriera su cui è eventualmente registrato. La registrazione di un processo su una barriera può essere richiesta da un qualunque processo, non necessariamente coincidente con il processo da registrare, e può essere cambiata in un qualunque momento, anche quando il processo da registrare è già bloccato su un'altra barriera. In quest'ultimo caso il processo viene spostato sulla nuova barriera, come se si fosse sincronizzato su quest'ultima invece che sulla precedente.

Le barriere sono create all'avvio del sistema e non possono essere distrutte.

Per rappresentare una barriera introduciamo le seguenti strutture dati:

```
struct barrier {
    natl nproc;
    natl narrived;
    des_proc *waiting;
};

barrier barriers[MAX_BARRIERS];
```

Dove: *nproc* è il numero di processi registrati sulla barriera; *narrived* conta i processi arrivati alla barriera dall'ultima apertura (o dalla creazione, quando la barriera non è ancora mai stata aperta); *waiting* è la coda dei processi che attendono l'apertura della barriera. L'array *barriers* contiene tutte le barriere del sistema. L'identificatore di una barriera è il suo indice all'interno dell'array.

Aggiungiamo inoltre i seguenti campi al descrittore di processo:

```
natl barrier_id;
bool waiting;
```

Dove: *barrier_id* è l'identificatore della barriera su cui il processo è registrato (0xFFFFFFFF se nessuna); *waiting* è *true* se e solo se il processo è bloccato sulla barriera.

Aggiungiamo infine le seguenti primitive:

- **void barrier()** (da realizzare): fa giungere il processo corrente alla barriera su cui è registrato. È un errore se il processo non è registrato su alcuna barriera.

- `bool reg(natl pid, natl bid)` (da realizzare): registra il processo di identificatore `pid` sulla barriera di identificatore `bid`, o su nessuna barriera se `bid` è `0xFFFFFFFF`. È un errore tentare di registrare il processo su una barriera che non esiste. La primitiva restituisce `false` se il processo non esiste, e `true` in tutti gli altri casi.

Le primitive abortiscono il processo chiamante in caso di errore e tengono conto della priorità tra i processi.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti.

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

```
// ( SOLUZIONE 2014-09-17
void maybe_open_barrier(barrier_t *b)
{
    if (b->narrived == b->nproc) {
        while (b->waiting) {
            des_proc *work = rimozione_lista(b->waiting);
            work->waiting = false;
            inserimento_lista(pronti, work);
        }
        b->narrived = 0;
    }
}

extern "C" void c_barrier()
{
    natl bid = esecuzione->barrier_id;

    if (bid == 0xFFFFFFFF) {
        flog(LOG_WARN, "processo non registrato");
        c_abort_p();
        return;
    }

    barrier_t *b = &barriers[bid];

    inserimento_lista(b->waiting, esecuzione);
    b->narrived++;
    esecuzione->waiting = true;
    maybe_open_barrier(b);
    schedulatore();
}

extern "C" void c_reg(natl pid, natl bid)
{
    if (bid >= MAX_BARRIERS && bid != 0xFFFFFFFF) {
        flog(LOG_WARN, "barriera inesistente: %d", bid);
        c_abort_p();
        return;
    }
    if (pid >= MAX_BARRIERS) {
        flog(LOG_WARN, "pid non valido: %d", pid);
        c_abort_p();
        return;
    }
    des_proc *p = des_p(pid);

    if (!p) {
        esecuzione->contesto[I_RAX] = (natq)false;
        return;
    }
    esecuzione->contesto[I_RAX] = (natq)true;

    if (p->barrier_id == bid)
        return;

    inspronti();
}
```

```

if (p->barrier_id != 0xFFFFFFFF) {
    barrier_t *b = &barriers[p->barrier_id];
    b->nproc--;
    if (p->waiting) {
        b->narrived--;
        rimozione_lista_id(b->waiting, pid);
    } else {
        maybe_open_barrier(b);
    }
}
if (bid != 0xFFFFFFFF) {
    barrier_t *b = &barriers[bid];
    b->nproc++;
    if (p->waiting) {
        inserimento_lista(b->waiting, p);
        b->narrived++;
        maybe_open_barrier(b);
    }
} else {
    if (p->waiting) {
        inserimento_lista(pronti, p);
        p->waiting = false;
    }
}
p->barrier_id = bid;
schedulatore();
}
// SOLUZIONE 2014-09-17 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

13 gennaio 2015

1. Introduciamo nel nucleo una versione semplificata del meccanismo dei socket per la comunicazione inter-processo, relativamente alla sola fase di connessione.

Un socket è un oggetto identificato tramite un numero naturale. È possibile creare una connessione tra due socket tramite le operazioni di `accept(nat1 id)` e `connect(nat1 src, nat1 dest)`. Un processo che esegue `accept()` su un socket s_1 si pone in attesa di richieste di connessione verso s_1 ; un processo può inviare una richiesta di connessione invocando `connect()` su un altro socket s_2 , specificando s_1 come destinazione. La `accept()` completa la connessione creando un nuovo socket, s_3 , e ne restituisce l'identificatore al chiamante. A questo punto i socket s_2 e s_3 sono connessi e possono essere usati per scambiare dati tra i processi (cosa che non realizziamo), mentre il socket s_1 è di nuovo disponibile per altri usi.

Su uno stesso socket ci può essere più di un processo in attesa di connessioni (più processi possono invocare `accept()` su uno stesso socket): in quel caso, le eventuali richieste verranno servite in base alla priorità dei processi in attesa. Per poter invocare `accept()`, il socket non deve però essere già connesso o essere sorgente di una richiesta di connessione in corso. Per poter invocare `connect()`, né il socket sorgente, né il socket di destinazione devono essere già connessi o sorgenti di altre richieste di connessione in corso; il socket destinazione deve essere in fase di accettazione, mentre il socket sorgente no.

Per realizzare questo meccanismo definiamo le seguenti strutture dati:

```
enum sock_state {
    SOCK_AVAIL,
    SOCK_ACCEPTING,
    SOCK_CONNECTING,
    SOCK_CONNECTED
};

struct des_sock {
    sock_state state;
    des_proc *connecting;
    des_proc *accepting;
};
```

I possibili stati di un socket sono i seguenti:

- `SOCK_AVAIL`: il socket non è al momento utilizzato;
- `SOCK_ACCEPTING`: c'è almeno un processo che sta accettando connessioni su questo socket;
- `SOCK_CONNECTING`: un processo sta tentando di connettere questo socket (come sorgente) ad un altro;
- `SOCK_CONNECTED`: il socket è connesso.

La lista `connecting` contiene i processi che stanno tentando di creare una connessione che ha questo socket come destinazione; la lista `accepting` contiene i processi che stanno accettando connessioni su questo socket.

Aggiungiamo inoltre le seguenti primitive (abortiscono il processo in caso di errore):

- `natl socket()` (già realizzata): crea un socket e ne restituisce l'identificatore (0xFFFFFFFF se non è stato possibile crearlo);
- `natl accept(natl id)` (da realizzare): pone il processo in attesa di connessioni sul socket `id`; restituisce 0xFFFFFFFF se il socket non è nello stato giusto; è un errore se il socket non esiste.
- `bool connect(natl src, natl dest)` (da realizzare): tenta di connettere il socket `src` con il socket `dest`; restituisce `false` se uno dei due socket non è nello stato giusto; è un errore se uno dei due socket non esiste.

Tenere conto di eventuali preemption. **Attenzione:** può essere necessario aggiungere informazioni ai descrittori di processo.

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

```
// ( SOLUZIONE 2015-01-13
    carica_gateTIPO_ACCEPTa_accept    LIV_UTENTE
    carica_gateTIPO_CONNECT          a_connect    LIV_UTENTE
//  SOLUZIONE 2015-01-13 )
// ( SOLUZIONE 2015-01-13

a_accept:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_accept
    call carica_stato
    iretq
    .cfi_endproc

a_connect:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_connect
    call carica_stato
    iretq
    .cfi_endproc
//  SOLUZIONE 2015-01-13 )
```

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

```
// ( SOLUZIONE 2015-01-13 #1
//  SOLUZIONE 2015-01-13 )
// ( SOLUZIONE 2015-01-13 #3
//  SOLUZIONE 2015-01-13 )
// ( SOLUZIONE 2015-01-13 #2
extern "C" void c_accept(natl id)
{
    if (!sock_valid(id)) {
        flog(LOG_WARN, "socket id %d non valido", id);
        c_abort_p();
        return;
    }

    des_sock *s = &array_des_sock[id];

    if (!sock_can_accept(s)) {
        flog(LOG_WARN, "accept non ammessa su socket %d", id);
        esecuzione->contesto[I_RAX] = 0xFFFFFFFF;
        return;
    }
    s->state = SOCK_ACCEPTING;
```

```

        inserimento_lista(s->accepting, esecuzione);
        schedulatore();
    }

extern "C" void c_connect(natl id, natl dest)
{
    if (!sock_valid(id)) {
        flog(LOG_WARN, "socket id %d non valido", id);
        c_abort_p();
        return;
    }

    if (!sock_valid(dest)) {
        flog(LOG_WARN, "socket id %d non valido", dest);
        c_abort_p();
        return;
    }

    des_sock *s = &array_des_sock[id];
    des_sock *as = &array_des_sock[dest];

    if (!sock_src_ok(s)) {
        flog(LOG_WARN, "connect non ammessa su socket %d", id);
        esecuzione->contesto[I_RAX] = (natq)false;
        return;
    }

    if (!sock_dest_ok(as)) {
        flog(LOG_WARN, "socket %d non puo' essere destinazione",
dest);
        esecuzione->contesto[I_RAX] = (natq)false;
        return;
    }

    s->state = SOCK_CONNECTING;

    des_proc *work = rimozione_lista(as->accepting);
    if (!as->accepting)
        as->state = SOCK_AVAIL;

    inspronti();
    inserimento_lista(pronti, work);

    natl c = c_socket();
    work->contesto[I_RAX] = c;
    if (c == 0xffffffff) {
        esecuzione->contesto[I_RAX] = (natq)false;
    } else {
        des_sock *cs = &array_des_sock[c];

        s->state = SOCK_CONNECTED;
        cs->state = SOCK_CONNECTED;

        esecuzione->contesto[I_RAX] = (natq)true;
    }
    schedulatore();
}
// SOLUZIONE 2015-01-13 )

```


Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

29 gennaio 2015

1. Introduciamo nel nucleo una versione semplificata del meccanismo dei socket per la comunicazione inter-processo, relativamente alla sola fase di connessione.

Un socket è un oggetto identificato tramite un numero naturale. È possibile creare una connessione tra due socket tramite le operazioni di `accept(nat1 id)` e `connect(nat1 src, nat1 dest)`.

Prima di poter eseguire `accept()` deve essere stata eseguita almeno una `listen()` sullo stesso socket. La primitiva `listen()`, inoltre, dichiara che il socket è disponibile a ricevere richieste di connessione come destinatario di una `connect()`.

Un processo che esegue `accept()` su un socket s_1 si pone in attesa di richieste di connessione verso s_1 ; un processo può inviare una richiesta di connessione invocando `connect()` su un altro socket s_2 , specificando s_1 come destinazione. La `accept()` completa la connessione creando un nuovo socket, s_3 , e ne restituisce l'identificatore al chiamante. A questo punto i socket s_2 e s_3 sono connessi e possono essere usati per scambiare dati tra i processi (cosa che non realizziamo), mentre il socket s_1 è di nuovo disponibile per altre connessioni.

Su uno stesso socket ci può essere più di un processo in attesa di connessioni (più processi possono invocare `accept()` su uno stesso socket): in quel caso, le eventuali richieste verranno servite in base alla priorità dei processi in attesa. Per poter invocare `accept()`, il socket non deve però essere già connesso o essere sorgente di una richiesta di connessione in corso. Per poter invocare `connect()`, né il socket sorgente, né il socket di destinazione devono essere già connessi o sorgenti di altre richieste di connessione in corso; il socket destinazione deve essere disponibile ad accettare connessioni (qualcuno deve avere eseguito `listen()` su di esso), mentre il socket sorgente no.

Per realizzare questo meccanismo definiamo le seguenti strutture dati:

```
enum sock_state {
    SOCK_AVAIL,
    SOCK_LISTENING,
    SOCK_ACCEPTING,
    SOCK_CONNECTING,
    SOCK_CONNECTED
};

struct des_sock {
    sock_state state;
    des_proc *connecting;
    des_proc *accepting;
};
```

I possibili stati di un socket sono i seguenti:

- `SOCK_AVAIL`: il socket non è al momento utilizzato;
- `SOCK_LISTENING`: il socket può essere usato da una `accept()` e può essere destinatario di `connect()`;

- `SOCK_ACCEPTING`: c'è almeno un processo che sta accettando connessioni su questo socket;
- `SOCK_CONNECTING`: un processo sta tentando di connettere questo socket (come sorgente) ad un altro;
- `SOCK_CONNECTED`: il socket è connesso.

La lista `connecting` contiene i processi che stanno tentando di creare una connessione che ha questo socket come destinazione; la lista `accepting` contiene i processi che stanno accettando connessioni su questo socket.

Aggiungiamo inoltre le seguenti primitive (abortiscono il processo in caso di errore):

- `natl socket()` (già realizzata): crea un socket e ne restituisce l'identificatore (0xFFFFFFFF se non è stato possibile crearlo);
- `bool listen(nat1 id)` (già realizzata): mette il socket in stato `SOCK_LISTENING`, se possibile. restituisce `true` in caso di successo e `false` in caso di fallimento; è un errore se il socket non esiste;
- `natl accept(nat1 id)` (da realizzare): pone il processo in attesa di connessioni sul socket `id`; restituisce 0xFFFFFFFF se il socket non è nello stato giusto; è un errore se il socket non esiste.
- `bool connect(nat1 src, natl dest)` (da realizzare): tenta di connettere il socket `src` con il socket `dest`; restituisce `false` se uno dei due socket non è nello stato giusto; è un errore se uno dei due socket non esiste.

Tenere conto di eventuali preemption. **Attenzione:** può essere necessario aggiungere informazioni ai descrittori di processo.

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

```
// ( SOLUZIONE 2015-01-29
    carica_gateTIPO_ACCEPTa_accept    LIV_UTENTE
    carica_gateTIPO_CONNECT          a_connect    LIV_UTENTE
//   SOLUZIONE 2015-01-29 )
// ( SOLUZIONE 2015-01-29

a_accept:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_accept
    call carica_stato
    iretq
    .cfi_endproc

a_connect:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_connect
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2015-01-29 )
```

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

```
// ( SOLUZIONE 2015-01-29 #1
    natl sock;
//   SOLUZIONE 2015-01-29 )
// ( SOLUZIONE 2015-01-29 #3
    p->sock = 0xffffffff;
//   SOLUZIONE 2015-01-29 )
// ( SOLUZIONE 2015-01-29 #2
extern "C" void c_accept(natl id)
{
    if (!sock_valid(id)) {
        flog(LOG_WARN, "socket id %d non valido", id);
        c_abort_p();
        return;
    }

    des_sock *as = &array_des_sock[id];

    if (!sock_can_accept(as)) {
        flog(LOG_WARN, "accept non ammessa su socket %d", id);
        esecuzione->contesto[I_RAX] = 0xFFFFFFFF;
        return;
    }
```

```

if (as->connecting) {
    des_proc *work = rimozione_lista(as->connecting);

    inspronti();
    inserimento_lista(pronti, work);

    natl c = c_socket();
    esecuzione->contesto[I_RAX] = c;
    if (c == 0xffffffff) {
        work->contesto[I_RAX] = (natq)false;
    } else {
        des_sock *ns = &array_des_sock[c];
        des_sock *cs = &array_des_sock[work->sock];

        ns->state = SOCK_CONNECTED;
        cs->state = SOCK_CONNECTED;

        work->sock = 0xffffffff;
        work->contesto[I_RAX] = (natq)true;
    }
} else {
    as->state = SOCK_ACCEPTING;
    inserimento_lista(as->accepting, esecuzione);
}
schedulatore();
}

extern "C" void c_connect(natl id, natl dest)
{
    if (!sock_valid(id)) {
        flog(LOG_WARN, "socket id %d non valido", id);
        c_abort_p();
        return;
    }

    if (!sock_valid(dest)) {
        flog(LOG_WARN, "socket id %d non valido", dest);
        c_abort_p();
        return;
    }

    des_sock *cs = &array_des_sock[id];
    des_sock *as = &array_des_sock[dest];

    if (!sock_src_ok(cs)) {
        flog(LOG_WARN, "connect non ammessa su socket %d", id);
        esecuzione->contesto[I_RAX] = (natq)false;
        return;
    }

    if (!sock_dest_ok(as)) {
        flog(LOG_WARN, "socket %d non puo' essere destinazione",
dest);
        esecuzione->contesto[I_RAX] = (natq)false;
        return;
    }

    if (as->accepting) {
        des_proc *work = rimozione_lista(as->accepting);
        if (!as->accepting)

```

```

        as->state = SOCK_LISTENING;

inspronti();
inserimento_lista(pronti, work);

natl c = c_socket();
work->contesto[I_RAX] = c;
if (c == 0xffffffff) {
    esecuzione->contesto[I_RAX] = (natq) false;
} else {
    des_sock *ns = &array_des_sock[c];

    ns->state = SOCK_CONNECTED;
    cs->state = SOCK_CONNECTED;

    esecuzione->contesto[I_RAX] = (natq) true;
}
} else {
    cs->state = SOCK_CONNECTING;
    esecuzione->sock = id;
    inserimento_lista(as->connecting, esecuzione);
}
schedulatore();
}

// SOLUZIONE 2015-01-29 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

18 febbraio 2015

1. Introduciamo nel nucleo una versione semplificata del meccanismo dei socket per la comunicazione inter-processo, relativamente alle sole fasi di connessione e chiusura.

Un socket è un oggetto identificato tramite un numero naturale. È possibile creare una connessione tra due socket tramite le operazioni di `accept(nat1 id)` e `connect(nat1 src, nat1 dest)`.

Un processo che esegue `accept()` su un socket si pone in attesa di richieste di connessione verso verso quel socket. Su uno stesso socket ci può essere più di un processo in attesa di connessioni (più processi possono invocare `accept()` su uno stesso socket).

La primitiva `connect()` permette ad un processo di richiedere una connessione da un certo socket sorgente verso un socket destinatario. Se sul socket di destinazione vi sono processi in attesa, quello a più alta priorità viene risvegliato e la connessione viene completata. I socket connessi saranno: il socket sorgente della `connect()`; un nuovo socket, il cui identificatore sarà restituito dalla `accept()`. Se non vi sono processi in attesa, la `connect()` fallisce.

In qualunque momento può essere invocata una `close()` su un socket, che rende il socket nuovamente disponibile per qualunque uso. Se vi sono processi in attesa di connessioni, questi devono essere tutti risvegliati facendo fallire la primitiva che avevano invocato. Se il socket è connesso con un altro socket, il processo che invoca la `close()` deve attendere che anche l'altro socket venga chiuso. Se il socket era già utilizzato, la primitiva fallisce.

Per realizzare questo meccanismo definiamo le seguenti strutture dati:

```
enum sock_state {
    SOCK_AVAIL,
    SOCK_ACCEPTING,
    SOCK_CONNECTED,
    SOCK_CLOSING
};

struct des_sock {
    sock_state state;
    des_proc *waiting;
    des_sock *peer;
};
```

I possibili stati di un socket sono i seguenti:

- `SOCK_AVAIL`: il socket non è al momento utilizzato;
- `SOCK_ACCEPTING`: c'è almeno un processo che sta accettando connessioni su questo socket;
- `SOCK_CONNECTED`: il socket è connesso;
- `SOCK_CLOSING`: il socket è in fase di chiusura.

La lista `waiting` contiene i processi che sono in attesa di qualche evento sul socket (connessioni o completamento della chisura); il campo `peer` punta all'altro socket nel caso di socket connessi.

Aggiungiamo inoltre le seguenti primitive (abortiscono il processo in caso di errore):

- `natl socket()` (già realizzata): crea un socket e ne restituisce l'identificatore (0xFFFFFFFF se non è stato possibile crearlo);
- `natl accept(natl id)` (già realizzata): pone il processo in attesa di connessioni sul socket `id`; restituisce 0xFFFFFFFF in caso di fallimento; è un errore se il socket non esiste.
- `bool connect(natl src, natl dest)` (già realizzata): tenta di connettere il socket `src` con il socket `dest`; restituisce `false` in caso di fallimento; è un errore se uno dei due socket non esiste.
- `bool close(natl id)` (da realizzare): chiude il socket; restituisce `false` in caso di fallimento.

Tenere conto di eventuali preemption.

```

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


// ( SOLUZIONE 2015-02-18
    carica_gateTIPO_CLOSE a_close          LIV_UTENTE
//   SOLUZIONE 2015-02-18 )
// ( SOLUZIONE 2015-02-18

a_close:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_close
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2015-02-18 )

*****


* sistema/sistema.cpp
*****


// ( SOLUZIONE 2015-02-18
extern "C" void c_close(natl id)
{
    des_proc *work;

    if (!sock_valid(id)) {
        flog(LOG_WARN, "socket id %d non valido", id);
        c_abort_p();
        return;
    }

    des_sock *s = &array_des_sock[id];
    esecuzione->contesto[I_RAX] = (natq)true;

    switch (s->state) {
    case SOCK_ACCEPTING:
        inspronti();
        while (s->waiting) {
            work = rimozione_lista(s->waiting);
            work->contesto[I_RAX] = 0xffffffff;
            inserimento_lista(pronti, work);
        }
        s->state = SOCK_AVAIL;
        break;
    case SOCK_CONNECTED:
        s->state = s->peer->state = SOCK_CLOSING;
        inserimento_lista(s->waiting, esecuzione);
        break;
    case SOCK_CLOSING:
        inspronti();
        work = rimozione_lista(s->peer->waiting);
        inserimento_lista(pronti, work);
        s->state = s->peer->state = SOCK_AVAIL;
    }
}

```

```
s->waiting = s->peer->waiting = 0;
s->peer->peer = 0;
s->peer = 0;
break;
default:
    esecuzione->contesto[I_RAX] = (natq)false;
    return;
}
schedulatore();
}
// SOLUZIONE 2015-02-18 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

11 giugno 2015

1. Aggiungiamo al nucleo la seguente primitiva (in caso di errore abortisce il processo):

- `bool set_prio(natl id, natl prio)` (tipo 0x5c, da realizzare): assegna la priorità `prio` al processo di identificatore `id`. Restituisce `false` se il processo non esiste e `true` altrimenti (salvo errori). È un errore se un processo tenta di assegnare una priorità maggiore della propria, oppure se il processo `id` è di livello sistema.

La primitiva può essere usata per modificare dinamicamente la priorità di qualunque processo utente, incluso il processo che la invoca.

ATTENZIONE: la primitiva deve funzionare qualunque sia lo stato del processo di cui si vuole cambiare la priorità (pronto, bloccato o in esecuzione) e devono essere gestite correttamente eventuali *preemption*.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare la primitiva.

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

```
// ( SOLUZIONE 2015-06-11
    carica_gateTIPO_SP           a_set_prio LIV_UTENTE
//   SOLUZIONE 2015-06-11 )
// ( SOLUZIONE 2015-06-11
```

```
a_set_prio:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_set_prio
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2015-06-11 )
```

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

```
// ( SOLUZIONE 2015-06-11
des_proc *estrazione_lista(des_proc *&lista, natl id)
{
    des_proc *prec, *scorri;

    for (prec = 0, scorri = lista; scorri; prec = scorri, scorri =
scorri->puntatore)
        if (scorri->id == id)
            break;

    if (scorri) {
        if (prec)
            prec->puntatore = scorri->puntatore;
        else
            lista = scorri->puntatore;
        scorri->puntatore = 0;
    }

    return scorri;
}
```

```
extern "C" void c_set_prio(natl id, natl prio)
{
    if (id >= MAX_PROC_ID) {
        flog(LOG_WARN, "id non valido: %d", id);
        c_abort_p();
        return;
    }
    des_proc *dest = des_p(id);

    if (prio > esecuzione->precedenza) {
        flog(LOG_WARN, "priorita' non ammessa: %d", prio);
        c_abort_p();
        return;
    }
```

```

    }
    if (!dest) {
        esecuzione->contesto[I_RAX] = false;
        return;
    }
    if (dest->livello == LIV_SISTEMA) {
        flog(LOG_WARN, "impossibile cambiare priorita' di %d", id);
        c_abort_p();
        return;
    }
    esecuzione->contesto[I_RAX] = true;

    if (esecuzione->id == id) {
        esecuzione->precedenza = prio;
        inserimento_lista(pronti, esecuzione);
        schedulatore();
        return;
    }
    des_proc *work = estrazione_lista(pronti, id);
    if (work) {
        work->precedenza = prio;
        inspronti();
        inserimento_lista(pronti, work);
        schedulatore();
        return;
    }

    for (natl sem = 0; sem < MAX_SEM + sem_allocati_sistema; sem++) {
        des_sem *s = &array_dess[sem];

        work = estrazione_lista(s->pointer, id);
        if (work) {
            work->precedenza = prio;
            inserimento_lista(s->pointer, work);
            return;
        }

        if (sem == sem_allocati_utente - 1)
            sem = MAX_SEM;
    }

    for (richiesta *r = p_sospesi; r; r = r->p_rich) {
        if (r->pp->id == id) {
            r->pp->precedenza = prio;
            return;
        }
    }

    /* impossibile arrivare qui */
    esecuzione->contesto[I_RAX] = false;
}
// SOLUZIONE 2015-06-11 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

2 luglio 2015

- Nel nucleo che abbiamo studiato tutta la memoria, con l'esclusione delle pile, è condivisa tra tutti i processi. Vogliamo aggiungere un meccanismo tramite il quale solo alcuni processi, e non necessariamente tutti, possano condividere della memoria.

Prevediamo quindi che un processo possa creare delle zone di memoria, dette **shmem**, ciascuna con un identificatore unico. I processi che vogliono accedere ad una **shmem** devono aggiungerla al proprio spazio di indirizzamento, specificandone l'identificatore. Una volta aggiunta, la **shmem** sarà disponibile contiguamente all'interno della parte utente/condivisa dello spazio di indirizzamento del processo. Un processo può aggiungere più **shmem** al proprio spazio e le diverse **shmem** non devono sovrapporsi. Non è importante che i processi che condividono una stessa **shmem** la vedano tutti allo stesso indirizzo.

Per descrivere una **shmem** aggiungiamo al nucleo la seguente struttura dati:

```
struct des_shmem {  
    natl npag;  
    natl first_frame_number;  
};
```

Il campo **npag** contiene la dimensione (in pagine) della **shmem**. Tutti i frame che contengono la **shmem**, nell'ordine in cui devono comparire nella memoria di tutti i processi che la condividono, sono mantenuti in una lista, implementata tramite l'array **vdf**. Il numero del primo frame della lista è contenuto nel campo **first_frame_number** e l'elemento **vdf[fn]** di ogni **fn** della lista contiene il numero del frame successivo (il valore 0 termina la lista).

Inoltre, aggiungiamo il seguente campo ai descrittori di processo:

```
addr avail_addr;
```

Questo campo contiene il primo indirizzo libero nella parte utente/condivisa del processo. Tutti gli indirizzi da **avail_addr** fino a **fin_utn_c** (escluso) sono disponibili per contenere zone di memoria condivisa.

Aggiungiamo infine le seguenti primitive:

- **natl shmem_create(natl npag)** (tipo 0x5c, già realizzata): Crea una nuova zona di memoria condivisibile tra più processi, grande **npag** pagine, e ne restituisce l'identificatore. La primitiva si limita ad allocare i frame necessari e a inserirli in una lista. Se non vi sono frame liberi a sufficienza, restituisce 0xFFFFFFF.
- **addr shmem_attach(natl id)** (tipo 0x5d, da realizzare): Permette ad un processo di aggiungere la **shmem** **id** al proprio spazio di indirizzamento e ne restituisce l'indirizzo di partenza. Abortisce il processo se **id** non corrisponde ad una **shmem** esistente. Restituisce 0 se non è stato possibile aggiungere la zona (perché non vi sono indirizzi liberi sufficienti a contenerla).

Modificare i file **sistema.cpp** e **sistema.S** in modo da realizzare le primitive appena descritte.

```

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


// ( SOLUZIONE 2015-07-02
    carica_gateTIPO_SA           a_shmem_attach    LIV_UTENTE
//   SOLUZIONE 2015-07-02 )
// ( SOLUZIONE 2015-07-02

a_shmem_attach:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_shmem_attach
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2015-07-02 )

*****


* sistema/sistema.cpp
*****


// ( SOLUZIONE 2015-07-02
extern "C" void c_shmem_attach(natl id)
{
    if (!shmem_valid(id)) {
        flog(LOG_WARN, "shmem non valida: %d", id);
        c_abort_p();
        return;
    }

    des_shmem *sh = &array_desshmem[id];

    esecuzione->contesto[I_RAX] = 0;

    vaddr start = esecuzione->avail_addr;
    vaddr end    = start + sh->npag * DIM_PAGINA;

    if (end < start || end >= fin_utn_c) {
        flog(LOG_WARN, "Impossibile mappare %d pagine da %p",
              sh->npag, start);
        return;
    }

    natl fn = sh->first_frame_number;
    vaddr last = map(esecuzione->cr3, start, end, BIT_RW|BIT_US,
                      [&](vaddr) {
                          natl cfn = fn;
                          fn = vdf[fn].next_shmem;
                          return cfn * DIM_PAGINA;
                      });
    if (last != end) {
        unmap(esecuzione->cr3, start, last, [](vaddr, int) {});
        return;
    }
    esecuzione->avail_addr = end;
}

```

```
    esecuzione->contesto[I_RAX] = start;  
}  
// SOLUZIONE 2015-07-02 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

23 luglio 2015

1. Nel nucleo che abbiamo studiato tutta la memoria, con l'esclusione delle pile, è condivisa tra tutti i processi. Vogliamo aggiungere un meccanismo tramite il quale solo alcuni processi, e non necessariamente tutti, possano condividere della memoria.

Prevediamo quindi che un processo possa creare delle zone di memoria, dette **shmem**, ciascuna con un identificatore unico. I processi che vogliono accedere ad una **shmem** devono aggiungerla al proprio spazio di indirizzamento, specificandone l'identificatore. Una volta aggiunta, la **shmem** sarà disponibile contiguamente all'interno della parte utente/condivisa dello spazio di indirizzamento del processo. Un processo può aggiungere più **shmem** al proprio spazio e le diverse **shmem** non devono sovrapporsi. Non è importante che i processi che condividono una stessa **shmem** la vedano tutti allo stesso indirizzo.

Per descrivere una **shmem** aggiungiamo al nucleo la seguente struttura dati:

```
struct des_shmem {  
    natl npag;  
    natl first_frame_number;  
};
```

Il campo **npag** contiene la dimensione (in pagine) della **shmem**. Tutti i frame che contengono la **shmem**, nell'ordine in cui devono comparire nella memoria di tutti i processi che la condividono, sono mantenuti in una lista, implementata tramite l'array **vdf**. Il numero del primo frame della lista è contenuto nel campo **first_frame_number** e l'elemento **vdf[fn]** di ogni **fn** della lista contiene il numero del frame successivo (il valore 0 termina la lista).

Inoltre, aggiungiamo il seguente campo ai descrittori di processo:

```
addr avail_addr;
```

Questo campo contiene il primo indirizzo libero nella parte utente/condivisa del processo. Tutti gli indirizzi da **avail_addr** fino a **fin_utn_c** (escluso) sono disponibili per contenere zone di memoria condivisa.

Aggiungiamo infine le seguenti primitive:

- **natl shmem_create(natl npag)** (tipo 0x5c, già realizzata): Crea una nuova zona di memoria condivisibile tra più processi, grande **npag** pagine, e ne restituisce l'identificatore. La primitiva si limita ad allocare i frame necessari e a inserirli in una lista. Se non vi sono frame liberi a sufficienza, restituisce 0xFFFFFFF.
- **addr shmem_attach(natl id)** (tipo 0x5d, da realizzare): Permette ad un processo di aggiungere la **shmem** **id** al proprio spazio di indirizzamento e ne restituisce l'indirizzo di partenza. Abortisce il processo se **id** non corrisponde ad una **shmem** esistente. Restituisce 0 se non è stato possibile aggiungere la zona (perché non vi sono indirizzi liberi sufficienti a contenerla).

Modificare i file **sistema.cpp** e **sistema.S** in modo da realizzare le primitive appena descritte.

```

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


// ( SOLUZIONE 2015-07-23
    carica_gateTIPO_SD           a_shmem_detach    LIV_UTENTE
//   SOLUZIONE 2015-07-23 )
// ( SOLUZIONE 2015-07-23

a_shmem_detach:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_shmem_detach
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2015-07-23 )

*****


* sistema/sistema.cpp
*****


// ( SOLUZIONE 2015-07-23
extern "C" void c_shmem_detach(natl id)
{
    des_attached **patt = nullptr;

    for (patt = &esecuzione->att; *patt; patt = &(*patt)->next)
        if ((*patt)->id == id)
            break;
    if (!*patt) {
        flog(LOG_WARN, "shmem id non valido: %d", id);
        c_abort_p();
        return;
    }
    des_shmem *sh = &array_desshmem[id];
    natq npag = sh->npag;

    vaddr indv = (*patt)->start;
    des_attached* old = *patt;
    *patt = (*patt)->next;
    delete old;

    unmap(esecuzione->cr3, indv, indv + npag * DIM_PAGINA,
          [] (vaddr v, int) {
              invalida_entrata_TLB(v);
          });
}
//   SOLUZIONE 2015-07-23 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

15 settembre 2015

- Nel nucleo che abbiamo studiato tutta la memoria, con l'esclusione delle pile, è condivisa tra tutti i processi. Vogliamo aggiungere un meccanismo tramite il quale solo alcuni processi, e non necessariamente tutti, possano condividere della memoria.

Prevediamo quindi che un processo possa creare delle zone di memoria, dette `shmem`, ciascuna con un identificatore unico. I processi che vogliono accedere ad una `shmem` devono aggiungerla al proprio spazio di indirizzamento, specificandone l'identificatore. Una volta aggiunta, la `shmem` sarà disponibile contiguamente all'interno della parte utente/condivisa dello spazio di indirizzamento del processo. Un processo può aggiungere più `shmem` al proprio spazio e le diverse `shmem` non devono sovrapporsi. Non è importante che i processi che condividono una stessa `shmem` la vedano tutti allo stesso indirizzo.

Per descrivere una `shmem` aggiungiamo al nucleo la seguente struttura dati:

```
struct des_shmem {  
    natl npag;  
    natl first_frame_number;  
};
```

Il campo `npag` contiene la dimensione (in pagine) della `shmem`. Tutti i frame che contengono la `shmem`, nell'ordine in cui devono comparire nella memoria di tutti i processi che la condividono, sono mantenuti in una lista, implementata tramite l'array `vdf`. Il numero del primo frame della lista è contenuto nel campo `first_frame_number` e l'elemento `vdf[fn]` di ogni `fn` della lista contiene il numero del frame successivo (il valore 0 termina la lista).

Inoltre, aggiungiamo il seguente campo ai descrittori di processo:

```
addr avail_addr;
```

Questo campo contiene il primo indirizzo libero nella parte utente/condivisa del processo. Tutti gli indirizzi da `avail_addr` fino a `fin_utn_c` (escluso) sono disponibili per contenere zone di memoria condivisa.

Aggiungiamo infine le seguenti primitive:

- `natl shmem_create(natl npag)` (tipo 0x5c, già realizzata): Crea una nuova zona di memoria condivisibile tra più processi, grande `npag` pagine, e ne restituisce l'identificatore. La primitiva si limita ad allocare i frame necessari e a inserirli in una lista. Se non vi sono frame liberi a sufficienza, restituisce 0xFFFFFFF.
- `addr shmem_attach(natl id)` (tipo 0x5d, da realizzare): Permette ad un processo di aggiungere la `shmem` `id` al proprio spazio di indirizzamento e ne restituisce l'indirizzo di partenza. Abortisce il processo se `id` non corrisponde ad una `shmem` esistente. Restituisce 0 se non è stato possibile aggiungere la zona (perché non vi sono indirizzi liberi sufficienti a contenerla).

Modificare i file `sistema.cpp` e `sistema.S` in modo da realizzare le primitive appena descritte.

```

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


// ( SOLUZIONE 2015-09-15
    carica_gateTIPO_SD           a_shmem_detach    LIV_UTENTE
    carica_gateTIPO_SY           a_shmem_destroy   LIV_UTENTE
//  SOLUZIONE 2015-09-15 )
// ( SOLUZIONE 2015-09-15

a_shmem_detach:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_shmem_detach
    call carica_stato
    iretq
    .cfi_endproc

a_shmem_destroy:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_shmem_destroy
    call carica_stato
    iretq
    .cfi_endproc
//  SOLUZIONE 2015-09-15 )

*****


* sistema/sistema.cpp
*****


// ( SOLUZIONE 2015-09-15
des_attached **shmem_find_att(natl id)
{
    des_attached **patt = nullptr;

    for (patt = &esecuzione->att; *patt; patt = &(*patt)->next)
        if ((*patt)->id == id)
            break;
    return patt;
}

void shmem_free(des_shmem *sh)
{
    natl f = sh->first_frame_number;
    while (f) {
        natl n = vdf[f].next_shmem;
        rilascia_frame(f * DIM_PAGINA);
        f = n;
    }
    sh->first_frame_number = 0;
    sh->npag = 0;
}

```

```

extern "C" void c_shmem_detach(natl id)
{
    des_attached **patt = shmem_find_att(id);
    if (!*patt) {
        flog(LOG_WARN, "shmem id non valido: %d", id);
        c_abort_p();
        return;
    }
    des_shmem *sh = &array_desshmem[id];
    natq npag = sh->npag;

    vaddr indv = (*patt)->start;
    des_attached* old = *patt;
    *patt = (*patt)->next;
    delete old;

    unmap(esecuzione->cr3, indv, indv + npag * DIM_PAGINA,
          [](vaddr v, int) {
              invalida_entrata_TLB(v);
          });
    sh->nusers--;
    if (!sh->nusers && sh->wait_detach) {
        shmem_free(sh);
        inspronti();
        while (sh->wait_detach) {
            des_proc *work = rimozione_lista(sh->wait_detach);
            inserimento_lista(pronti, work);
        }
        schedulatore();
    }
}

extern "C" void c_shmem_destroy(natl id)
{
    if (!shmem_valid(id)) {
        flog(LOG_WARN, "shmem non valida: %d", id);
        c_abort_p();
        return;
    }

    des_shmem *sh = &array_desshmem[id];
    des_attached **patt = shmem_find_att(id);
    if (*patt) {
        flog(LOG_WARN, "shmem_destroy su shmem attaccata");
        c_abort_p();
        return;
    }

    if (sh->nusers) {
        inserimento_lista(sh->wait_detach, esecuzione);
        schedulatore();
        return;
    }

    shmem_free(sh);
}
// SOLUZIONE 2015-09-15 )

```


Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

19 gennaio 2016

1. Due processi possono comunicare tramite una **pipe**, un canale con una estremità di scrittura e una di lettura attraverso il quale viaggia una sequenza di caratteri. I caratteri inviati dall'estremità di scrittura possono essere letti dall'estremità di lettura.

Per realizzare le **pipe** aggiungiamo le seguenti primitive (abortiscono il processo in caso di errore):

- **natl inipipe()** (tipo 0x5c, già realizzata): Crea una nuova pipe e ne restituisce l'identificatore (0xFFFFFFFF se non è stato possibile creare una nuova pipe).
- **void writepipe(natl p, char *buf, natl n)** (tipo 0x5d, da realizzare): Invia **n** caratteri dal buffer **buf** sulla pipe di identificatore **p**. È un errore se la pipe **p** non esiste.
- **void readpipe(natl p, char *buf, natl n)** (tipo 0x53, da realizzare): Riceve **n** caratteri dal dalla pipe di identificatore **p** e li scrive nel buffer **buf**. È un errore se la pipe **p** non esiste.

Previdiamo un tipo di **pipe** senza un buffer interno. Per la **writepipe**, questo vuol dire che i caratteri possono essere trasferiti solo se un altro processo è pronto a riceverli tramite una **readpipe**, altrimenti la primitiva deve attendere. Inoltre, il processo che ha invocato la **readpipe** potrebbe aver chiesto meno caratteri di quelli da inviare: in questo caso si devono inviare i caratteri possibili e continuare ad attendere; questa operazione potrebbe ripetersi più volte fino a quando tutti i caratteri non sono stati trasferiti. Analoghe considerazioni valgono per la **readpipe**.

Per semplicità non trattiamo i casi in cui più di un processo voglia accedere alla stessa estremità della stessa pipe. Inoltre, assumiamo che i buffer passati alla **readpipe** e alla **writepipe** appartengano allo spazio utente comune.

Per descrivere una **pipe** aggiungiamo al nucleo la seguente struttura dati:

```
struct des_pipe {  
    des_proc* w_wait;  
    char *w_buf;  
    natl w_pending;  
  
    des_proc* r_wait;  
    char *r_buf;  
    natl r_pending;  
};
```

Il campo **w_wait** punta all'eventuale processo in attesa di poter completare una **writepipe**; vale NULL se non ve ne sono. Se **w_wait** non è nullo, **w_buf** punta al buffer contenente i caratteri ancora da trasferire e **w_pending** ne indica il numero. I campi **r_wait**, **r_buf** e **r_pending** svolgono un ruolo analogo per i processi in attesa di completare una **readpipe**.

Modificare i file **sistema.cpp** e **sistema.s** in modo da realizzare le primitive mancanti.

```

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


// ( SOLUZIONE 2016-01-19
extern "C" void c_writepipe(natl p, char *buf, natl n)
{
    if (!pipe_valid(p)) {
        flog(LOG_WARN, "pipe non valida %d", p);
        c_abort_p();
        return;
    }

    if (!c_access(reinterpret_cast<vaddr>(buf), n, false)) {
        flog(LOG_WARN, "buf non valido");
        c_abort_p();
        return;
    }

    struct des_pipe *dp = &array_despipe[p];

    for ( ; dp->r_pending && n; dp->r_pending--, n--)
        *dp->r_buf++ = *buf++;

    if (n) {
        dp->w_pending = n;
        dp->w_buf = buf;
        inserimento_lista(dp->w_wait, esecuzione);
    } else {
        inspronti();
    }

    if (!dp->r_pending && dp->r_wait) {
        des_proc *work = rimozione_lista(dp->r_wait);
        inserimento_lista(pronti, work);
    }
}

schedulatore();
}

extern "C" void c_readpipe(natl p, char *buf, natl n)
{
    if (!pipe_valid(p)) {
        flog(LOG_WARN, "pipe non valida %d", p);
        c_abort_p();
        return;
    }

    if (!c_access(reinterpret_cast<vaddr>(buf), n, true)) {
        flog(LOG_WARN, "buf non valido");
        c_abort_p();
        return;
    }

    struct des_pipe *dp = &array_despipe[p];

    for ( ; dp->w_pending && n; dp->w_pending--, n--)
        *buf++ = *dp->w_buf++;

    if (n) {

```

```
    dp->r_pending = n;
    dp->r_buf = buf;
    inserimento_lista(dp->r_wait, esecuzione);
} else {
    inspronti();
}

if (!dp->w_pending && dp->w_wait) {
    des_proc *work = rimozione_lista(dp->w_wait);
    inserimento_lista(pronti, work);
}

schedulatore();
}
// SOLUZIONE 2016-01-19 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

8 febbraio 2016

1. Due processi possono comunicare tramite una **pipe**, un canale con una estremità di scrittura e una di lettura attraverso il quale viaggia una sequenza di caratteri. I caratteri inviati dall'estremità di scrittura possono essere letti dall'estremità di lettura.

Per realizzare le **pipe** aggiungiamo le seguenti primitive (abortiscono il processo in caso di errore):

- **natl inipipe()** (tipo 0x5c, da realizzare): Crea una nuova pipe e ne restituisce l'identificatore (0xFFFFFFFF se non è stato possibile creare una nuova pipe).
- **void writepipe(natl p, char *buf, natl n)** (tipo 0x5d, da realizzare): Invia **n** caratteri dal buffer **buf** sulla pipe di identificatore **p**. È un errore se la pipe **p** non esiste.
- **void readpipe(natl p, char *buf, natl n)** (tipo 0x53, da realizzare): Riceve **n** caratteri dalla pipe di identificatore **p** e li scrive nel buffer **buf**. È un errore se la pipe **p** non esiste.

Prevediamo un tipo di **pipe** senza un buffer interno. Per la **writepipe**, questo vuol dire che i caratteri possono essere trasferiti solo se un altro processo è pronto a riceverli tramite una **readpipe**, altrimenti la primitiva deve attendere. Inoltre, il processo che ha invocato la **readpipe** potrebbe aver chiesto meno caratteri di quelli da inviare: in questo caso si devono inviare i caratteri possibili e continuare ad attendere; questa operazione potrebbe ripetersi più volte fino a quando tutti i caratteri non sono stati trasferiti. Analoghe considerazioni valgono per la **readpipe**.

Per semplicità non trattiamo i casi in cui più di un processo voglia accedere alla stessa estremità della stessa pipe. Inoltre, assumiamo che i buffer passati alla **readpipe** e alla **writepipe** appartengano allo spazio utente comune.

Per descrivere una **pipe** aggiungiamo al nucleo la seguente struttura dati:

```
struct des_pipe {  
    natl reader_ready;  
    natl write_done;  
    char *r_buf;  
    natl r_pending;  
};
```

Il campo **reader_ready** è l'indice di un semaforo di sincronizzazione usato per notificare che un processo è in attesa di completare una **readpipe**, nel qual caso **r_pending** contiene il numero di caratteri ancora da trasferire e **r_buf** l'indirizzo a cui devono essere trasferiti. Il campo **write_done** è l'indice di un semaforo di sincronizzazione usato per notificare che il trasferimento richiesto dall'ultima **readpipe** è stato completato.

Modificare i file **sistema.cpp** e **sistema.S** in modo da realizzare le primitive mancanti.

SUGGERIMENTO: è possibile utilizzare le primitive semaforiche.

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

```
// ( SOLUZIONE 2016-02-08
extern "C" natl c_inpipe()
{
    if (nextpipe >= MAX_PIPES)
        return 0xFFFFFFFF;

    des_pipe *dp = &array_despipe[nextpipe];

    dp->reader_ready = sem_ini(0);
    dp->write_done = sem_ini(0);
    if (dp->reader_ready == 0xFFFFFFFF || dp->write_done == 0xFFFFFFFF)
        return 0xFFFFFFFF;

    return nextpipe++;
}

extern "C" void c_writepipe(natl p, char *buf, natl n)
{
    if (!pipe_valid(p)) {
        flog(LOG_WARN, "pipe non valida %d", p);
        abort_p();
    }

    if (!c_access(reinterpret_cast<vaddr>(buf), n, false)) {
        flog(LOG_WARN, "buf non valido");
        abort_p();
    }

    struct des_pipe *dp = &array_despipe[p];

    while (n) {
        while (!dp->r_pending)
            sem_wait(dp->reader_ready);

        for ( ; dp->r_pending && n; dp->r_pending--, n--)
            *dp->r_buf++ = *buf++;

        if (!dp->r_pending)
            sem_signal(dp->write_done);
    }
}

extern "C" void c_readpipe(natl p, char *buf, natl n)
{
    if (!pipe_valid(p)) {
        flog(LOG_WARN, "pipe non valida %d", p);
        abort_p();
    }

    if (!c_access(reinterpret_cast<vaddr>(buf), n, true)) {
        flog(LOG_WARN, "buf non valido");
        abort_p();
    }

    struct des_pipe *dp = &array_despipe[p];
```

```
dp->r_pending = n;
dp->r_buf = buf;

while (dp->r_pending) {
    sem_signal(dp->reader_ready);
    sem_wait(dp->write_done);
}

dp->r_buf = 0;
}
// SOLUZIONE 2016-02-08 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

25 febbraio 2016

1. Due processi possono comunicare tramite una **pipe**, un canale con una estremità di scrittura e una di lettura attraverso il quale viaggia una sequenza di caratteri. I caratteri inviati dall'estremità di scrittura possono essere letti dall'estremità di lettura.

Per realizzare le **pipe** aggiungiamo le seguenti primitive (abortiscono il processo in caso di errore):

- **natl inipipe()** (tipo 0x5c, da realizzare): Crea una nuova pipe e ne restituisce l'identificatore (0xFFFFFFFF se non è stato possibile creare una nuova pipe).
- **void writepipe(nat1 p, char *buf, nat1 n)** (tipo 0x5d, da realizzare): Invia **n** caratteri dal buffer **buf** sulla pipe di identificatore **p**. È un errore se la pipe **p** non esiste.
- **void readpipe(nat1 p, char *buf, nat1 n)** (tipo 0x53, già realizzata): Riceve **n** caratteri dalla pipe di identificatore **p** e li scrive nel buffer **buf**. È un errore se la pipe **p** non esiste.

Previdamo un tipo di **pipe** con buffer interno. La **writepipe**, trasferisce i byte nel buffer interno e la **readpipe** li preleva dal buffer. La **writepipe** blocca il processo chiamante solo quando il buffer è pieno e la **readpipe** solo quando il buffer è vuoto. Entrambe ritornano al chiamante solo quando l'intero trasferimento è stato completato (quindi è possibile che il processo sia bloccato e risvegliato più volte).

Per semplicità non trattiamo i casi in cui più di un processo voglia accedere alla stessa estremità della stessa pipe.

Per descrivere una **pipe** aggiungiamo al nucleo la seguente struttura dati:

```
struct des_pipe {
    natl not_full;
    bool writer_waiting;
    natl not_empty;
    bool reader_waiting;

    char buf [BUFSIZE];
    natl head;
    natl tail;
    natl n;
};
```

Il campo **not_full** è l'indice di un semaforo di sincronizzazione su cui attendere che il buffer non sia pieno; Il booleano **writer_waiting** è vero se e solo se lo scrittore è bloccato sulla pipe; Il campo **not_empty** è l'indice di un semaforo di sincronizzazione su cui attendere che il buffer non sia vuoto; Il booleano **reader_waiting** è vero se e solo se il lettore è bloccato sulla pipe; I campi **buf**, **head**, **tail** e **n** servono a realizzare il buffer interno come una coda circolare (**n** è il numero di byte che il buffer contiene).

Modificare i file **sistema.cpp** e **sistema.S** in modo da realizzare le primitive mancanti.

SUGGERIMENTO: è possibile utilizzare le primitive semaforiche.

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

```
// ( SOLUZIONE 2016-02-25
    carica_gateTIPO_INIPIPE      a_inpipe    LIV_UTENTE
    carica_gateTIPO_WRITEPIPE   a_writepipe LIV_UTENTE
//  SOLUZIONE 2016-02-25 )
// ( SOLUZIONE 2016-02-25

a_inpipe:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_inpipe
    iretq
    .cfi_endproc

a_writepipe:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_writepipe
    call carica_stato
    iretq
    .cfi_endproc
//  SOLUZIONE 2016-02-25 )
```

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

```
// ( SOLUZIONE 2016-02-25
extern "C" natl c_inpipe()
{
    if (nextpipe >= MAX_PIPES)
        return 0xFFFFFFFF;

    des_pipe *dp = &array_despipe[nextpipe];

    dp->not_full = sem_ini(0);
    dp->not_empty = sem_ini(0);
    if (dp->not_full == 0xFFFFFFFF || dp->not_empty == 0xFFFFFFFF)
        return 0xFFFFFFFF;

    dp->head = dp->tail = dp->n = 0;
    dp->writer_waiting = dp->reader_waiting = false;

    return nextpipe++;
}

extern "C" void c_writepipe(natl p, char *buf, natl n)
{
    if (!pipe_valid(p)) {
        flog(LOG_WARN, "pipe non valida %d", p);
        abort_p();
    }
}
```

```

}

if (!c_access(reinterpret_cast<vaddr>(buf), n, false)) {
    flog(LOG_WARN, "buf non valido");
    abort_p();
}

struct des_pipe *dp = &array_despipe[p];

while (n) {
    if (dp->n == BUFSIZE) {
        dp->writer_waiting = true;
        sem_wait(dp->not_full);
    }

    while (n && dp->n < BUFSIZE) {
        dp->buf[dp->head] = *buf++;
        dp->head = (dp->head + 1) % BUFSIZE;
        dp->n++;
        n--;
    }

    if (dp->reader_waiting) {
        dp->reader_waiting = false;
        sem_signal(dp->not_empty);
    }
}
// SOLUZIONE 2016-02-25 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

15 giugno 2016

- 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 periferiche di ingresso in grado di operare in PCI Bus Mastering. I registri accessibili al programmatore sono i seguenti:

- BMPTR** (indirizzo *b*, 4 byte): puntatore al buffer di destinazione;
- BMLEN** (indirizzo *b* + 4, 4 byte): numero di byte da trasferire;
- CMD** (indirizzo *b* + 8, 4 byte): registro di comando;
- STS** (indirizzo *b* + 12, 4 byte): registro di stato.

L'interfaccia è in grado di trasferire in memoria BMLEN byte, partendo dall'indirizzo fisico contenuto in BMPTR e proseguendo agli indirizzi fisici contigui. Per iniziare il trasferimento è necessario scrivere 1 nel registro di comando. L'interfaccia invia una richiesta di interruzione dopo aver trasferito l'ultimo byte. Le interruzioni sono sempre abilitate e la lettura del registro di stato funziona da risposta alle richieste di interruzione (l'interfaccia non invia una nuova richiesta se una richiesta precedente non ha ancora avuto risposta).

Vogliamo fornire all'utente una primitiva

```
cedmaread(natl id, char *buf, natl quanti)
```

che permetta di leggere quanti byte dalla periferica numero **id** (tra quelle di tipo **ce**) copiandoli nel buffer **buf**. Notare che **buf** è un indirizzo virtuale e il buffer potrebbe attraversare più pagine virtuali: la primitiva dovrà funzionare in ogni caso, eventualmente eseguendo più trasferimenti.

Per descrivere le periferiche **ce** aggiungiamo le seguenti strutture dati al modulo I/O:

```
struct des_ce {
    ioaddr iBMPTR, iBMLEN, iCMD, iSTS;
    natl sync;
    natl mutex;
    char *buf;
    natl quanti;
};

des_ce array_ce[MAX_CE];
natl next_ce;
```

La struttura **des_ce** descrive una periferica di tipo **ce** e contiene al suo interno gli indirizzi dei registri BMPTR, BMLEN, CND e STS, l'indice di un semaforo inizializzato a zero (**sync**), l'indice di un semaforo inizializzato a 1 (**mutex**), il numero di byte che restano da trasferire (**quanti**) e l'indirizzo virtuale a cui vanno trasferiti (**buf**).

I primi `next_ce` elementi del vettore `array_ce` contengono i descrittori, opportunamente inizializzati, delle periferiche di tipo `ce` effettivamente rilevate in fase di avvio del sistema. Ogni periferica è identificata dall'indice del suo descrittore.

Modificare i file `io.s` e `io.cpp` in modo da realizzare la primitiva come descritto.

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

```
// ( SOLUZIONE 2016-06-15
extern "C" void c_cedmaread(natl id, char *buf, natl quanti)
{
    if (id >= next_ce) {
        flog(LOG_WARN, "dispositivo %d non esistente", id);
        abort_p();
    }

    if (!access(buf, quanti, true)) {
        flog(LOG_WARN, "parametri non validi: %p, %d\n", buf, quanti);
        abort_p();
    }

    des_ce *c = &array_ce[id];
    sem_wait(c->mutex);
    paddr f = trasforma(buf);
    natw rem = 4096 - ((natq)f & 0xffff);
    if (rem > quanti)
        rem = quanti;
    flog (LOG_DEBUG, "virtuale %lx fisico %lx primo trasferimento: %d
byte",
          buf, f, rem);
    c->buf = buf + rem;
    c->quanti = quanti - rem;
    outputl((natq)f, c->iBMPTR);
    outputl(rem, c->iBMLEN);
    outputl(1, c->iCMD);
    sem_wait(c->sync);
    sem_signal(c->mutex);
}

extern "C" void estern_ce(int id)
{
    des_ce *c = &array_ce[id];

    for (;;) {
        inputl(c->iSTS);
        if (c->quanti > 0) {
            natw rem = c->quanti;
            if (rem > 4096)
                rem = 4096;
            paddr f = trasforma(c->buf);
            flog (LOG_DEBUG, "virtuale %lx fisico %lx trasferimento:
%d byte",
                  c->buf, f, rem);
            c->buf += rem;
            c->quanti -= rem;
            outputl((natq)f, c->iBMPTR);
            outputl(rem, c->iBMLEN);
            outputl(1, c->iCMD);
        } else {
            sem_signal(c->sync);
        }
        wfi();
    }
}
```

```
//      SOLUZIONE 2016-06-15 )

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

```
// ( SOLUZIONE 2016-06-15
    fill_io_gate    IO_TIPO_CEREAD    a_cedmaread
//  SOLUZIONE 2016-06-15 )
// ( SOLUZIONE 2016-06-15
    .extern    c_cedmaread
a_cedmaread:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_cedmaread
    iretq
    .cfi_endproc
//  SOLUZIONE 2016-06-15 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

6 luglio 2016

- Collegiamo 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 periferiche di ingresso in grado di generare interruzioni. I registri accessibili al programmatore sono i seguenti:

- CTL** (indirizzo *b*, 1 byte): registro di controllo; il bit numero 0 permette di abilitare (1) o disabilitare (0) le richieste di interruzione;
- STS** (indirizzo *b* + 4, 1 byte): registro di stato; il bit numero 0 vale 1 se e solo se il registro RBR contiene un dato non ancora letto;
- RBR** (indirizzo *b* + 8, 1 byte): registro di lettura;

L'interfaccia genera una interruzione se le interruzioni sono abilitate e il registro RBR contiene un valore non ancora letto. L'interfaccia non presenta nuovi valori in RBR se questo ne contiene uno non ancora letto, quindi la lettura di RBR funge da risposta alla richiesta di interruzione.

Vogliamo fornire all'utente una primitiva

```
ceread(nat1 id, char *buf, nat1& quanti, char stop)
```

Il parametro **id** identifica una delle periferiche **ce** installate. La primitiva permette di leggere da tale periferica una sequenza di byte che termina con il carattere **stop** passato come quarto argomento. I byte letti saranno scritti a partire dall'indirizzo **buf**. Il parametro **quanti** è usato sia come argomento di ingresso che di uscita: in ingresso l'utente specifica il numero massimo di byte da leggere (anche se **stop** non è stato ricevuto) e in uscita la primitiva dice all'utente il numero di byte che sono stati effettivamente letti (che può essere inferiore al massimo, quando si riceve **stop**).

Per descrivere le periferiche **ce** aggiungiamo le seguenti strutture dati al modulo I/O:

```
des_ce array_ce[MAX_CE];
nat1 next_ce;
```

I primi **next_ce** elementi del vettore **array_ce** contengono i destrittori, opportunamente inizializzati, delle periferiche di tipo **ce** effettivamente rilevate in fase di avvio del sistema. Ogni periferica è identificata dall'indice del suo descrittore. La struttura **des_ce** deve essere definita dal candidato.

Modificare i file **io.s** e **io.cpp** in modo da realizzare la primitiva come descritto.

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

```
// ( SOLUZIONE 2016-07-06
struct des_ce {
    ioaddr iCTL, iSTS, iRBR;
    natl sync;
    natl mutex;
    char *buf;
    natl quanti;
    char stop;
};
// SOLUZIONE 2016-07-06 )
// ( SOLUZIONE 2016-07-06
extern "C" void c_ceread(natl id, char *buf, natl& quanti, char stop)
{
    if (id >= next_ce) {
        flog(LOG_WARN, "dispositivo %d non esistente", id);
        abort_p();
    }

    if (!access(&quanti, sizeof(quanti), false)) {
        flog(LOG_WARN, "parametro non valido: %p", &quanti);
        abort_p();
    }

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

    des_ce *c = &array_ce[id];
    sem_wait(c->mutex);
    c->buf = buf;
    c->quanti = quanti;
    c->stop = stop;
    outputb(1, c->iCTL);
    sem_wait(c->sync);
    quanti -= c->quanti;
    sem_signal(c->mutex);
}

extern "C" void estern_ce(int id)
{
    des_ce *c = &array_ce[id];

    for (;;) {
        outputb(0, c->iCTL);
        natb b = inputb(c->iRBR);
        *c->buf++ = b;
        c->quanti--;
        if (c->quanti == 0 || b == c->stop) {
            sem_signal(c->sync);
        } else {
            outputb(1, c->iCTL);
        }
        wfi();
    }
}
```

```
//      SOLUZIONE 2016-07-06 )

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

```
// ( SOLUZIONE 2016-07-06
    fill_io_gate    IO_TIPO_CEREAD    a_ceread
//  SOLUZIONE 2016-07-06 )
// ( SOLUZIONE 2016-07-06
    .extern    c_cedmaread
a_ceread:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_ceread
    iretq
    .cfi_endproc
//  SOLUZIONE 2016-07-06 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

27 luglio 2016

- Collegiamo 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 periferiche di ingresso in grado di operare in PCI Bus Mastering. I registri accessibili al programmatore sono i seguenti:

- BMPTR** (indirizzo *b*, 4 byte): puntatore ai descrittori di trasferimento;
- CMD** (indirizzo *b* + 4, 4 byte): registro di comando;
- STS** (indirizzo *b* + 8, 4 byte): registro di stato.

La periferica accumula internamente dei byte da una fonte esterna e ogni volta che si scrive il valore 1 nel registro CMD cerca di trasferirli tutti in memoria. Non è possibile sapere *a-priori* il numero di byte disponibili all'interno della periferica. I byte verranno trasferiti in una sequenza di zone di memoria descritte da un vettore di *descrittori di trasferimento*, il cui indirizzo è contenuto in BMPTR. Ciascun descrittore specifica un indirizzo fisico di partenza e una dimensione. La periferica userà tutte le zone in ordine, fino al trasferimento di tutti i byte disponibili al suo interno. È possibile che le zone non siano sufficienti, nel qual caso i byte in eccesso saranno persi. In ogni caso la periferica invia una richiesta di interruzione al completamento dell'operazione (o perché non ha più byte da trasferire, o perché sono terminate le zone).

Le interruzioni sono sempre abilitate. La lettura del registro di stato funziona da risposta alle richieste di interruzione.

I descrittori di trasferimento hanno la seguente forma:

```
struct ce_buf_des {  
    natl addr;  
    natl len;  
    natb eod;  
    natb eot;  
};
```

Prima di avviare una operazione il campo **addr** deve contenere l'indirizzo fisico di una zona di memoria e **len** la sua dimensione in byte; il campo **eod** deve valere 1 se questo è l'ultimo descrittore. Al completamento dell'operazione la periferica scrive in **len** quanti byte della zona ha utilizzato e scrive 1 in **eot** se con questa zona è riuscita a completare il trasferimento di tutti i byte interni.

Modificare i file **io.s** e **io.cpp** in modo da realizzare la primitiva

```
bool cedmaread(natl id, natl& quanti, char *buf)
```

che permette di leggere al massimo **quanti** byte dalla periferica numero **id** (tra quelle di questo tipo), copiandoli nel buffer **buf**. La primitiva scrive nel parametro **quanti** il numero di byte effettivamente letti. Inoltre, la primitiva restituisce **true** se il buffer è stato sufficiente a contenere tutti i byte da trasferire, e **false** altrimenti.

È un errore se **buf** non è allineato alla pagina e se **quanti** è zero o è più grande di 10 pagine. In caso di errore la primitiva abortisce il processo chiamante. Controllare tutti i problemi di Cavallo di Troia.

Per descrivere le periferiche **ce** aggiungiamo le seguenti strutture dati al modulo I/O:

```
struct des_ce {
    natw iBMPTR, iCMD, iSTS;
    natl sync;
    natl mutex;
    ce_buf_des buf_des[MAX_CE_BUF_DES];
} __attribute__((aligned(128)));
des_ce array_ce[MAX_CE];
natl next_ce;
```

La struttura **des_ce** descrive una periferica di tipo **ce** e contiene al suo interno gli indirizzi dei registri BMPTR, STS e RBR, l'indice di un semaforo inizializzato a zero (**sync**), l'indice di un semaforo inizializzato a 1 (**mutex**) e un vettore di descrittori di trasferimento.

I primi **next_ce** elementi del vettore **array_ce** contengono i destrittori, opportunamente inizializzati, delle periferiche di tipo **ce** effettivamente rilevate in fase di avvio del sistema. Ogni periferica è identificata dall'indice del suo descrittore. Durante l'inizializzazione, il registro BMPTR della periferica viene fatto puntare al campo **buf_des** del suo descrittore.

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

```
// ( SOLUZIONE 2016-07-27
extern "C" bool c_cedmaread(natl id, natl& quanti, char *buf)
{
    if (id >= next_ce) {
        flog(LOG_WARN, "ce non riconosciuto: %d", id);
        abort_p();
    }

    if (!access(&quanti, sizeof(quanti), false)) {
        flog(LOG_WARN, "parametro non valido: %p", &quanti);
        abort_p();
    }

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

    if ((natq)buf & 0xffff) {
        flog(LOG_WARN, "indirizzo %x non allineato alla pagina", buf);
        abort_p();
    }

    if (quanti == 0 || quanti > MAX_CE_BUF_DES * 4096) {
        flog(LOG_WARN, "valore quanti non valido: %d", quanti);
        abort_p();
    }

    des_ce *ce = &array_ce[id];
    sem_wait(ce->mutex);
    flog(LOG_DEBUG, "virt %p len %d", buf, quanti);
    int i;
    for (i = 0; i < MAX_CE_BUF_DES && quanti; i++) {
        natw len = quanti;
        if (len > 4096)
            len = 4096;
        ce->buf_des[i].addr = (natq)trasforma(buf);
        ce->buf_des[i].len = len;
        ce->buf_des[i].eot = ce->buf_des[i].eod = 0;
        quanti -= len;
        buf += len;
        flog(LOG_DEBUG, "des[%d] addr %x len %d", i, ce->buf_des[i].addr, ce->buf_des[i].len);
    }
    ce->buf_des[i - 1].eod = 1;
    outputl(1, ce->iCMD);
    sem_wait(ce->sync);
    quanti = 0;
    int j;
    bool complete = false;
    for (j = 0; j < i; j++) {
        quanti += ce->buf_des[j].len;
        if (ce->buf_des[j].eot) {
            complete = true;
            break;
        }
    }
}
```

```

        }
        sem_signal(ce->mutex);
        return complete;
    }

extern "C" void estern_ce(int id)
{
    des_ce *ce = &array_ce[id];

    for (;;) {
        inputl(ce->iSTS);
        sem_signal(ce->sync);
        wfi();
    }
}
// SOLUZIONE 2016-07-27 )

*****
* io/io.s
*****
// ( SOLUZIONE 2016-07-27
    fill_io_gate    IO_TIPO_CEDMAREAD      a_cedmaread
//  SOLUZIONE 2016-07-27 )
// ( SOLUZIONE 2016-07-27
    .extern      c_cedmaread
a_cedmaread:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_cedmaread
    iretq
    .cfi_endproc
//  SOLUZIONE 2016-07-27 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

20 settembre 2016

1. Introduciamo un meccanismo di *broadcast* tramite il quale un processo può inviare un messaggio ad un insieme di processi. Per ricevere un broadcast i processi si devono preventivamente registrare. Un processo può inviare un broadcast tramite la primitiva `broadcast(msg)`, la quale attende anche che tutti i processi che risultano registrati ricevano il messaggio. Un processo registrato può ricevere un broadcast invocando la primitiva `listen()`, che attende che sia disponibile il prossimo messaggio.

Per realizzare i broadcast introduciamo la seguente struttura dati:

```
struct broadcast {
    int registered;
    int nlisten;
    natl msg;
    des_proc *listeners;
    des_proc *broadcaster;
};
```

Dove: `registered` è il numero di processi registrati; `nlisten` conta i processi che hanno invocato `listen()` dall'ultimo completamento di una operazione di broadcast; `msg` contiene l'ultimo messaggio di broadcast; `listeners` è la coda dei processi in attesa del prossimo messaggio; `broadcaster` è la coda in cui attende il processo che vuole inviare il broadcast, in attesa che tutti i processi registrati invochino `listen()`.

Aggiungiamo inoltre le seguenti primitive (abortiscono il processo in caso di errore):

- `void reg()` (tipo 0x3a, già realizzata): registra il processo per la ricezione dei broadcast; non fa niente se il processo è già registrato;
- `natl listen()` (tipo 0x3b, da realizzare): riceve il prossimo messaggio di broadcast; è un errore se il processo non è registrato;
- `void broadcast(natl msg)` (tipo 0x3c, da realizzare): invia in broadcast il messaggio `msg`; è un errore se il processo è registrato.

Le primitive abortiscono il processo chiamante in caso di errore e tengono conto della priorità tra i processi.

Per semplicità si assuma che, durante tutta la sua esecuzione, ogni processo provi al più una sola volta a inviare un broadcast o ad ascoltare un messaggio. Inoltre, al più un processo alla volta tenta di eseguire un broadcast.

Modificare i file `sistema.cpp` e `sistema.S` in modo da realizzare le primitive mancanti.

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

```
// ( SOLUZIONE 2016-09-20
    carica_gateTIPO_LS           a_listen    LIV_UTENTE
    carica_gateTIPO_B            a_broadcast LIV_UTENTE
//   SOLUZIONE 2016-09-20 )
// ( SOLUZIONE 2016-09-20

a_listen:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_listen
    call carica_stato
    iretq
    .cfi_endproc

a_broadcast:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_broadcast
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2016-09-20 )
```

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

```
// ( SOLUZIONE 2016-09-20
void broadcast_all()
{
    broadcast_t *b = &global_broadcast;
    des_proc *work;

    while (b->listeners) {
        work = rimozione_lista(b->listeners);
        work->contesto[I_RAX] = b->msg;
        inserimento_lista(pronti, work);
    }
    b->nlisten = 0;
}

extern "C" void c_listen()
{
    broadcast_t *b = &global_broadcast;

    if (!esecuzione->listen_reg) {
        flog(LOG_WARN, "listen non registrata");
        c_abort_p();
        return;
    }
}
```

```

    }

    b->nlisten++;
    if (!b->broadcaster) {
        inserimento_lista(b->listeners, esecuzione);
    } else {
        esecuzione->contesto[I_RAX] = b->msg;
        inserimento_lista(pronti, esecuzione);
        if (b->nlisten == b->registered) {
            broadcast_all();
            des_proc *work = rimozione_lista(b->broadcaster);
            inserimento_lista(pronti, work);
        }
    }
    schedulatore();
}

extern "C" void c_broadcast(natl msg)
{
    broadcast_t *b = &global_broadcast;

    if (esecuzione->listen_reg) {
        flog(LOG_WARN, "broadcast da processo listener");
        c_abort_p();
        return;
    }
    b->msg = msg;
    if (b->nlisten == b->registered) {
        inspronti();
        broadcast_all();
    } else {
        inserimento_lista(b->broadcaster, esecuzione);
    }
    schedulatore();
}
// SOLUZIONE 2016-09-20 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

18 gennaio 2017

1. Introduciamo un meccanismo di *broadcast* tramite il quale un processo può inviare un messaggio ad un insieme di processi. Per ricevere o inviare un broadcast i processi si devono preventivamente registrare come *listener* o *broadcaster*, rispettivamente. Un solo processo alla volta può essere registrato come broadcaster (un nuovo processo può diventare broadcaster solo quando il precedente termina).

Il sistema ricorda tutti i messaggi di broadcast inviati (fino ad un massimo dato dalla costante MAX_BROADCAST) e ciascun processo listener li riceve tutti, in ordine. I messaggi sono di tipo natl.

Per realizzare il sistema aggiungiamo il seguente tipo enumerato:

```
enum broadcast_role { B_NONE, B_BROADCASTER, B_LISTENER };
```

e il seguente campo ai descrittori di processo:

```
broadcast_role b_reg;
```

Il campo è posto a B_NONE alla creazione del processo.

Aggiungiamo infine le seguenti primitive:

- **void reg(broadcast_role role)** (tipo 0x3a, da realizzare): registra il processo per il ruolo dato da **role**; è un errore se **role** non specifica né un broadcaster, né un listener, se il processo era già registrato (per lo stesso o un altro ruolo), o se c'è già un broadcaster e si tenta di registrargli un altro;
- **natl listen()** (tipo 0x3b, da realizzare): restituisce il prossimo messaggio di broadcast non ancora letto dal processo; se il processo li ha già letti tutti, si blocca in attesa del prossimo; è un errore se il processo non è registrato come listener;
- **void broadcast(natl msg)** (tipo 0x3c, da realizzare): invia in broadcast il messaggio **msg**; è un errore se il processo non è registrato come broadcaster o se il limite di messaggi di broadcast è stato superato.

Le primitive abortiscono il processo chiamante in caso di errore e tengono conto della priorità tra i processi.

Modificare i file **sistema.cpp** e **sistema.S** in modo da realizzare le primitive mancanti. Attenzione: il candidato deve definire anche le necessarie strutture dati.

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

```
// ( SOLUZIONE 2017-01-18
    carica_gateTIPO_R          a_reg      LIV_UTENTE
    carica_gateTIPO_LS         a_listen   LIV_UTENTE
    carica_gateTIPO_B         a_broadcast LIV_UTENTE
//  SOLUZIONE 2017-01-18 )
// ( SOLUZIONE 2017-01-18
```

```
a_reg:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_reg
    call carica_stato
    iretq
    .cfi_endproc
```

```
a_listen:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_listen
    call carica_stato
    iretq
    .cfi_endproc
```

```
a_broadcast:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_broadcast
    call carica_stato
    iretq
    .cfi_endproc
//  SOLUZIONE 2017-01-18 )
```

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

```
// ( SOLUZIONE 2017-01-18
    natl b_id;
//  SOLUZIONE 2017-01-18 )
// ( SOLUZIONE 2017-01-18
    natl last_id;
    natl msg[MAX_BROADCAST];
    des_proc* listeners;
//  SOLUZIONE 2017-01-18 )
// ( SOLUZIONE 2017-01-18
    global_broadcast.last_id = 0;
```

```

    global_broadcast.listeners = 0;
//  SOLUZIONE 2017-01-18 )
// ( SOLUZIONE 2016-09-20
extern "C" void c_reg(enum broadcast_role role)
{
    broadcast_t *b = &global_broadcast;

    if (role != B_BROADCASTER && role != B_LISTENER) {
        flog(LOG_WARN, "parametro non valido: %d", role);
        c_abort_p();
        return;
    }
    if (esecuzione->b_reg != B_NONE) {
        flog(LOG_WARN, "gia' registrato come %s",
             (esecuzione->b_reg == B_BROADCASTER ? "broadcaster" :
"listener"));
        c_abort_p();
        return;
    }
    if (role == B_BROADCASTER) {
        if (b->broadcaster_registered) {
            flog(LOG_WARN, "broadcaster gia' registrato");
            c_abort_p();
            return;
        }
        b->broadcaster_registered = true;
    }
    esecuzione->b_reg = role;
}

extern "C" void c_listen()
{
    broadcast_t *b = &global_broadcast;

    if (esecuzione->b_reg != B_LISTENER) {
        flog(LOG_WARN, "listen non registrata");
        c_abort_p();
        return;
    }

    if (esecuzione->b_id < b->last_id) {
        esecuzione->contesto[I_RAX] = b->msg[esecuzione->b_id];
        esecuzione->b_id++;
        return;
    }

    inserimento_lista(b->listeners, esecuzione);
    schedulatore();
}

extern "C" void c_broadcast(natl msg)
{
    broadcast_t *b = &global_broadcast;

    if (esecuzione->b_reg != B_BROADCASTER) {
        flog(LOG_WARN, "broadcast da processo non registrato");
        c_abort_p();
        return;
    }
    if (b->last_id >= MAX_BROADCAST) {

```

```
    flog(LOG_WARN, "troppi messaggi");
    c_abort_p();
    return;
}
b->msg[b->last_id] = msg;
b->last_id++;
inspronti();
while (b->listeners) {
    des_proc* work = rimozione_lista(b->listeners);
    work->contesto[I_RAX] = msg;
    work->b_id++;
    inserimento_lista(pronti, work);
}
schedulatore();
}
// SOLUZIONE 2016-09-20 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

18 settembre 2017

1. Vogliamo aggiungere al nucleo un meccanismo tramite il quale un processo sistema può temporaneamente trasformarsi in un processo utente, eseguendo una funzione del modulo utente e riprendendo il controllo quando questa termina (tramite `terminate_p()` o, in caso di errore, `abort_p()`).

Per farlo aggiungiamo la seguente primitiva (invocabile solo da livello sistema):

```
bool call_user(addr f, natq regs[N_REG_GEN], natq *stack, natl n);
```

La primitiva deve saltare a `f` dopo aver riportato il processore a livello utente. Dopo il salto, il contenuto dei registri generali deve essere quello contenuto nell'array `regs` e la pila utente deve contenere le parole quadruple `stack[0]` (in cima) ... `stack[n-1]` (in fondo). Quando la funzione `f` termina (invocando `terminate_p()` o `abort_p()`) l'esecuzione deve tornare al chiamante della `call_user()`, nuovamente a livello sistema; l'array `regs` deve ora contenere i valori dei registri generali al momento della terminazione della funzione `f`. Infine, la primitiva `call_user()` deve restituire `false` se la trasformazione in processo utente non è andata a buon fine (per esempio, non è stato possibile creare la pila utente) e `true` altrimenti.

Per realizzare la primitiva aggiungiamo i seguenti campi al descrittore di processo:

```
natq contesto_salvato[N_REG_GEN];
natq pila_salvata[5];
natq *regs;
```

Dove `contesto_salvato` serve a memorizzare lo stato dei registri generali (corrispondenti ai primi `N_REG_GEN` campi del campo `contesto`) prima di scrivervi il contenuto dell'array `regs` (in modo da poterli ripristinare alla terminazione della funzione utente); il campo `regs` contiene un puntatore all'array omonimo passato alla `call_user()`; si veda sotto per il campo `pila_salvata`.

Si noti che la `call_user()`, come tutte le primitive, sarà invocata tramite una istruzione INT, la quale salverà in pila sistema le 5 parole che una successiva IRETQ potrà estrarre per tornare all'istruzione successiva alla INT stessa. Un puntatore alla prima (dall'alto) di queste parole è contenuto nel campo `contesto[I_RSP]` del descrittore di ogni processo.

Per svolgere il suo compito, la `call_user()` deve modificare opportunamente le 5 parole lunghe salvate dalla INT che l'ha messa in esecuzione, e quindi terminare. Questo deve produrre il salto a livello utente nello stato descritto precedentemente. Le precedenti parole lunghe devono essere memorizzate nel descrittore di processo (nel nuovo campo `pila_salvata`). La `terminate_p()` e la `abort_p()`, quando trovano un valore non-nullo nel campo `regs` del descrittore del processo che le ha invocate, capiscono che questo processo aveva precedentemente invocato una `call_user()` e, invece di distruggerlo, ripristinano il contesto e la pila salvati, in modo da ritornare al chiamante della `call_user()` stessa.

Modificare il file `sistema.cpp` per aggiungere le parti mancanti nella realizzazione di questo meccanismo.

ATTENZIONE: i processi sistema non hanno pila utente e hanno un campo `punt_nucleo` nullo. Per poter trasformare il processo sistema in utente, la `call_user()` dovrà anche creare la pila utente e inizializzare `punt_nucleo`.

SUGGERIMENTO: si tenga presente che la parte di pila sistema che si trova sotto le 5 parole può contenere informazioni utili per il processo sistema che ha invocato la `call_user()`, e dunque non deve essere sovrascritta mentre è in esecuzione la funzione utente.

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

```
// ( SOLUZIONE 2017-09-18
extern "C" void c_call_user(vaddr entry, natq regs[N_REG])
{
    // pila utente
    if (!crea_pila(esecuzione->cr3, fin_utn_p, DIM_USR_STACK,
LIV_UTENTE)) {
        flog(LOG_WARN, "creazione pila utente fallita");
        esecuzione->contesto[I_RAX] = false;
        return;
    }

    // pila sistema
    natq *pl = reinterpret_cast<natq*>(esecuzione->contesto[I_RSP]);
    for (int i = 0; i < 5; i++) {
        esecuzione->pila_salvata[i] = pl[i];
    }
    pl[0] = entry;
    pl[1] = SEL_CODICE_UTENTE;
    pl[2] = BIT_IF;
    pl[3] = fin_utn_p - sizeof(natq);
    pl[4] = SEL_DATI_UTENTE;
    esecuzione->punt_nucleo = reinterpret_cast<vaddr>(&pl[5]);

    // contesto
    regs[I_RSP] = esecuzione->contesto[I_RSP];
    for (int i = 0; i < N_REG; i++) {
        esecuzione->contesto_salvato[i] = esecuzione->contesto[i];
        esecuzione->contesto[i] = regs[i];
    }
    esecuzione->regs = regs;
}
// SOLUZIONE 2017-09-18 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

17 gennaio 2018

1. Vogliamo aggiungere al nucleo un meccanismo tramite il quale un processo può terminare forzatamente l'esecuzione di un altro processo. Aggiungiamo però la seguente limitazione: un processo può terminare solo sé stesso, oppure uno dei suoi *descendenti* (cioè i processi creati da esso stesso, oppure creati da questi, e così via). È un errore se un processo tenta di terminare un altro processo che non appartiene alla sua discendenza.

Per realizzare il meccanismo aggiungiamo la seguente primitiva:

```
bool kill(natl id);
```

La primitiva deve terminare forzatamente il processo di identificatore `id`, qualunque cosa esso stesse facendo (quindi anche se era bloccato su un semaforo o sospeso sulla coda del timer). Se non esiste alcun processo di identificatore `id`, la primitiva restituisce `false`, altrimenti `true`. Abortisce il processo in caso di errore.

Per tenere traccia della discendenza aggiungiamo il seguente campo al descrittore di processo:

```
struct des_proc *parent;
```

Il campo deve puntare al più giovane antenato vivente del processo. Alla creazione, punta al processo padre (quello che ha invocato la `activate_p()`); se il processo padre termina prima del figlio, il campo `parent` dovrà essere opportunamente aggiornato.

Modificare il file `sistema.cpp` per aggiungere le parti mancanti nella realizzazione di questo meccanismo.

ATTENZIONE: quando si fa terminare forzatamente un processo, tutte le sue risorse devono essere rilasciate come il processo se avesse chiamato `terminate_p()`. Fare anche attenzione a lasciare sempre semafori in uno stato consistente.

```

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


// ( SOLUZIONE 2018-01-17
    carica_gateTIPO_KILL  a_kill           LIV_UTENTE
//   SOLUZIONE 2018-01-17 )
// ( SOLUZIONE 2018-01-17

a_kill:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_kill
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2018-01-17 )

*****


* sistema/sistema.cpp
*****


// ( SOLUZIONE 2018-01-17
    des_proc *dp = des_p(id);
    if (dp && dp->parent == p)
        dp->parent = p->parent;
//   SOLUZIONE 2018-01-17 )
// ( SOLUZIONE 2018-01-17
bool descendant(des_proc *parent, des_proc *target)
{
    if (parent == target)
        return true;
    for (des_proc *scan = target->parent; scan; scan = scan->parent)
        if (scan == parent)
            return true;
    return false;
}

extern "C" void c_kill(natl id)
{
    des_proc *dest = des_p(id);

    if (!dest) {
        esecuzione->contesto[I_RAX] = (natq)false;
        return;
    }

    // controllo sulla parentela
    if (!descendant(esecuzione, dest)) {
        flog(LOG_WARN, "%d->kill(%d) non permessa", esecuzione->id,
id);
        c_abort_p();
        return;
    }

    esecuzione->contesto[I_RAX] = (natq)true;
}

```

```

if (esecuzione == dest) {
    flog(LOG_DEBUG, "self kill");
    c_terminate_p();
    return;
}
des_proc* work = estrazione_lista(pronti, id);
if (!work) {
    for (natl sem = 0; sem < MAX_SEM + sem_allocati_sistema;
sem++) {
        des_sem *s = &array_dess[sem];

        work = estrazione_lista(s->pointer, id);
        if (work) {
            s->counter++;
            break;
        }

        if (sem + 1 >= sem_allocati_utente)
            sem = MAX_SEM - 1;
    }
}
if (!work) {
    work = rimozione_lista_attesa(id);
}
if (!work) {
    flog(LOG_ERR, "%d non trovato", id);
    panic("errore interno");
}
distruggi_processo(work);
processi--;           //
flog(LOG_INFO, "Processo %d ucciso da %d", id, esecuzione->id);
}
// SOLUZIONE 2018-01-17 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

7 febbraio 2018

1. Colleghiamo al sistema delle periferiche PCI di tipo **ce**, con vendorID **0xedce** e deviceID **0x1234**. Ogni periferica **ce** usa 32 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione BAR0, sia *b*.

Le periferiche **ce** contengono uno o più contatori indipendenti (fino ad un massimo di 4). Le periferiche decrementano i contatori ogni secondo e inviano una richiesta di interruzione quando uno qualunque di essi raggiunge lo zero. In qualunque momento è possibile scrivere in un contatore, facendo così ripartire il conteggio dal nuovo valore e rimandando la richiesta di interruzione. Scrivendo zero il contatore si ferma, senza inviare interruzioni.

I registri accessibili al programmatore, tutti di 4 byte, sono i seguenti:

1. **CFG** (indirizzo *b*): (sola lettura) specifica il numero di contatori contenuti nel dispositivo; in ogni periferica i contatori sono identificati dai numeri da 0 fino a **CFG – 1**;
2. **SWD** (indirizzo *b + 4*): (scrivibile) scrivendo *i* in questo registro si fa ripartire il contatore *i* dal valore contenuto nel registro CNT;
3. **CNT** (indirizzo *b + 8*): (scrivibile) valore iniziale per il contatore selezionato da SWD;
4. **EWD** (indirizzo *b + 12*): (sola lettura) contiene l'identificatore di un contatore che è arrivato a zero.

Le interruzioni sono sempre abilitate. La lettura del registro EWD funziona da risposta alle richieste di interruzione.

Vogliamo usare queste periferiche per controllare che i processi non impieghino troppo tempo a svolgere le proprie operazioni, per esempio perché entrano in un ciclo infinito a causa di un bug. Prima di iniziare un'operazione critica, un processo avvia un contatore con un certo valore *e*, alla fine, lo ferma. Se l'interruzione arriva prima che il processo riesca a fermare il contatore, il processo viene terminato forzatamente. Ogni contatore può essere utilizzato da un solo processo alla volta. Un processo occupa un contatore da quando lo avvia a quando il contatore arriva a zero (o da solo, o perché il processo lo ferma). È un errore cercare di avviare o fermare un contatore che non esiste, oppure cercare di fermare un contatore che il processo non aveva precedentemente avviato.

Per realizzare questo meccanismo introduciamo le seguenti primitive di livello I/O (abortiscono il processo in caso di errore):

- **natl startwatchdog(natl p, natl secs):** (da realizzare) inizializza uno dei contatori liberi della periferica *p* con il valore *secs* e ne restituisce l'identificatore; restituisce **0xffffffff** se tutti i contatori della periferica sono occupati.
- **void stopwatchdog(natl p, natl wd):** (da realizzare) ferma il contatore *wd* della periferica *p*.

Modificare i file **io.s** e **io.cpp** in modo da realizzare le due primitive.

Nota: il modulo sistema mette a disposizione la primitiva **natl getpid()** per ottenere l'identificatore del processo attualmente in esecuzione; la primitiva **void kill(natl id)** può essere usata per terminare forzatamente il processo di identificatore *id*.

```

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


// ( SOLUZIONE 2018-02-07
struct ce_counter {
    bool busy;
    natl proc;
};

// ( SOLUZIONE 2018-02-07 )
// ( SOLUZIONE 2018-02-07
natl mutex;
ce_counter counters[MAX_CE_CTR];
natl nctr;
// ( SOLUZIONE 2018-02-07 )
// ( SOLUZIONE 2018-02-07
natl findfreewd(des_ce *ce)
{
    for (natl i = 0; i < ce->nctr; i++)
        if (!ce->counters[i].busy) {
            ce->counters[i].busy = true;
            return i;
        }
    return 0xffffffff;
}

extern "C" natl c_startwatchdog(natl id, natl secs)
{
    if (id >= next_ce) {
        flog(LOG_WARN, "ce: id %d non valido", id);
        abort_p();
    }
    des_ce *ce = &array_ce[id];
    sem_wait(ce->mutex);
    natl wd = findfreewd(ce);
    if (wd == 0xffffffff) {
        sem_signal(ce->mutex);
        return wd;
    }
    ce_counter *c = &ce->counters[wd];
    c->proc = getpid();
    outputl(secs, ce->iCNT);
    outputl(wd, ce->iSWD);
    sem_signal(ce->mutex);
    return wd;
}

extern "C" void c_stopwatchdog(natl id, natl wd)
{
    if (id >= next_ce) {
        flog(LOG_WARN, "ce: id %d non valido", id);
        abort_p();
    }
    des_ce *ce = &array_ce[id];
    if (wd >= ce->nctr) {
        flog(LOG_WARN, "ce %d: wd %d non valido", id, wd);
        abort_p();
    }
    ce_counter *c = &ce->counters[wd];
    sem_wait(ce->mutex);
}

```

```

if (!c->busy || !(c->proc == getpid())) {
    flog(LOG_WARN, "ce %d: wd %d non valido", id, wd);
    sem_signal(ce->mutex);
    abort_p();
}
outputl(0, ce->iCNT);
outputl(wd, ce->iSWD);
c->busy = false;
sem_signal(ce->mutex);
}

extern "C" void estern_ce(int id)
{
    des_ce *ce = &array_ce[id];
    natl wd;

    for (;;) {
        wd = inputl(ce->iEWD);
        ce_counter *c = &ce->counters[wd];
        sem_wait(ce->mutex);
        if (c->busy) {
            kill(c->proc);
            c->busy = false;
        }
        sem_signal(ce->mutex);
        wfi();
    }
}
// SOLUZIONE 2018-02-07 )
// ( SOLUZIONE 2018-02-07
    if ((ce->mutex = sem_ini(1)) == 0xFFFFFFFF)
        return false;
    ce->nctr = inputl(ce->iCFG);
    flog(LOG_INFO, "ce%d: %d counters", next_ce, ce->nctr);
    for (natl i = 0; i < ce->nctr; i++) {
        ce->counters[i].busy = false;
    }
// SOLUZIONE 2018-02-07 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

23 febbraio 2018

1. Vogliamo aggiungere al sistema una primitiva

```
natl newproc(void f(int), int a, natl prio, natl secs)
```

La primitiva è simile alla `activate_p()` e permette di creare un nuovo processo con corpo `f` e parametro `a`, eseguito a priorità `prio`. A differenza della `activate_p()`, però, il nuovo processo avrà sempre privilegio utente e dovrà terminare (cioè, invocare la primitiva `terminate_p()`) entro `secs` secondi, altrimenti verrà terminato forzatamente. Come la `activate_p()`, la primitiva `newproc()` restituisce l'identificatore del processo creato o `0xffffffff` se non è stato possibile portare a termine la creazione.

Per motivi di convenienza realizziamo la nuova primitiva nel modulo I/O e sfruttiamo i seguenti meccanismi già disponibili nel modulo I/O e nel modulo sistema:

- modulo I/O: possiamo usare le funzioni `natl startwatchdog(natl secs)` e `stopwatchdog(natl wd)`;
- modulo sistema: possiamo usare la primitiva `bool call_user(void f(int), int a)`.

La funzione `startwatchdog(secs)` avvia un timer di `secs` secondi e ne restituisce l'identificatore (`0xffffffff` se non vi sono più timer disponibili). La funzione `stopwatchdog(wd)` ferma il timer di identificatore `wd`. Se il timer non viene fermato in tempo (cioè, prima che siano passati i `secs` secondi con cui era stato inizializzato), il processo che aveva invocato `startwatchdog()` viene terminato forzatamente.

La primitiva `call_user(f, a)`, invocabile solo dal modulo I/O, trasforma temporaneamente il processo corrente in un processo utente, passando ad eseguire il corpo `f` con parametro `a`. Se e quando il corpo `f` chiama `terminate_p()`, la primitiva `call_user()` torna al chiamante e l'esecuzione prosegue a livello sistema. Se invece il processo utente viene terminato forzatamente, l'intero processo termina la propria esecuzione.

Modificare i file `io.s` e `io.cpp` in modo da realizzare la primitiva `newproc()`.

```

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


// ( SOLUZIONE 2018-02-23
struct newproc_param_t {
    void (*f)(int);
    int a;
    int s;
    natl avail;
} newproc_params;

void proxy(natq)
{
    void (*f)(int);
    int a, s;

    f = newproc_params.f;
    a = newproc_params.a;
    s = newproc_params.s;
    sem_signal(newproc_params.avail);

    natl wd = startwatchdog(s);
    if (wd == 0xffffffff) {
        flog(LOG_WARN, "nessun watchdog disponibile");
        abort_p();
    }
    if (!call_user(f, a)) {
        flog(LOG_WARN, "call_user fallita");
        stopwatchdog(wd);
        abort_p();
    }
    stopwatchdog(wd);
    terminate_p();
}

extern "C" natl c_newproc(void f(int), int a, natl prio, natl secs)
{
    sem_wait(newproc_params.avail);
    newproc_params.f = f;
    newproc_params.a = a;
    newproc_params.s = secs;
    natl id = activate_p(proxy, 0, prio, LIV_SISTEMA);
    return id;
}

bool newproc_init()
{
    newproc_params.avail = sem_ini(1);
    return (newproc_params.avail != 0xFFFFFFFF);
}
// SOLUZIONE 2018-02-23 )

*****
```

```

        fill_io_gate    IO_TIPO_NP a_newproc
//  SOLUZIONE 2018-02-23 )
// ( SOLUZIONE 2018-02-23
    .extern c_newproc
a_newproc:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_newproc
    iretq
    .cfi_endproc
//  SOLUZIONE 2018-02-23 )

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


// ( SOLUZIONE 2018-02-23
extern "C" void c_call_user(vaddr entry, int a)
{
    // pila utente
    if (!crea_pila(esecuzione->cr3, fin_utn_p, DIM_USR_STACK,
LIV_UTENTE)) {
        flog(LOG_WARN, "creazione pila utente fallita");
        esecuzione->contesto[I_RAX] = false;
        return;
    }

    // pila sistema
    natq *pl = reinterpret_cast<natq*>(esecuzione->contesto[I_RSP]);
    for (int i = 0; i < 5; i++) {
        esecuzione->pila_salvata[i] = pl[i];
    }
    pl[0] = entry;
    pl[1] = SEL_CODICE_UTENTE;
    pl[2] = BIT_IF;
    pl[3] = fin_utn_p - sizeof(natq);
    pl[4] = SEL_DATI_UTENTE;
    esecuzione->punt_nucleo = reinterpret_cast<vaddr>(&pl[5]);

    // contesto
    for (int i = 0; i < N_REG; i++) {
        esecuzione->contesto_salvato[i] = esecuzione->contesto[i];
    }
    esecuzione->livello = LIV_UTENTE;
    esecuzione->downgraded = true;
    esecuzione->contesto[I_RDI] = a;
    esecuzione->contesto[I_RAX] = true;
}
//  SOLUZIONE 2018-02-23 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

13 giugno 2018

1. Collegiamo al sistema una periferica PCI di tipo **ce**, con vendorID **0xedce** e deviceID **0x1234**. Le periferiche **ce** sono schede di rete che operano in PCI Bus Mastering. Il software deve preparare i messaggi in dei buffer di memoria e la scheda è in grado di spedirli autonomamente.

Per permettere al software di operare in parallelo con l'invio, la scheda usa una coda circolare di descrittori di messaggi. Ogni descrittore deve contenere l'indirizzo fisico di un messaggio e la sua lunghezza in byte. La coda ha 8 posizioni, numerate da 0 a 7. La scheda possiede due registri, **HEAD**, di sola lettura, e **TAIL**, di lettura/scrittura.

I due registri contengono numeri di posizioni e inizialmente sono entrambi pari a zero. Per inviare un nuovo messaggio il software deve: 1) inizializzare il descrittore numero **TAIL**; 2) incrementare il di 1 (modulo 8) il contenuto di **TAIL**. Più messaggi possono essere accodati mentre altri (accodati precedentemente) sono ancora in fase di invio da parte della scheda. Ogni volta che la scheda ha terminato di inviare un messaggio incrementa (modulo 8) il contenuto di **HEAD** e, se non sta aspettando una risposta ad una richiesta di interruzione precedente, invia una nuova richiesta di interruzione. La lettura di **HEAD** funge da risposta alla richiesta. È dunque possibile (e, anzi, normale) che quando il software legge **HEAD** questo sia avanzato di più di una posizione rispetto all'ultima lettura: vuol semplicemente dire che tra le due letture la scheda ha finito di inviare più di un messaggio. I descrittori dei messaggi inviati saranno quelli che si trovano tra l'ultima posizione letta da **HEAD** (inclusa) e la nuova (esclusa).

Ogni periferica **ce** usa 16 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione **BAR0**, sia *b*. I registri accessibili al programmatore sono i seguenti:

1. **HEAD** (indirizzo *b*, 4 byte): posizione di testa;
2. **TAIL** (indirizzo *b* + 4, 4 byte): posizione di coda;
3. **RING** (indirizzo *b* + 8, 4 byte): indirizzo fisico del primo descrittore della coda circolare;

Supponiamo che ogni computer collegato alla rete possieda un indirizzo numerico di 4 byte. Ogni computer possiede un'unica perifera di tipo **ce**. Vogliamo fornire all'utente una primitiva

```
bool send(nat1 dst, char *msg, nat1 len)
```

che permetta di inviare il messaggio puntato da **msg**, e lungo **len** byte, al computer di indirizzo **dst**. Per far questo, la primitiva alloca un buffer e vi scrive prima una “intestazione” con tre campi (ciascuno grande 4 byte): l'indirizzo del computer che invia (preso da una variabile globale, **myaddr**), l'indirizzo **dst** e la lunghezza **len**. Subito dopo l'intestazione, la primitiva copia nel buffer il messaggio **msg**. Infine, invia il buffer usando la periferica **ce**. La lunghezza del messaggio dell'utente (esclusa l'intestazione) deve essere inferiore a **MAX_PAYLOAD**. La primitiva restituisce **false** in caso di messaggio troppo lungo o se non è stato possibile allocare il buffer, o se la coda circolare era piena; restituisce **true** altrimenti. La primitiva ritorna *senza attendere* che il messaggio sia stato spedito. La memoria allocata per il buffer dovrà essere liberata *dopo* che il messaggio è stato effettivamente spedito.

Per descrivere le periferiche **ce** aggiungiamo le seguenti strutture dati al modulo I/O:

```

struct slot {
    natl addr;
    natl len;
};

struct des_ce {
    natw iHEAD, iTAIL, iRING;
    natl mutex;
    slot *s;
// altri campi
} net;

```

La struttura `slot` rappresenta un descrittore di buffer (indirizzo fisico in `addr` e lunghezza in `len`). La struttura `des_ce` descrive una periferica di tipo `ce` e contiene al suo interno gli indirizzi dei registri HEAD, TAIL e RING, il puntatore `s` alla coda circolare di descrittori, l'indice di un semaforo inizializzato a 1 (`mutex`) ed eventuali altri campi che dovessero essere utili.

Modificare i file `io.S` e `io.cpp` in modo da realizzare la primitiva come descritto.

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

```
// ( SOLUZIONE 2018-06-13
    natl old_head;
    msg* m[DIM_RING];
//  SOLUZIONE 2018-06-13 )
// ( SOLUZIONE 2018-06-13
extern "C" bool c_send(natl dst, const char *buf, natl len)
{
    if (len > MAX_PAYLOAD)
        return false;

    if (!access(buf, len, false)) {
        flog(LOG_WARN, "parametri errati: %p %d", buf, len);
        abort_p();
    }

    sem_wait(net.mutex);

    natl tail = inputl(net.iTAIL);

    if (ring_full(tail, net.old_head)) {
        sem_signal(net.mutex);
        return false;
    }

    msg *m = new(align_val_t{1024}) msg;
    if (!m) {
        sem_signal(net.mutex);
        return false;
    }

    m->src = myaddr;
    m->dst = dst;
    m->len = len;
    memcpy(m->payload, buf, len);
    /* ricordiamo l'indirizzo virtuale del buffer
     * in modo da poterlo poi liberare
     */
    net.m[tail] = m;

    slot *s = &net.s[tail];
    s->addr = trasforma(m);
    s->len = len + 3 * sizeof(natl);
    tail = ring_next(tail);
    outputl(tail, net.iTAIL);

    sem_signal(net.mutex);

    return true;
}

extern "C" void estern_net(int i)
{
    for (;;) {
        natl new_head = inputl(net.iHEAD);
        sem_wait(net.mutex);
        for (natl h = net.old_head; h != new_head; h = ring_next(h)) {
```

```

        delete net.m[h];
        net.m[h] = nullptr;
    }
    net.old_head = new_head;
    sem_signal(net.mutex);
    wfi();
}
}

// non usare: serve al programma di test
extern "C" natl c_waitnet()
{
    for (;;) {
        if (inputl(net.iHEAD) == inputl(net.iTAIL))
            break;
        delay(10);
    }
    return inputl(net.iRING + 4);
}
// SOLUZIONE 2018-06-13 )

```

```

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


// ( SOLUZIONE 2018-06-13
    fill_io_gate    IO_TIPO_SEND      a_send
//   SOLUZIONE 2018-06-13 )
// ( SOLUZIONE 2018-06-13
    .extern c_send
a_send:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_send
    iretq
    .cfi_endproc
//   SOLUZIONE 2018-06-13 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

4 luglio 2018

1. Collegiamo al sistema una periferica PCI di tipo **ce**, con vendorID **0xedce** e deviceID **0x1234**. Le periferiche **ce** sono schede di rete che operano in PCI Bus Mastering. Il software deve preparare dei buffer vuoti, che la scheda riempie autonomamente con messaggi ricevuti dalla rete.

Per permettere al software di operare in parallelo con la ricezione, la scheda usa una coda circolare di descrittori di buffer. Ogni descrittore deve contenere l'indirizzo fisico di un buffer e la sua lunghezza in byte. La coda di buffer ha 8 posizioni, numerate da 0 a 7.

La scheda possiede due registri, **HEAD**, di sola lettura, e **TAIL**, di lettura/scrittura. I due registri contengono numeri di posizioni e inizialmente sono entrambi pari a zero. La scheda può usare soltanto i descrittori che vanno da **HEAD** in avanti (circolarmente) senza toccare **TAIL**. Inizialmente, dunque, la scheda non può usare alcun descrittore. Il software deve allocare dei buffer a partire dal descrittore puntato da **TAIL** e poi scrivere in **TAIL** l'indice del primo descrittore che la scheda non può usare. Conviene allocare sempre il massimo numero possibile di buffer, perché la scheda butta via i messaggi ricevuti quando non ha a disposizione buffer in cui copiarli. Si noti che il massimo numero di descrittori che la scheda può usare è pari a 7 (dimensione della coda meno 1), in quanto la configurazione con **HEAD** uguale a **TAIL** è interpretata dalla scheda come “coda vuota”.

Ogni volta che la scheda ha terminato di ricevere un messaggio incrementa (modulo 8) il contenuto di **HEAD** e, se non sta aspettando una risposta ad una richiesta di interruzione precedente, invia una nuova richiesta di interruzione. La lettura di **HEAD** funge da risposta alla richiesta. È dunque possibile (e, anzi, normale) che quando il software legge **HEAD** questo sia avanzato di più di una posizione rispetto all'ultima lettura: vuol semplicemente dire che tra le due letture la scheda ha finito di ricevere più di un messaggio. I descrittori dei messaggi ricevuti saranno quelli che si trovano tra l'ultima posizione letta da **HEAD** (inclusa) e la nuova (esclusa).

Ogni periferica **ce** usa 16 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione BAR0, sia *b*. I registri accessibili al programmatore sono i seguenti:

1. **HEAD** (indirizzo *b*, 4 byte): posizione di testa;
2. **TAIL** (indirizzo *b* + 4, 4 byte): posizione di coda;
3. **RING** (indirizzo *b* + 8, 4 byte): indirizzo fisico del primo descrittore della coda circolare;

Supponiamo che ogni computer collegato alla rete possieda un indirizzo numerico di 4 byte. Ogni computer possiede un'unica periferica di tipo **ce**. I messaggi che viaggiano sulla rete contengono una “intestazione” con tre campi (ciascuno grande 4 byte): l'indirizzo del computer che invia, l'indirizzo **dst** del computer a cui il messaggio è destinato, e la lunghezza **len** del resto del messaggio (esclusa l'intestazione). L'intestazione è seguita dal messaggio vero e proprio, di lunghezza massima **MAX_PAYLOAD**.

Vogliamo fornire all'utente una primitiva

```
bool receive(nat1& src, char *msg, natq& len)
```

che permetta di ricevere un messaggio nel buffer `msg`. Il parametro `len` contiene inizialmente la dimensione del buffer e, dopo la ricezione, contiene la dimensione effettiva del messaggio ricevuto. Il parametro `src` conterrà l'indirizzo del computer che ha inviato il messaggio. Attenzione: l'utente deve ricevere in `msg` solo il messaggio vero e proprio, esclusa l'intestazione. Si noti che ogni invocazione della primitiva restituisce un solo messaggio: eventuali altri messaggi in coda verranno restituiti alla prossime invocazioni. Se, invece, non vi sono messaggi in coda, la primitiva blocca il processo in attesa che ne arrivi almeno uno. La primitiva restituisce `false` se il buffer `msg` non è sufficiente a contenere il prossimo messaggio da ricevere, e `true` altrimenti.

Per descrivere le periferiche `ce` aggiungiamo le seguenti strutture dati al modulo I/O:

```
struct slot {
    natl addr;
    natl len;
};

const natl DIM_RING = 8;
struct des_ce {
    natw iHEAD, iTAIL, iRING;
    natl mutex;
    natl messages;
    slot s[DIM_RING];
    natl toread;
    natl old_head;
} net;
```

La struttura `slot` rappresenta un descrittore di buffer (indirizzo fisico in `addr` e lunghezza in `len`). La struttura `des_ce` descrive una periferica di tipo `ce` e contiene al suo interno: gli indirizzi dei registri HEAD, TAIL e RING; la coda circolare di descrittori, `s`; l'indice di un semaforo di mutua esclusione (`mutex`); l'indice di un semaforo `messages`, inizializzato a zero; il campo `old_head`, utile a memorizzare l'ultimo valore letto dal registro HEAD; il campo `toread`, utile a memorizzare l'indice del prossimo buffer da leggere tramite la `receive`.

Si può assumere che la scheda riceva solo messaggi effettivamente destinati al computer a cui è collegata. Modificare i file `io.s` e `io.cpp` in modo da realizzare la primitiva come descritto.

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

```
// ( SOLUZIONE 2018-07-04
extern "C" bool c_receive(natl& src, char *buf, natq& len)
{
    bool rv = true;
    natl tail;

    if (!access(&src, sizeof(src), true) ||
        !access(&len, sizeof(len), true) ||
        !access(buf, len, true))
    {
        flog(LOG_WARN, "parametri non validi");
        abort_p();
    }

    sem_wait(net.mutex);

    sem_wait(net.messages);

    slot *s = &net.s[net.toread];
    msg *m = to_msg(s->addr);
    if (m->len > len) {
        rv = false;
        goto out;
    }
    src = m->src;
    len = m->len;
    memcpy(buf, m->payload, len);
    net.toread = ring_next(net.toread);

    /* diamo alla scheda il permesso di usare un altro buffer */
    tail = inputl(net.iTAIL);
    tail = ring_next(tail);
    outputl(tail, net.iTAIL);

out:
    sem_signal(net.mutex);

    return rv;
}
// SOLUZIONE 2018-07-04 )
// ( SOLUZIONE 2018-07-04
/* lo riempiamo di buffer (riuseremo sempre gli stessi) */
for (natl i = 0; i < DIM_RING; i++) {
    natl size = MAX_PAYLOAD + 3 * sizeof(natl);
    void* buf = new(aligned_val_t{4096}) char[size];
    if (!buf) {
        flog(LOG_ERR, "memoria insufficiente");
        return false;
    }
    net.s[i].addr = trasforma(buf);
    net.s[i].len = size;
    flog(LOG_DEBUG, "slot %d: addr %p size %d", i, net.s[i].addr,
net.s[i].len);
}
/* inizializziamo TAIL in modo che il ring risulti pieno */
outputl(DIM_RING - 1, net.iTAIL);
```

// SOLUZIONE 2018-07-04)

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

25 luglio 2018

1. Collegiamo al sistema una periferica PCI di tipo **ce**, con vendorID **0xedce** e deviceID **0x1234**. Le periferiche **ce** sono schede di rete che operano in PCI Bus Mastering. Il software deve preparare dei buffer vuoti, che la scheda riempie autonomamente con messaggi ricevuti dalla rete.

La scheda usa una coda circolare di descrittori di buffer. Ogni descrittore deve contenere l'indirizzo fisico di un buffer e la sua lunghezza in byte. La coda di buffer ha 8 posizioni, numerate da 0 a 7.

La scheda possiede due registri, **HEAD**, di sola lettura, e **TAIL**, di lettura/scrittura. I due registri contengono numeri di posizioni e inizialmente sono entrambi pari a zero. La scheda può usare soltanto i descrittori che vanno da **HEAD** in avanti (circolarmente) senza toccare **TAIL**. Inizialmente, dunque, la scheda non può usare alcun descrittore. Il software deve allocare dei buffer a partire dal descrittore puntato da **TAIL** e poi scrivere in **TAIL** l'indice del primo descrittore che la scheda non può usare. Conviene allocare sempre il massimo numero possibile di buffer, perché la scheda butta via i messaggi ricevuti quando non ha a disposizione buffer in cui copiarli. Si noti che il massimo numero di descrittori che la scheda può usare è pari a 7 (dimensione della coda meno 1), in quanto la configurazione con **HEAD** uguale a **TAIL** è interpretata dalla scheda come “coda vuota”.

Ogni volta che la scheda ha terminato di ricevere un messaggio incrementa (modulo 8) il contenuto di **HEAD** e, se non sta aspettando una risposta ad una richiesta di interruzione precedente, invia una nuova richiesta di interruzione. La lettura di **HEAD** funge da risposta alla richiesta. È dunque possibile (e, anzi, normale) che quando il software legge **HEAD** questo sia avanzato di più di una posizione rispetto all'ultima lettura: vuol semplicemente dire che tra le due letture la scheda ha finito di ricevere più di un messaggio. I descrittori dei messaggi ricevuti saranno quelli che si trovano tra l'ultima posizione letta da **HEAD** (inclusa) e la nuova (esclusa).

Ogni periferica **ce** usa 16 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione BAR0, sia *b*. I registri accessibili al programmatore sono i seguenti:

1. **HEAD** (indirizzo *b*, 4 byte): posizione di testa;
2. **TAIL** (indirizzo *b* + 4, 4 byte): posizione di coda;
3. **RING** (indirizzo *b* + 8, 4 byte): indirizzo fisico del primo descrittore della coda circolare;

Supponiamo che ogni computer collegato alla rete possieda un indirizzo numerico di 4 byte. Ogni computer possiede un'unica perifera di tipo **ce**. I messaggi che viaggiano sulla rete contengono una “intestazione” con quattro campi (ciascuno grande 4 byte): l'indirizzo del computer che invia, l'indirizzo **dst** del computer a cui il messaggio è destinato, un numero di *porta* (compreso tra 0 e 15) e la lunghezza **len** del resto del messaggio (esclusa l'intestazione). L'intestazione è seguita dal messaggio vero e proprio, di lunghezza massima **MAX_PAYLOAD**.

Vogliamo fornire all'utente una primitiva

```
bool receive(nat1 port, char *msg, natq& len)
```

che permetta di ricevere nel buffer `msg` un messaggio destinato alla porta `port`. Il parametro `len` contiene inizialmente la dimensione del buffer e, dopo la ricezione, contiene la dimensione effettiva del messaggio ricevuto. Attenzione: l'utente deve ricevere in `msg` solo il messaggio vero e proprio, esclusa l'intestazione. Si noti che ogni invocazione della primitiva restituisce un solo messaggio: eventuali altri messaggi in coda verranno restituiti alla prossime invocazioni. Se, invece, non vi sono messaggi in coda, la primitiva blocca il processo in attesa che ne arrivi almeno uno. La primitiva restituisce `false` se il buffer `msg` non è sufficiente a contenere il prossimo messaggio da ricevere o se `port` non è valido, e `true` altrimenti.

Si può assumere che la scheda riceva solo messaggi effettivamente destinati al computer a cui è collegata. Modificare i file `io.s` e `io.cpp` completando le parti mancanti.

NOTA: le strutture dati necessarie sono tutte descritte nel file `io.cpp`, all'interno dei marcatori `ESAME`.

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

```
// ( SOLUZIONE 2018-07-25
    msg *m[DIM_RING];
    natl old_head;
//  SOLUZIONE 2018-07-25 )
// ( SOLUZIONE 2018-07-25
    if (!access(&len, sizeof(len), true) ||
        !access(buf, len, true))
    {
        flog(LOG_WARN, "parametri errati");
        abort_p();
    }
//  SOLUZIONE 2018-07-25 )
// ( SOLUZIONE 2018-07-25
    for (;;) {
        natl new_head = inputl(net.iHEAD);
        natl tail = inputl(net.iTAIL);
        flog(LOG_DEBUG, "HEAD %d (old %d) TAIL %d", new_head,
net.old_head, tail);
        for (natl scan = net.old_head ; scan != new_head; scan =
ring_next(scan)) {
            msg *m = net.m[scan];
            flog(LOG_DEBUG, "scan %d port %d", scan, m->port);
            if (m->port >= MAX_PORTS)
                continue;
            des_port *p = &net.ports[m->port];
            sem_wait(p->mutex);
            accoda_msg(p, m);
            sem_signal(p->mutex);
            sem_signal(p->messages);
            void* buf = new(align_val_t{4096}) msg;
            if (!buf) {
                panic("memoria insufficiente");
            }
            net.ring[scan].addr = trasforma(buf);
            net.ring[scan].len = sizeof(msg);
            net.m[scan] = static_cast<msg*>(buf);
            tail = ring_next(tail);
        }
        net.old_head = new_head;
        outputl(tail, net.iTAIL);
        wfi();
    }
//  SOLUZIONE 2018-07-25 )
// ( SOLUZIONE 2018-07-25
    /* comunichiamo l'indirizzo del ring all'interfaccia */
    outputl(trasforma(net.ring), net.iRING);
    /* lo riempiamo di buffer (riuseremo sempre gli stessi) */
    for (natl i = 0; i < DIM_RING; i++) {
        void* buf = new(align_val_t{4096}) msg;
        if (!buf) {
            flog(LOG_ERR, "memoria insufficiente");
            return false;
        }
        net.ring[i].addr = trasforma(buf);
        net.ring[i].len = sizeof(msg);
        net.m[i] = static_cast<msg*>(buf);
```

```

        flog(LOG_DEBUG, "slot %d: addr %p size %d", i,
net.ring[i].addr, net.ring[i].len);
    }
/* inizializziamo TAIL in modo che il ring risulti pieno */
outputl(DIM_RING - 1, net.iTAIL);

/* inizializziamo i semafori */
for (natl i = 0; i < MAX_PORTS; i++) {
    des_port *p = &net.ports[i];
    p->mutex = sem_ini(1);
    if (p->mutex == 0xFFFFFFFF) {
        flog(LOG_ERR, "semafori insufficienti");
        return false;
    }
    p->messages = sem_ini(0);
    if (p->messages == 0xFFFFFFFF) {
        flog(LOG_ERR, "semafori insufficienti");
        return false;
    }
}
}

// SOLUZIONE 2018-07-25 )

```

```

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

// ( SOLUZIONE 2018-07-25
    .extern c_waitnet
a_waitnet:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_waitnet
    iretq
    .cfi_endproc

    .extern c_send
a_receive:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_receive
    iretq
    .cfi_endproc
// SOLUZIONE 2018-07-25 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

19 settembre 2018

1. Collegiamo al sistema una periferica PCI di tipo **ce**, con vendorID **0xedce** e deviceID **0x1234**. Le periferiche **ce** sono schede di rete che operano in PCI Bus Mastering. Il software deve preparare dei “pacchetti” in dei buffer di memoria e la scheda è in grado di spedirli autonomamente. Ogni buffer ha una dimensione massima di 64 byte, e dunque questa è anche la massima dimensione di ogni pacchetto.

Per permettere al software di operare in parallelo con l’invio, la scheda usa una coda circolare di descrittori di buffer. Ogni descrittore deve contenere l’indirizzo fisico di un buffer e la sua lunghezza in byte. La coda ha 8 posizioni, numerate da 0 a 7. La scheda possiede due registri, **HEAD**, di sola lettura, e **TAIL**, di lettura/scrittura.

I due registri contengono numeri di posizioni e inizialmente sono entrambi pari a zero. È possibile inviare più di un pacchetto per volta. Per inviare n pacchetti il software deve: 1) inizializzare i descrittori dal numero **TAIL** al numero **TAIL + n - 1**; 2) incrementare di n (modulo 8) il contenuto di **TAIL**. Ogni volta che la scheda ha terminato di inviare uno o più pacchetti, incrementa (modulo 8) il contenuto di **HEAD** e, se non sta aspettando una risposta ad una richiesta di interruzione precedente, invia una nuova richiesta di interruzione. La lettura di **HEAD** funge da risposta alla richiesta.

Ogni periferica **ce** usa 16 byte nello spazio di I/O a partire dall’indirizzo base specificato nel registro di configurazione BAR0, sia b . I registri accessibili al programmatore sono i seguenti:

1. **HEAD** (indirizzo b , 4 byte): posizione di testa;
2. **TAIL** (indirizzo $b + 4$, 4 byte): posizione di coda;
3. **RING** (indirizzo $b + 8$, 4 byte): indirizzo fisico del primo descrittore della coda circolare;

Supponiamo che ogni computer collegato alla rete possieda un indirizzo numerico di 4 byte. Ogni computer possiede un’unica perifera di tipo **ce**. Vogliamo fornire all’utente una primitiva

```
bool send(nat1 dst, char *msg, nat1 len)
```

che permetta di inviare il messaggio puntato da **msg**, e lungo **len** byte, al computer di indirizzo **dst**. La primitiva permette all’utente di inviare messaggi più grandi del massimo permesso dalla dimensione dei buffer della scheda, preparando e inviando una *sequenza* di pacchetti, ciascuno dei quali contiene parte del messaggio. Non è comunque possibile inviare un messaggio che richieda più pacchetti di quanti ne possono essere contenuti in un ring: in questo caso, la primitiva abortisce il processo.

I pacchetti sono composti da una “intestazione”, creata dalla primitiva, seguita dai byte che l’utente vuole inviare, presi da **msg**. L’intestazione ha quattro campi, ciascuno grande 4 byte: l’indirizzo del computer che invia (preso da una variabile globale, **myaddr**), l’indirizzo **dst**, il campo **len**, che contiene il numero di byte del messaggio contenuti in questo pacchetto, e il campo **seq**, che contiene un numero progressivo, a partire da 0, per ogni pacchetto relativo allo stesso messaggio.

I buffer destinati a contenere i pacchetti sono allocati dalla primitiva stessa e la primitiva restituisce **false** se una allocazione fallisce. La primitiva attende (eventualmente sospendendo il processo) che vi sia un numero di descrittori liberi sufficiente per tutti i pacchetti da inviare, prepara tutti i pacchetti e ordina

alla scheda di inviarli, ma poi ritorna *senza attendere* che siano stati effettivamente spediti. La memoria allocata per ogni buffer dovrà essere liberata *dopo* che il pacchetto contenuto è stato effettivamente spedito.

Per descrivere le periferiche **ce** aggiungiamo le seguenti strutture dati al modulo I/O:

```
struct slot {
    natl addr;
    natl len;
};

struct des_net {
    natw iHEAD, iTAIL, iRING;
    slot s[DIM_RING];
    natl mutex;
    natl sync;
};

natl old_head;
bool sender_waiting;
msg *m[DIM_RING];
} net;
```

La struttura **slot** rappresenta un descrittore di buffer (indirizzo fisico in **addr** e lunghezza in **len**). La struttura **des_net** descrive una periferica di tipo **ce** e contiene al suo interno gli indirizzi dei registri HEAD, TAIL e RING, la coda circolare di descrittori **s**, l'indice di un semaforo inizializzato a 1 (**mutex**) di uno inizializzato a 0 (**sync**). Il capo **old_head** contiene il valore letto dal registro HEAD nell'ultima interruzione. mentre l'array **m** serve a ricordare gli indirizzi virtuali dei buffer allocati nella coda. Il campo **sender_wairing** deve valere **true** se un processo sta aspettando che si liberino degli slot per poter trasmettere.

Modificare i file **io.s** e **io.cpp** in modo da realizzare la primitiva come descritto.

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

```
// ( SOLUZIONE 2018-09-19
extern "C" bool c_send(natl dst, const char *buf, natl len)
{
    if (!access(buf, len, false)) {
        flog(LOG_WARN, "parametri non validi");
        abort_p();
    }

    if (len > MAX_PAYLOAD * (DIM_RING - 1)) {
        flog(LOG_WARN, "len non valido: %d");
        abort_p();
    }

    sem_wait(net.mutex);

    natl frags = num_frag(len);
    natl tail = inputl(net.iTAIL);
    while (ring_avail(tail, net.old_head) < frags) {
        sem_signal(net.mutex);
        sem_wait(net.sync);
        sem_wait(net.mutex);
        tail = inputl(net.iTAIL);
    }

    natl newtail = tail;
    for (natl i = 0; i < frags; i++) {
        msg* m = new(aligned_val_t{64}) msg;
        if (!m)
            goto cleanup;

        m->src = myaddr;
        m->dst = dst;
        m->len = (len > MAX_PAYLOAD ? MAX_PAYLOAD : len);
        m->seq = i;
        memcpy(m->payload, buf, m->len);
        buf += m->len;
        len -= m->len;
        net.m[newtail] = m;

        slot *s = &net.s[newtail];
        s->addr = trasforma(m);
        s->len = m->len + 4 * sizeof(natl);
        newtail = ring_next(newtail);
    }
    outputl(newtail, net.iTAIL);

    sem_signal(net.mutex);

    return true;

cleanup:
    for (; tail != newtail; tail = ring_next(tail)) {
        delete net.m[tail];
        net.m[tail] = nullptr;
    }
    sem_signal(net.mutex);
```

```
        return false;
}

// SOLUZIONE 2018-09-19 )

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


// ( SOLUZIONE 2018-09-19
    .extern c_waitnet
a_waitnet:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_waitnet
    iretq
    .cfi_endproc

    .extern c_send
a_send:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_send
    iretq
    .cfi_endproc
// SOLUZIONE 2018-09-19 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

16 gennaio 2019

1. Collegiamo al sistema una periferica PCI di tipo **ce**, con vendorID **0xedce** e deviceID **0x1234**. Le periferiche **ce** sono schede di rete che operano in PCI Bus Mastering. Il software deve preparare dei buffer vuoti, che la scheda riempie autonomamente con messaggi ricevuti dalla rete.

Per permettere al software di operare in parallelo con la ricezione, la scheda usa una coda circolare di descrittori di buffer. Ogni descrittore deve contenere l'indirizzo fisico di un buffer, la sua lunghezza in byte e un flag di “fine messaggio”. La coda di buffer ha 8 posizioni, numerate da 0 a 7. Il flag è necessario perché la scheda può usare più di un buffer per un singolo messaggio, se questo è più lungo della dimensione del buffer.

La scheda possiede due registri, **HEAD**, di sola lettura, e **TAIL**, di lettura/scrittura. I due registri contengono numeri di posizioni all'interno della coda. La scheda può usare soltanto i descrittori che vanno da **HEAD** in avanti (circolarmente) senza toccare **TAIL**.

All'avvio il software prepara tutti i buffer e inizializza tutti i descrittori, quindi scrive 7 in **TAIL**. In questo modo la scheda può usare i descrittori da 0 a 6 (uno deve essere sempre non utilizzato, per distinguere gli stati di coda piena e coda vuota).

Ogni volta che la scheda ha terminato di ricevere un messaggio lo copia in uno o più buffer partendo da quello puntato dal descrittore indicato da **HEAD** e andando avanti (circolarmente) e incrementando ogni volta **HEAD**, eventualmente fermandosi se raggiunge **TAIL**. Oltre a copiare il messaggio, la scheda modifica i descrittori utilizzati per scrivervi il numero di byte scritti nel corrispondente buffer (sovrascrivendo il campo del descrittore che conteneva la lunghezza del buffer) e settando opportunamente il flag di “fine messaggio”. In un momento qualunque (anche prima di aver terminato un messaggio), la scheda può inviare una richiesta di interruzione per segnalare che **HEAD** è stato modificato (e dunque alcuni descrittori sono stati usati). La lettura di **HEAD** funge da risposta alla richiesta. I descrittori utilizzati saranno quelli che si trovano tra l'ultima posizione letta da **HEAD** (inclusa) e la nuova (esclusa).

Ogni periferica **ce** usa 16 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione **BAR0**, sia *b*. I registri accessibili al programmatore sono i seguenti:

1. **HEAD** (indirizzo *b*, 4 byte): posizione di testa;
2. **TAIL** (indirizzo *b* + 4, 4 byte): posizione di coda;
3. **RING** (indirizzo *b* + 8, 4 byte): indirizzo fisico del primo descrittore della coda circolare;

Supponiamo che ogni computer collegato alla rete possieda un indirizzo numerico di 4 byte. Ogni computer possiede un'unica periferica di tipo **ce**. I messaggi che viaggiano sulla rete contengono una “intestazione” con due campi (ciascuno grande 4 byte): l'indirizzo del computer che invia e l'indirizzo **dst** del computer a cui il messaggio è destinato. L'intestazione è seguita dal messaggio vero e proprio.

Vogliamo fornire all'utente una primitiva

```
bool receive(nat1& src, char *buf, natq& len)
```

che permetta di ricevere un messaggio nel buffer `buf`. Il parametro `len` contiene inizialmente la dimensione del buffer e, dopo la ricezione, contiene la dimensione effettiva del messaggio ricevuto. Il parametro `src` conterrà l'indirizzo del computer che ha inviato il messaggio. Attenzione: l'utente deve ricevere in `buf` solo il messaggio vero e proprio, esclusa l'intestazione. Si noti che ogni invocazione della primitiva restituisce un solo messaggio: eventuali altri messaggi in coda verranno restituiti alla prossime invocazioni. Se, invece, non vi sono messaggi in coda, la primitiva blocca il processo in attesa che ne arrivi almeno uno. La primitiva restituisce `true` se il buffer `msg` è sufficiente a contenere il prossimo messaggio da ricevere; altrimenti copia in `buf` solo la parte di messaggio che vi entra e restituisce `false`.

Per descrivere le periferiche `ce` aggiungiamo le seguenti strutture dati al modulo I/O:

```
struct slot {
    natl addr;
    natw len;
    natw eop;
};

const natl DIM_RING = 8;
struct des_ce {
    natw iHEAD, iTAIL, iRING;
    slot s[DIM_RING];
    natl mutex;
    natl slots_ready;
    natl old_head;
    natl toread;
} net;
```

La struttura `slot` rappresenta un descrittore di buffer (indirizzo fisico in `addr` e lunghezza in `len`). Dopo che la scheda ha usato un descrittore, `len` contiene il numero di byte scritti nel buffer e `eop` è diverso da zero se il messaggio è terminato con questo descrittore. La struttura `des_ce` descrive una periferica di tipo `ce` e contiene al suo interno: gli indirizzi dei registri `HEAD`, `TAIL` e `RING`; la coda circolare di descrittori, `s`; l'indice di un semaforo di mutua esclusione (`mutex`); l'indice di un semaforo `slots_ready`, inizializzato a zero; il campo `old_head`, utile a memorizzare l'ultimo valore letto dal registro `HEAD`; il campo `toread`, utile a memorizzare l'indice del prossimo buffer da leggere tramite la `receive`.

Si può assumere che la scheda riceva solo messaggi effettivamente destinati al computer a cui è collegata.

Modificare i file `io.s` e `io.cpp` in modo da realizzare la primitiva come descritto.

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

```
// ( SOLUZIONE 2019-01-16
extern "C" bool c_receive(natl& src, char *buf, natq& len)
{
    if (!access(&src, sizeof(src), true) ||
        !access(&len, sizeof(len), true) ||
        !access(buf, len, true))
    {
        flog(LOG_WARN, "parametri errati");
        abort_p();
    }

    // mutua escusione tra i processi che vogliono usare receive
    sem_wait(net.mutex);

    bool rv = true;    // valore di ritorno
    natq bufsz = len; // spazio rimasto nel buffer utente
    len = 0;           // lunghezza del messaggio, da scoprire
    slot *s;
    do {
        char *from; // da dove copiare
        natq howmany; // quanti byte copiare

        // aspettiamo che ci sia uno slot pronto
        sem_wait(net.slots_ready);
        s = &net.s[net.toread];
        if (!len) {
            // primo segmento del messaggio, contiene l'intestazione
            msg_hdr* m = to_msg_hdr(s->addr);
            src = m->src;
            // i byte da copiare si trovano subito dopo
            l'intestazione
            from = to_char_ptr(s->addr) + sizeof(msg_hdr);
            len = s->len - sizeof(msg_hdr);
            howmany = len;
        } else {
            // segmenti successivi: solo dati
            from = to_char_ptr(s->addr);
            howmany = s->len;
            len += s->len;
        }
        if (howmany > bufsz) {
            // spazio esaurito: tronchiamo il messaggio
            // e ci ricordiamo di restituire errore
            howmany = bufsz;
            rv = false;
        }
        memcpy(buf, from, howmany);
        buf += howmany;
        bufsz -= howmany;
        net.toread = ring_next(net.toread);

        // ripristiniamo lo slot corrente e diamo alla scheda il
        // permesso di usarne un altro
        s->len = RXBUFSZ;
        natl tail;
        tail = inputl(net.iTAIL);
```

```
    tail = ring_next(tail);
    outputl(tail, net.iTAIL);
} while (!s->eop);

sem_signal(net.mutex);

return rv;
}
// SOLUZIONE 2019-01-16 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

6 febbraio 2019

1. Collegiamo al sistema una periferica PCI di tipo **ce**, con vendorID **0xedce** e deviceID **0x1234**. Le periferiche **ce** sono schede di rete che operano in PCI Bus Mastering. Il software deve preparare dei buffer vuoti, che la scheda riempie autonomamente con messaggi ricevuti dalla rete.

Per permettere al software di operare in parallelo con la ricezione, la scheda usa una coda circolare di descrittori di buffer. Ogni descrittore deve contenere l'indirizzo fisico di un buffer, la sua lunghezza in byte e un flag di “fine messaggio”. La coda di buffer ha 8 posizioni, numerate da 0 a 7. Il flag è necessario perché la scheda può usare più di un buffer per un singolo messaggio, se questo è più lungo della dimensione del buffer.

La scheda possiede due registri, **HEAD**, di sola lettura, e **TAIL**, di lettura/scrittura. I due registri contengono numeri di posizioni all'interno della coda. La scheda può usare soltanto i descrittori che vanno da **HEAD** in avanti (circolarmente) senza toccare **TAIL**.

All'avvio il software prepara tutti i buffer e inizializza tutti i descrittori, quindi scrive 7 in **TAIL**. In questo modo la scheda può usare i descrittori da 0 a 6 (uno deve essere sempre non utilizzato, per distinguere gli stati di coda piena e coda vuota).

Ogni volta che la scheda ha terminato di ricevere un messaggio lo copia in uno o più buffer partendo da quello puntato dal descrittore indicato da **HEAD** e andando avanti (circolarmente) e incrementando ogni volta **HEAD**, eventualmente fermandosi se raggiunge **TAIL**. Oltre a copiare il messaggio, la scheda modifica i descrittori utilizzati per scrivervi il numero di byte scritti nel corrispondente buffer (sovrascrivendo il campo del descrittore che conteneva la lunghezza del buffer) e settando opportunamente il flag di “fine messaggio”. In un momento qualunque (anche prima di aver terminato un messaggio), la scheda può inviare una richiesta di interruzione per segnalare che **HEAD** è stato modificato (e dunque alcuni descrittori sono stati usati). La lettura di **HEAD** funge da risposta alla richiesta. I descrittori utilizzati saranno quelli che si trovano tra l'ultima posizione letta da **HEAD** (inclusa) e la nuova (esclusa).

Ogni periferica **ce** usa 16 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione BAR0, sia *b*. I registri accessibili al programmatore sono i seguenti:

1. **HEAD** (indirizzo *b*, 4 byte): posizione di testa;
2. **TAIL** (indirizzo *b* + 4, 4 byte): posizione di coda;
3. **RING** (indirizzo *b* + 8, 4 byte): indirizzo fisico del primo descrittore della coda circolare;

Vogliamo fornire all'utente due primitive

```
natq waitnet();  
void receive(char *buf, natq len);
```

La prima primitiva attende l'arrivo di un messaggio completo e ne restituisce la lunghezza. La seconda copia nel buffer **buf** i primi **len** byte dell'ultimo messaggio ricevuto (se il messaggio è più lungo, i rimanenti byte vengono scartati). Se si chiama più volte **waitnet** senza chiamare **receive**, la **waitnet** continua a restituire la lunghezza dell'ultimo messaggio senza attenderne uno nuovo. Se si chiama più

volte `receive` senza aver chiamato `waitnet`, il comportamento è indefinito (si può assumere che gli utenti non facciano quest'ultima cosa). I messaggi sono sempre lunghi almeno 8 byte.

Per descrivere le periferiche `ce` aggiungiamo le seguenti strutture dati al modulo I/O:

```
struct slot {
    natl addr;
    natw len;
    natw eop;
};

const natl DIM_RING = 8;
struct des_ce {
    natw iHEAD, iTAIL, iRING;
    slot s[DIM_RING];
    natl mutex;
    natl slots_ready;
    natl old_head;
    natl toread;
    natl last_idx;
    natl last_len;
} net;
```

La struttura `slot` rappresenta un descrittore di buffer (indirizzo fisico in `addr` e lunghezza in `len`). Dopo che la scheda ha usato un descrittore, `len` contiene il numero di byte scritti nel buffer e `eop` è diverso da zero se il messaggio è terminato con questo descrittore. La struttura `des_ce` descrive una periferica di tipo `ce` e contiene al suo interno: gli indirizzi dei registri `HEAD`, `TAIL` e `RING`; la coda circolare di descrittori, `s`; l'indice di un semaforo di mutua esclusione (`mutex`); l'indice di un semaforo `slots_ready`, inizializzato a zero; il campo `old_head`, utile a memorizzare l'ultimo valore letto dal registro `HEAD`; il campo `toread`, utile a memorizzare l'indice del prossimo slot da leggere; i campi `last_idx` e `last_len` che contengono, rispettivamente, l'indice del primo slot e la lunghezza complessiva (in byte) dell'ultimo messaggio ricevuto ma non ancora copiato (se `last_len` vale zero non ci sono messaggi da copiare).

Modificare i file `io.s` e `io.cpp` in modo da realizzare la primitiva come descritto. Controllare eventuali problemi di Cavallo di Troia.

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

```
// ( SOLUZIONE 2019-02-06

extern "C" natq c_waitnet()
{
    slot *s;
    natq len;

    // mutua escusione tra i processi che vogliono usare net
    sem_wait(net.mutex);

    if (!net.last_len) {
        do {
            // aspettiamo che ci sia uno slot pronto
            sem_wait(net.slots_ready);
            if (!net.last_len)
                net.last_idx = net.toread;
            s = &net.s[net.toread];
            net.last_len += s->len;
            net.toread = ring_next(net.toread);
        } while (!s->eop);
    }
    len = net.last_len;

    sem_signal(net.mutex);

    return len;
}

extern "C" void c_receive(char *buf, natq len)
{
    if (!access(buf, len, true)) {
        flog(LOG_WARN, "parametri non validi");
        abort_p();
    }

    // mutua escusione tra i processi che vogliono usare net
    sem_wait(net.mutex);

    slot *s;
    natl idx = net.last_idx;
    natl n = 0; // quanti slot stiamo liberando
    do {
        s = &net.s[idx];

        natq howmany = s->len;
        if (howmany > len) {
            // spazio esaurito: tronchiamo il messaggio
            howmany = len;
        }
        memcpy(buf, to_char_ptr(s->addr), howmany);
        buf += howmany;
        len -= howmany;

        // ripristiniamo lo slot corrente
        s->len = RXBUFSZ;
    }
}
```

```

        n++;
        idx = ring_next(idx);
    } while (!s->eop);
// diamo alla scheda il permesso di usare n slot in piu'
natl tail = inputl(net.iTAIL);
tail += n;
if (tail >= DIM_RING)
    tail -= DIM_RING;
outputl(tail, net.iTAIL);

/* ora si può ricevere un nuovo messaggio */
net.last_len = 0;

sem_signal(net.mutex);
}
// SOLUZIONE 2019-02-06 )

*****
* io/io.s
*****
// ( SOLUZIONE 2019-02-06
    .extern c_waitnet
a_waitnet:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_waitnet
    iretq
    .cfi_endproc

    .extern c_receive
a_receive:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_receive
    iretq
    .cfi_endproc
// SOLUZIONE 2019-02-06 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

22 febbraio 2019

- Collegiamo al sistema una periferica PCI di tipo **ce**, con vendorID **0xedce** e deviceID **0x1234**. Le periferiche **ce** sono schede di rete che operano in PCI Bus Mastering. Il software deve preparare dei buffer vuoti, che la scheda riempie autonomamente con messaggi ricevuti dalla rete.

Per permettere al software di operare in parallelo con la ricezione, la scheda usa una coda circolare di descrittori di buffer. Ogni descrittore deve contenere l'indirizzo fisico di un buffer e la sua lunghezza in byte. La coda di buffer ha 4 posizioni, numerate da 0 a 3.

La scheda possiede due registri, **HEAD**, di sola lettura, e **TAIL**, di lettura/scrittura. I due registri contengono numeri di posizioni all'interno della coda. La scheda può usare soltanto i descrittori che vanno da **HEAD** in avanti (circolarmente) senza toccare **TAIL**.

All'avvio il software prepara tutti i buffer e inizializza tutti i descrittori, quindi scrive 3 in **TAIL**. In questo modo la scheda può usare i descrittori da 0 a 2 (uno deve essere sempre non utilizzato, per distinguere gli stati di coda piena e coda vuota).

Ogni volta che la scheda ha terminato di ricevere un messaggio lo copia in un buffer partendo da quello puntato dal descrittore indicato da **HEAD**, andando avanti (circolarmente) e incrementando ogni volta **HEAD**, eventualmente fermandosi se raggiunge **TAIL**. Oltre a copiare il messaggio la scheda modifica i descrittori utilizzati per scrivervi il numero di byte scritti nel corrispondente buffer (sovrascrivendo il campo del descrittore che conteneva la lunghezza del buffer). In un momento qualunque la scheda può inviare una richiesta di interruzione per segnalare che **HEAD** è stato modificato e, dunque, alcuni descrittori sono stati usati. La lettura di **HEAD** funge da risposta alla richiesta. I descrittori utilizzati saranno quelli che si trovano tra l'ultima posizione letta da **HEAD** (inclusa) e la nuova (esclusa).

Ogni periferica **ce** usa 16 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione BAR0, sia *b*. I registri accessibili al programmatore sono i seguenti:

- HEAD** (indirizzo *b*, 4 byte): posizione di testa;
- TAIL** (indirizzo *b* + 4, 4 byte): posizione di coda;
- RING** (indirizzo *b* + 8, 4 byte): indirizzo fisico del primo descrittore della coda circolare;

Vogliamo fornire all'utente due primitive

```
natq waitnet();
void receive(char *buf, natq len);
```

La prima primitiva (già realizzata) serve ad attendere l'arrivo del prossimo messaggio e a conoscerne la lunghezza, mentre la seconda (da realizzare) serve a copiarlo nel buffer **buf** di lunghezza **len** (se il messaggio è più lungo, i rimanenti byte vengono scartati). Dopo che un messaggio è stato copiato il corrispondente slot può essere reso nuovamente disponibile per la ricezione di nuovi messaggi.

Più processi possono invocare le due primitive in qualunque ordine. Supponiamo che un processo *P* invochi **waitnet()** e questa restituisca la lunghezza di un messaggio *m*. Quando *P* invocherà **receive()**,

questa copierà proprio m , anche se nel frattempo altri processi hanno invocato `waitnet()` e/o `receive()`. In particolare, se un processo $Q \neq P$ invoca `waitnet()` prima che P invochi `receive()`, Q riceverà la lunghezza del prossimo messaggio da copiare, sia p , anche se m non è stato ancora copiato. È dunque possibile che p venga copiato (da Q) prima che P copi m .

Fino a quando P non invoca `receive()`, ogni ulteriore invocazione di `waitnet()` da parte di P continuerà a restituire la lunghezza di m , senza attendere nuovi messaggi. È invece un errore invocare `receive()` senza aver prima chiamato `waitnet()`, e in quel caso la primitiva abortisce il processo.

Per descrivere le periferiche `ce` aggiungiamo le seguenti strutture dati al modulo I/O:

```
struct slot {
    natl addr;
    natl len;
};

const natl DIM_RING = 4;
struct des_ce {
    natw iHEAD, iTAIL, iRING;
    slot s[DIM_RING];
    natl procs[DIM_RING];
    natl mutex;
    natl slots_ready;
    natl tocopy;
    natl towait;

    natl old_head;
} net;
```

La struttura `slot` rappresenta un descrittore di buffer (indirizzo fisico in `addr` e lunghezza in `len`). Dopo che la scheda ha usato un descrittore, `len` contiene il numero di byte scritti nel buffer. La struttura `des_net` descrive una periferica di tipo `ce` e contiene al suo interno: gli indirizzi dei registri `HEAD`, `TAIL` e `RING`; la coda circolare di descrittori, `s`, una coda `procs` per ricordare quale processo deve copiare ogni messaggio (il valore zero può essere usato per indicare “nessun processo”); l’indice di un semaforo di mutua esclusione (`mutex`); l’indice di un semaforo `slots_ready`, inizializzato a zero; il campo `tocopy`, che memorizza l’indice del prossimo slot ancora da copiare; il campo `towait`, che memorizza l’indice del prossimo slot su cui è necessario attendere che la scheda riceva un messaggio; il campo `old_head`, utile a memorizzare l’ultimo valore letto dal registro `HEAD`;

Modificare i file `io.s` e `io.cpp` in modo da realizzare la primitive mancanti. Controllare eventuali problemi di Cavallo di Troia.

ATTENZIONE: In base a quanto detto, i messaggi possono essere copiati in un ordine diverso da quello in cui sono arrivati. Fare attenzione, quando si avanza `TAIL`, a non passare alla scheda degli slot che contengano messaggi ancora da copiare.

```

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


// ( SOLUZIONE 2019-02-22
extern "C" void c_receive(char *buf, natq len)
{
    if (!access(buf, len, true)) {
        flog(LOG_WARN, "parametri errati");
        abort_p();
    }

    natl cur = getpid();

    // mutua escusione tra i processi che vogliono usare net
    sem_wait(net.mutex);

    // cerchiamo il primo slot che appartiene a cur
    natl scan;
    for (scan = net.tocopy; scan != net.towait; scan = ring_next(scan))
    {
        if (net.procs[scan] == cur)
            break;
    }
    if (scan == net.towait) {
        // il processo ha invocato receive() in modo errato
        sem_signal(net.mutex);
        flog(LOG_WARN, "receive() senza messaggi");
        abort_p();
    }
    slot *s = &net.s[scan];
    if (s->len < len)
        len = s->len;
    memcpy(buf, to_char_ptr(s->addr), len);
    // liberiamo lo slot corrente
    net.procs[scan] = 0;
    s->len = RXBUFSZ;
    // vediamo quanti slot possono essere restituiti alla scheda
    natl n;
    for (n = 0; net.tocopy != net.towait && !net.procs[net.tocopy];
    net.tocopy = ring_next(net.tocopy), n++)
    ;
    if (n > 0) {
        natl tail = inputl(net.iTAIL);
        tail += n;
        if (tail >= DIM_RING)
            tail -= DIM_RING;
        outputl(tail, net.iTAIL);
    }

    sem_signal(net.mutex);
}
// SOLUZIONE 2019-02-22 )

*****
```

```
    fill_io_gate      IO_TIPO_RECEIVE  a_receive
//  SOLUZIONE 2019-02-22 )
// ( SOLUZIONE 2019-02-22
    .extern c_receive
a_receive:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_receive
    iretq
    .cfi_endproc
//  SOLUZIONE 2019-02-22 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

12 giugno 2019

1. Vogliamo fornire ai processi la possibilità 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 P_1 . Quando (e se) un altro processo P_2 arriva a quell'indirizzo, il processo P_1 deve essere risvegliato. Si noti che il processo P_2 non si blocca e deve proseguire la sua esecuzione indisturbato (salvo che potrebbe dover cedere il processore a P_1 per via della preemption).

Prevediamo le seguenti limitazioni del meccanismo:

1. per ogni chiamata di `breakpoint(rip)` viene intercettato solo il primo processo che passa da `rip`: altri processi che dovessero passarvi dopo il primo non vengono intercettati;
2. Un solo processo alla volta può chiamare `breakpoint()`; la primitiva restituisce un errore se un altro processo sta già aspettando un breakpoint.

Si noti che se un processo esegue `int3` senza che ciò sia richiesto da una primitiva `breakpoint()` attiva, il processo deve essere abortito.

Si modifichino dunque i file `sistema/sistema.s` e `sistema/sistema.cpp` per implementare la seguente primitiva:

- `natl breakpoint(vaddr rip)`: (tipo `0x59`): blocca il processo chiamante in attesa che un altro processo provi ad eseguire l'istruzione all'indirizzo `rip`; restituisce l'id del processo intercettato, o `0xFFFFFFFF` se un altro processo sta già aspettando un breakpoint (a qualunque indirizzo); abortisce il processo se `rip` non appartiene all'intervallo `[ini_utn_c, fin_utn_c]` (zona utente/condivisa).

```

*****
* 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(natq rip)
{
    if (b_info.waiting) {
        esecuzione->contesto[I_RAX] = 0xFFFFFFFF;
        return;
    }

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

    natb *bytes = reinterpret_cast<natb*>(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 = reinterpret_cast<natb*>(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 pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

3 luglio 2019

1. Vogliamo fornire ai processi la possibilità di bloccare l'esecuzione di tutti i processi che passano da una certa istruzione. Per far questo forniamo alcune primitive. Con la primitiva `bpadd(vaddr rip)` si installa un breakpoint (istruzione `int3`, codice operativo `0xCC`) all'indirizzo `rip`. Da quel momento in poi, tutti i processi che passano da `rip` si bloccano e vengono accodati opportunamente. Nel frattempo, usando la primitiva `bpwait()`, un processo può sospendersi in attesa che un qualche altro processo passi dal breakpoint. La primitiva può essere invocata più volte, per attendere tutti i processi che si suppone debbano passare dal breakpoint. Infine, con la primitiva `bpremove()`, si rimuove il breakpoint e si risvegliano tutti i processi che vi si erano bloccati. I processi così risvegliati devono proseguire la loro esecuzione come se non fossero mai stati intercettati.

Prevediamo la seguente limitazione: ad ogni istante, nel sistema ci può essere al massimo un breakpoint installato tramite da `bpadd()`.

Si noti che se un processo esegue `int3` senza che ciò sia richiesto da una primitiva `bpadd()` attiva, il processo deve essere abortito.

Aggiungiamo al nucleo la seguente struttura dati:

```
struct b_info {
    des_proc *waiting;
    des_proc *intercepted;
    des_proc *to_wakeup;
    vaddr rip;
    natb orig;
    bool busy;
} b_info;
```

dove: `waiting` è una coda di processi che hanno invocato `bpwait()` e sono in attesa che qualche processo passi dal breakpoint; `intercepted` è una coda di processi che sono bloccati sul breakpoint e il cui identificatore non è stato ancora restituito da una `bpwait()`; `to_wakeup` è una coda di processi bloccati sul breakpoint e i cui identificatori sono stati già restituiti tramite `bpwait()`; `rip` è l'indirizzo a cui è installato il breakpoint; `orig` è il byte originariamente contenuto all'indirizzo `rip`; `busy` vale `true` se c'è un breakpoint installato.

Si modifichino i file `sistema/sistema.s` e `sistema/sistema.cpp` per implementare le seguenti primitive (abortiscono il processo in caso di errore):

- `bool bpadd(vaddr rip)`: (già realizzata): se non c'è un altro breakpoint già installato, installa il breakpoint all'indirizzo `rip` e restituisce `true`, altrimenti restituisce `false`; è un errore se `rip` non appartiene all'intervallo `[ini_utn_c, fin_utn_c]` (zona utente/condivisa).
- `natl bpwait()`: (già realizzata): attende che un qualche processo passi dal breakpoint e ne restituisce l'identificatore; può essere invocata più volte per ottenere gli identificatori di tutti i processi intercettati; è un errore invocare questa primitiva se non ci sono breakpoint installati;

- `void bpremove()` (da realizzare): rimuove il breakpoint e risveglia tutti i processi che erano stati intercettati; è un errore invocare questa primitiva se non ci sono breakpoint installati.

```

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

// ( SOLUZIONE 2019-07-03
    carica_gateTIPO_BPR    a_bpremove LIV_UTENTE
//   SOLUZIONE 2019-07-03 )
// ( SOLUZIONE 2019-07-03

a_bpremove:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_bpremove
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2019-07-03 )
// ( SOLUZIONE 2019-07-03
    movq $3, %rdi
    movq $0, %rsi
    movq (%rsp), %rdx
    call c_breakpoint_exception
//   SOLUZIONE 2019-07-03 )

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


// ( SOLUZIONE 2019-07-03
extern "C" void c_bpremove()
{
    if (b_info.waiting || !b_info.busy) {
        flog(LOG_WARN, "bpremove() errata");
        c_abort_p();
        return;
    }

    natb *bytes = reinterpret_cast<natb*>(b_info.rip);
    *bytes = b_info.orig;
    des_proc *work;
    while (b_info.intercepted) {
        work = rimozione_lista(b_info.intercepted);
        inserimento_lista(b_info.to_wakeup, work);
    }
    inspronti();
    while (b_info.to_wakeup) {
        work = rimozione_lista(b_info.to_wakeup);
        natq rsp_v = work->contesto[I_RSP];
        natq *rsp = reinterpret_cast<natq*>(trasforma(work->cr3,
rsp_v));
        (*rsp)--;
        inserimento_lista(pronti, work);
    }
    b_info.busy = false;
    schedulatore();
}

```

```
extern "C" void c_breakpoint_exception(int tipo, natq errore, vaddr rip)
{
    if (!b_info.busy || rip != b_info.rip + 1) {
        gestore_eccezioni(tipo, errore, rip);
        return;
    }
    if (b_info.waiting) {
        des_proc *work = rimozione_lista(b_info.waiting);
        work->contesto[I_RAX] = esecuzione->id;
        inserimento_lista(b_info.to_wakeup, esecuzione);
        inserimento_lista(pronti, work);
    } else {
        inserimento_lista(b_info.intercepted, esecuzione);
    }
    schedulatore();
}
// SOLUZIONE 2019-07-03 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

24 luglio 2019

1. Vogliamo fornire ai processi la possibilità di bloccare l'esecuzione di un processo a scelta, quando questo passa da una certa istruzione. Per far questo forniamo alcune primitive. Con la primitiva `bpattach(natl id, vaddr rip)` un processo *master* installa un breakpoint (istruzione `int3`, codice operativo `0xCC`) all'indirizzo `rip` per il processo `id`, che diventa *slave*. Da quel momento in poi il processo slave si blocca se passa da `rip`. Nel frattempo, usando la primitiva `bpwait()`, il processo master può sospendersi in attesa che lo slave passi dal breakpoint. Infine, con la primitiva `bpdetach()`, il processo master rimuove il breakpoint e risveglia eventualmente lo slave, il quale prosegue la sua esecuzione come se non fosse mai stato intercettato.

Si noti che processi che non sono slave non devono essere intercettati. Inoltre, se un processo esegue `int3` senza che ciò sia stato richiesto da un master, il processo deve essere abortito.

Aggiungiamo i seguenti campi ai descrittori di processo:

```
des_proc *bp_master;
des_proc *bp_slave;
vaddr bp_addr;
natb bp_orig;
natl bp_slave_id;
bool bp_waiting;
```

dove: `bp_master` punta al processo master di questo processo (nullo se il processo non ha un master); `bp_slave` punta al processo slave di questo processo (nullo se il processo non ha uno slave); `bp_addr`, `bp_orig` e `bp_slave_id` sono significativi solo per il processi master e contengono, rispettivamente, l'indirizzo a cui è installato breakpoint; il byte originariamente contenuto a quell'indirizzo e l'id dello slave; `bp_waiting` vale `true` nel descrittore di uno slave se il master è bloccato, e viceversa.

Si modifichino i file `sistema/sistema.s` e `sistema/sistema.cpp` per implementare le seguenti primitive (abortiscono il processo in caso di errore):

- `bool bpattach(natl id, vaddr rip)`: (realizzata in parte): se il processo chiamente non è slave e il processo `id` esiste e non è già uno slave o un master, installa il breakpoint all'indirizzo `rip` e restituisce `true`, altrimenti restituisce `false`; è un errore se `rip` non appartiene all'intervallo `[ini_utn_c, fin_utn_c]` (zona utente/condivisa), il processo è già master cerca di diventare master di se stesso.
- `natl bpwait()`: (già realizzata): attende che il processo slave passi dal breakpoint; è un errore invocare questa primitiva se il processo non è master;
- `void bpdetach()` (da realizzare): disfa tutto ciò che ha fatto la `bpattach()` e risveglia eventualmente il processo slave; è un errore invocare questa primitiva se il processo non è master;

Attenzione: bisogna fare in modo che solo i processi slave vengano intercettati, ma la parte utente/condivisa è appunto condivisa tra tutti i processi.

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

// ( SOLUZIONE 2019-07-24
    carica_gateTIPO_BPD    a_bpdetach LIV_UTENTE
//   SOLUZIONE 2019-07-24 )
// ( SOLUZIONE 2019-07-24
a_bpdetach:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_bpdetach
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2019-07-24 )
// ( SOLUZIONE 2019-07-24
    movq $3, %rdi
    movq $0, %rsi
//   movq (%rsp), %rdx
    movq %rsp, %rdx
//   call gestore_eccezioni
    call c_breakpoint_exception
//   SOLUZIONE 2019-07-24 )
```

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

// ( SOLUZIONE 2019-07-24
    if (!duplica(dest, rip))
        return;

    natb *bytes = reinterpret_cast<natb*>(trasforma(dest->cr3, rip));
    esecuzione->bp_addr = rip;
    esecuzione->bp_orig = *bytes;
    *bytes = 0xCC;
//   SOLUZIONE 2019-07-24 )
// ( SOLUZIONE 2019-07-24
void deduplica(des_proc *p, vaddr v)
{
    tab_iter it(p->cr3, v);
    for (it.post(); it; it.next_post()) {
        tab_entry e = it.get_e();
        paddr dst = extr_IND_FISICO(e);
        rilascia_frame(dst);
    }
    int idx = i_tab(v, MAX_LIV);
    get_entry(p->cr3, idx) = get_entry(esecuzione->cr3, idx);
}

extern "C" void c_bpdetach()
{
    des_proc *dest;

    if (!esecuzione->bp_slave) {
```

```
    flog(LOG_WARN, "bpremove() errata");
    c_abort_p();
    return;
}

dest = esecuzione->bp_slave;

deduplica(dest, esecuzione->bp_addr);

if (esecuzione->bp_waiting) {
    inspronti();
    inserimento_lista(pronti, dest);
    esecuzione->bp_waiting = false;
    schedulatore();
}

esecuzione->bp_slave = nullptr;
esecuzione->bp_addr = 0;
esecuzione->bp_orig = 0;
dest->bp_master = nullptr;
}

// SOLUZIONE 2019-07-24 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

23 settembre 2019

1. Vogliamo fornire ai processi la possibilità di bloccare l'esecuzione di un processo a scelta, ogni volta che questo passa da una certa istruzione. Per far questo forniamo alcune primitive. Con la primitiva `bpattach(nat1 id, vaddr rip)` un processo *master* installa un breakpoint (istruzione `int3`, codice operativo `0xCC`) all'indirizzo `rip` nella memoria virtuale del processo `id`, che diventa *slave*. Da quel momento in poi il processo slave si blocca se passa da `rip`. Nel frattempo, usando la primitiva `bpwait()`, il processo master può sospendersi in attesa che lo slave passi dal breakpoint. A quel punto il processo master può invocare la primitiva `bpcontinue()` per permettere al processo slave di continuare la propria esecuzione come se non fosse mai stato intercettato. Se lo slave ripassa dal breakpoint viene intercettato nuovamente e il meccanismo di ripete. Infine, con la primitiva `bpdetach()`, il processo master rimuove il breakpoint e, se necessario, risveglia un'ultima volta lo slave.

Si noti che processi che non sono slave non devono essere intercettati. Inoltre, se un processo esegue `int3` senza che ciò sia stato richiesto da un master, il processo deve essere abortito.

Aggiungiamo i seguenti campi ai descrittori di processo:

```
des_proc *bp_master;
des_proc *bp_slave;
vaddr bp_addr;
nattb bp_orig;
nat1 bp_slave_id;
bool bp_waiting;
```

dove: `bp_master` punta al processo master di questo processo (nullo se il processo non ha un master); `bp_slave` punta al processo slave di questo processo (nullo se il processo non ha uno slave); `bp_addr` e `bp_orig` sono significativi solo per i processi slave e contengono, rispettivamente, l'indirizzo a cui è installato il breakpoint e il byte originariamente contenuto a quell'indirizzo; il campo `bp_slave_id` è significativo solo per il processo master e contiene l'id dello slave; `bp_waiting` vale `true` nel descrittore di uno slave se il master è bloccato, e viceversa.

Si modifichino i file `sistema/sistema.s` e `sistema/sistema.cpp` per implementare le seguenti primitive (abortiscono il processo in caso di errore):

- `bool bpattach(nat1 id, vaddr rip)`: (tipo 0x59, già realizzata): se il processo che invoca la primitiva non è uno slave e il processo `id` esiste e non è già uno slave o un master, installa il breakpoint all'indirizzo `rip` e restituisce `true`, altrimenti restituisce `false`; è un errore se `rip` non appartiene all'intervallo `[ini_utn_c, fin_utn_c]` (zona utente/condivisa) o se il processo è già master o cerca di diventare master di se stesso.
- `void bpwait()`: (tipo 0x5a, già realizzata): attende che il processo slave passi dal breakpoint; è un errore invocare questa primitiva se il processo non è master;
- `void bpcontinue()`: (tipo 0x5c, da realizzare): permette allo slave di proseguire l'esecuzione, facendo in modo che venga intercettato nuovamente se ripassa dal breakpoint; è un errore invocare questa primitiva se il processo non è master o se lo slave non è bloccato sul breakpoint;

- `void bpdetach()` (tipo 0x5b, già realizzata): disfa tutto ciò che ha fatto la `bpattach()` e risveglia eventualmente il processo slave; è un errore invocare questa primitiva se il processo non è master;

Suggerimento: per realizzare la `bpcontinue()` si deve temporaneamente rimuovere il breakpoint, quindi reinserirlo non appena lo slave ha eseguito una istruzione. Per intercettare questo evento si può abilitare temporaneamente il single-step trap nel processo slave.

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

```
// ( SOLUZIONE 2019-09-23
    carica_gateTIPO_BPC    a_bpcontinue      LIV_UTENTE
//   SOLUZIONE 2019-09-23 )
// ( SOLUZIONE 2019-09-23

a_bpcontinue:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_bpcontinue
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2019-09-23 )
// ( SOLUZIONE 2019-09-23
    movq $3, %rdi
    movq $0, %rsi
    movq %rsp, %rdx
    call c_breakpoint_exception
//   SOLUZIONE 2019-09-23 )

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

```
// ( SOLUZIONE 2019-09-23
extern "C" void c_bpcontinue()
{
    des_proc *dest;

    // e' un errore se il processo non e' master;
    // i master hanno un campo bp_slave non-nullo
    if (!esecuzione->bp_slave) {
        flog(LOG_WARN, "bpcontinue() errata: chiamante non master");
        c_abort_p();
        return;
    }

    dest = esecuzione->bp_slave;

    // e' un errore anche se lo slave non e' bloccato sul breakpoint;
    // se e' bloccato deve essere nella lista bp_waiting del master
    // (si veda c_breakpoint_exception piu' sopra)
    if (!esecuzione->bp_waiting) {
        flog(LOG_WARN, "bpcontinue() errata: slave non bloccato");
        c_abort_p();
        return;
    }

    // settiamo il trap flag nella pila dello slave
    // vpila: indirizzo virtuale nella memoria dello slave
    vaddr vpila = dest->contesto[I_RSP];
```

```

    // se usassimo vpila scriveremmo nella pila del master invece che
in
    // quella dello slave, dal momento che la pila sistema usa una
    // traduzione diversa in ogni processo. Traduciamo allora vpila in
    // indirizzo fisico, in modo da poter accedere alla pila dello
slave
    // tramite la finestra FM
    paddr fpila = trasforma(dest->cr3, vpila);
    // (conversione in puntatore a natq, per convenienza)
    natq *pila = reinterpret_cast<natq *>(fpila);
    // ora scriviamo nella pila dello slave. In cima (pila[0])
    // c'è rip, segue il cpl (pila[1]) e poi i flag (pila[2])
    enable_single_step(pila[2]);

    // ripristiniamo il byte sovrascritto dal breakpoint. Per
    // esecuzione->bp_addrv vale un discorso analogo a vpila: non
possiamo usare
    // l'indirizzo direttamente, in quanto verrebbe tradotto nella
memoria
    // virtuale del master e non dello slave.
    paddr bp_paddr = trasforma(dest->cr3, dest->bp_addr);
    // (conversione a puntatore a natb, per convenienza)
    natb *bytes = reinterpret_cast<natb *>(bp_paddr);
    // accesso tramite la finestra FM
    *bytes = dest->bp_orig;
    // scriviamo 0xcc in bp_orig, per ricordarci che abbiamo fatto
    // continuare lo slave e che andra' ripristinato il breakpoint
dest->bp_orig = 0xCC;
    // (il rip salvato nella pila dello slave era gia' stato riportato
    // indietro di 1 nella c_breakpoint_exception)

    // risvegliamo lo slave
    inspronti();
    inserimento_lista(pronti, esecuzione->bp_waiting);
    esecuzione->bp_waiting = nullptr;
    schedulatore();
}

extern "C" void c_debug_exception(int tipo, natq errore, vaddr *p_rip)
{
    // Ripristiniamo il breakpoint
    // La traduzione attiva e' quella dello slave, quindi possiamo
    // usare esecuzione->bp_addr direttamente
    if (esecuzione->bp_orig == 0xCC) {
        natb *bytes = reinterpret_cast<natb*>(esecuzione->bp_addr);
        esecuzione->bp_orig = *bytes;
        *bytes = 0xCC;
    }

    // Resettiamo il TRAP_FLAG
    // (sfruttiamo il puntatore al rip, che punta in cima alla pila)
    natq* pila = reinterpret_cast<natq*>(p_rip);
    // anche qui possiamo usare direttamente l'indirizzo virtuale
    disable_single_step(pila[2]);
}
// SOLUZIONE 2019-09-23 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

16 gennaio 2020

1. Vogliamo fornire ai processi la possibilità di bloccarsi in attesa che un altro processo riceva una eccezione o termini. Un processo P deve prima registrarsi, tramite la primitiva `proc_attach(nat1 id)`, con il processo di identificatore `id`, chiamiamolo Q , di cui vuole controllare la terminazione. Diremo che P è il *master* di Q e che Q è lo *slave* di P . Successivamente il processo P può invocare la primitiva `proc_wait()` per bloccarsi in attesa che il processo Q termini (invocando `terminate_p()`) o riceva una eccezione. La primitiva `proc_wait()` restituisce al processo P il numero dell'eccezione ricevuta da Q , o il valore 32 in caso di terminazione normale. Si noti che la gestione dell'eccezione da parte del processo Q non cambia anche con questo nuovo meccanismo (quindi il processo Q deve essere comunque abortito, si veda `gestore_eccezioni()` in `sistema/sistema.cpp`).

Si modifichino i file `sistema/sistema.s` e `sistema/sistema.cpp` per implementare le seguenti primitive (abortiscono il processo in caso di errore):

- `bool proc_attach(nat1 id)`: (tipo 0x59, da realizzare) La primitiva restituisce `false` se il processo che la invoca è uno slave, oppure se il processo `id` non esiste oppure è già un master o uno slave. È un errore se il processo P è già master o cerca di diventare master di se stesso. Altrimenti fa in modo che P diventi il master di `id` e restituisce `true`.
- `nat1 proc_wait()`: (tipo 0x5a, da realizzare): attende che il processo slave termini, normalmente o per la ricezione di una eccezione (nota: si trascurino i page faults, tipo 14, e le interruzioni non mascherabili, tipo 2) e restituisce il numero dell'eccezione, o 32 nel caso di terminazione normale. È un errore invocare questa primitiva se il processo non è master;

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

```
// ( SOLUZIONE 2020-01-16
    des_proc *proc_master;
    des_proc *proc_slave;
    natl proc_exception;
    bool proc_waiting;
// SOLUZIONE 2020-01-16 )
// ( SOLUZIONE 2020-01-16
    des_proc* master = esecuzione->proc_master;

    if (master) {
        // il processo è un slave

        // scriviamo il tipo nel descrittore di processo del master.
        master->proc_exception = tipo;

        if (esecuzione->proc_waiting) {
            // il master è bloccato nella proc_wait().
            // Dobbiamo risvegliarlo e fare in modo che la
            // proc_wait() gli restituisca il tipo.
            master->contesto[I_RAX] = master->proc_exception;
            inserimento_lista(pronti, master);
            esecuzione->proc_waiting = false;
        }
    }
    // proseguiamo con la normale gestione dell'eccezione.
// SOLUZIONE 2020-01-16 )
// ( SOLUZIONE 2020-01-16
    p->proc_exception = 0xFFFFFFFF;
// SOLUZIONE 2020-01-16 )
// ( SOLUZIONE 2020-01-16
    des_proc *master = p->proc_master,
                *slave  = p->proc_slave;

    if (master) {
        // il processo p è uno slave (ha un master)
        if (master->proc_exception == 0xFFFFFFFF) {
            // siamo arrivati qui da una terminate_p() o una abort
            // e non da gestore_eccezioni()
            master->proc_exception = 32;
        }
        if (p->proc_waiting) {
            // il master è in attesa dentro proc_wait(), ma
            // lo slave sta terminando. Dobbiamo risvegliare
            // il master e fare in modo che la proc_wait()
            // restituisca il tipo di terminazione.
            master->contesto[I_RAX] = master->proc_exception;
            inserimento_lista(pronti, master);
            // (schedulatore() è già chiamata dal chiamante
            // di distruggi_processo())
            p->proc_waiting = false;
        }
        // lo slave sta terminando e il master non deve più
        // puntarla.
        master->proc_slave = nullptr;
    } else if (slave) {
        // il processo p è un master (ha uno slave).
```

```

        // Dobbiamo scollegare lo slave da p.
        slave->proc_master = nullptr;
    }
// SOLUZIONE 2020-01-16 )
// ( SOLUZIONE 2020-01-16

extern "C" void c_proc_wait()
{
    des_proc* slave = esecuzione->proc_slave;

    // è un errore se il processo non è master
    // (si noti che slave è nullptr anche quando lo slave è già
terminato,
    // ma in quel caso proc_exception sarà diverso dal valore iniziale
    // di 0xFFFFFFFF)
    if (!slave && esecuzione->proc_exception == 0xFFFFFFFF) {
        flog(LOG_WARN, "proc_wait() non chiamata da un master");
        c_abort_p();
        return;
    }

    // se lo slave non ha ancora aggiornato il campo
    // proc_exception blocchiamo il master
    if (esecuzione->proc_exception == 0xFFFFFFFF) {
        slave->proc_waiting = true;
        schedulatore();
    } else {
        // altrimenti restituiamo il tipo dell'eccezione
        esecuzione->contesto[I_RAX] = esecuzione->proc_exception;
    }
}

extern "C" void c_proc_attach(natl id)
{
    if (id >= MAX_PROC_ID) {
        flog(LOG_WARN, "id non valido: %d", id);
        c_abort_p();
        return;
    }

    des_proc* dest = des_p(id);

    // è un errore se il processo è già master
    if (esecuzione->proc_slave) {
        flog(LOG_WARN, "proc_attach() duplicata");
        c_abort_p();
        return;
    }

    // è un errore se un processo tenta di diventare
    // master di se stesso
    if (dest == esecuzione) {
        flog(LOG_WARN, "proc_attach() ricorsiva");
        c_abort_p();
        return;
    }

    // la primitiva fallisce se il processo esecuzione è uno slave,
    // oppure se il processo id non esiste o è già slave o master
}

```

```
    if (esecuzione->proc_master || !dest || dest->proc_master || dest-
>proc_slave) {
        esecuzione->contesto[I_RAX] = false;
        return;
    }

    // colleghiamo master e slave
    esecuzione->proc_slave = dest;
    dest->proc_master = esecuzione;

    esecuzione->contesto[I_RAX] = true;
}
// SOLUZIONE 2020-01-16 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

7 febbraio 2020

1. Vogliamo fornire ai processi la possibilità di bloccarsi in attesa che un altro processo riceva una eccezione o termini. Un processo *P* deve prima registrarsi, tramite la primitiva `proc_attach(nat1 id)`, con il processo di identificatore *id*, chiamiamolo *Q*, di cui vuole controllare la terminazione. Da questo momento in poi, se *Q* riceve una eccezione o invoca `terminate_p()`, deve essere messo in *pausa*. Diremo che *P* è il *master* di *Q* e che *Q* è lo *slave* di *P*. Un processo master può registrarsi con un numero qualunque di slave. Una volta registratosi, il processo master può invocare la primitiva `proc_wait()` per bloccarsi in attesa che almeno uno dei suoi slave vada in pausa (l'attesa può essere nulla se qualche slave era già andato in pausa nel frattempo). La primitiva `proc_wait()` restituisce al processo *P* il numero dell'eccezione ricevuta da uno dei suoi slave in pausa, o il valore 32 se lo slave aveva invocato `terminate_p()`. In caso di più slave in pausa, la primitiva restituisce il valore relativo allo slave con priorità maggiore. A questo punto lo slave in questione termina la pausa e completa la gestione dell'eccezione o della `terminate_p()` (in entrambi i casi viene di fatto distrutto). Una successiva invocazione della `proc_wait()` restituirà il valore relativo al successivo slave in pausa, in ordine di priorità, e così via fino all'esaurimento della coda.

Per realizzare questo meccanismo aggiungiamo i seguenti campi al descrittore di processo:

```
des_proc *slaves;
bool is_waiting;
des_proc *paused_slaves;

des_proc *master;
des_proc *next_slave;
nat1 last_exception;
```

I primi tre campi sono relativi ai master, con il seguente significato: `slaves` è una lista di tutti gli slave del master; `is_waiting` vale `true` se il master è in attesa nella `proc_wait()`; `paused_slaves` è una coda che contiene tutti gli slave attualmente in pausa. I secondi tre campi sono relativi agli slave, con il seguente significato: `master` punta al master dello slave; `next_slave` è usato per creare la lista di tutti gli slave dello stesso master (lista la cui testa è il puntatore `slaves` nel master); `last_exception` contiene il numero dell'ultima eccezione ricevuta dallo slave (o 32 se lo slave aveva invocato `terminate_p()`).

Si modifichino i file `sistema/sistema.s` e `sistema/sistema.cpp` per implementare il meccanismo e le seguenti primitive (abortiscono il processo in caso di errore):

- `bool proc_attach(nat1 id)`: (tipo 0x59, già realizzata) La primitiva restituisce `false` se il processo che la invoca è uno slave, oppure se il processo *id* non esiste oppure è già un master. È un errore se il processo *P* è già master o cerca di diventare master di se stesso. Altrimenti fa in modo che *P* diventi il master di *id* e restituisce `true`.
- `nat1 proc_wait()`: (tipo 0x5a, da realizzare): attende che un processo slave vada in pausa per la ricezione di una eccezione (nota: si trascurino i page faults, tipo 14, e le interruzioni non mascherabili, tipo 2) o termini normalmente e restituisce il numero dell'eccezione, o 32 nel caso di terminazione normale. È un errore invocare questa primitiva se il processo non è master;

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

```
// ( SOLUZIONE 2020-02-07
a_proc_attach:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_proc_attach
    call carica_stato
    iretq
    .cfi_endproc

a_proc_wait:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_proc_wait
    call carica_stato
    iretq
    .cfi_endproc
// SOLUZIONE 2020-02-07 )
```

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

```
// ( SOLUZIONE 2020-02-07
    // scolleghiamo tutti gli eventuali slave
    for (des_proc *slave = p->slaves; slave; slave = slave->next_slave)
    {
        slave->master = nullptr;
    }

    // distruggiamo tutti gli eventuali slave in pausa
    while (p->paused_slaves) {
        des_proc* slave = rimozione_lista(p->paused_slaves);
        distruggi_processo(slave);
        flog(sev, "Processo slave %d distrutto", p->id);
        processi--;
    }

    des_proc *master = p->master;
    if (master) {
        if (master->is_waiting) {
            master->contesto[I_RAX] = 32;
            inserimento_lista(pronti, master);
            master->is_waiting = false;
        } else {
            p->last_exception = 32;
            inserimento_lista(master->paused_slaves, p);
            p = nullptr; // non distruggiamo il processo
        }
    }
}
```

```

if (p) {
    distruggi_processo(p);
    flog(sev, "Processo %d %s", p->id, mode);
    processi--;
}
schedulatore();
// SOLUZIONE 2020-02-07 )
// ( SOLUZIONE 2020-02-07
extern "C" void c_proc_wait()
{
    // e' un errore se il processo non e' master;
    if (!esecuzione->slaves) {
        flog(LOG_WARN, "proc_wait() non chiamata da un master");
        c_abort_p();
        return;
    }

    if (esecuzione->paused_slaves) {
        des_proc *slave;

        slave = rimozione_lista(esecuzione->paused_slaves);
        esecuzione->contesto[I_RAX] = slave->last_exception;
        distruggi_processo(slave);
        flog(LOG_INFO, "Processo slave %d distrutto", slave->id);
        processi--;
    } else {
        esecuzione->is_waiting = true;
        schedulatore();
    }
}
// SOLUZIONE 2020-02-07 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

26 febbraio 2020

1. Vogliamo fornire ai processi la possibilità di bloccarsi in attesa che un altro processo riceva una eccezione, quindi decidere se tale processo deve proseguire nonostante l'eccezioni o essere distrutto. Un processo *P* deve prima registrarsi, tramite la primitiva `proc_attach(nat1 id)`, con il processo di identificatore *id*, chiamiamolo *Q*, di cui vuole controllare la ricezione delle eccezioni. Da questo momento in poi, se *Q* riceve una eccezione deve essere messo in *pausa*. Diremo che *P* è il *master* di *Q* e che *Q* è lo *slave* di *P*. Un processo master può registrarsi con un numero qualunque di slave. Una volta registratosi, il processo master può invocare la primitiva `proc_wait()` per bloccarsi in attesa che almeno uno dei suoi slave vada in pausa (l'attesa può essere nulla se qualche slave era già andato in pausa nel frattempo). La primitiva `proc_wait()` restituisce al processo *P* l'identificatore di uno dei suoi slave in pausa. In caso di più slave in pausa, la primitiva restituisce l'identificatore dello slave con priorità maggiore. A questo punto il master può terminare la pausa dello slave invocando la primitiva `proc_cont(nat1 id, bool terminate)`. Il parametro `terminate` permette di decidere se lo slave deve proseguire dal punto in cui aveva ricevuto l'eccezione, o essere distrutto. Nota: gli slave che terminano prima di ricevere una eccezione si scollano dal master; se il master è bloccato nella `proc_wait()` e tutti gli slave si scollano, la `proc_wait()` termina restituendo 0xFFFFFFFF; quando un master termina, tutti gli slave vengono scollati e quelli in pausa vengono distrutti.

Per realizzare questo meccanismo aggiungiamo i seguenti campi al descrittore di processo:

```
des_proc *slaves;
bool is_waiting;
des_proc *paused_slaves;

des_proc *master;
des_proc *next_slave;
nat1 last_exception;
```

I primi tre campi sono relativi ai master, con il seguente significato: `slaves` è una lista di tutti gli slave del master; `is_waiting` vale `true` se il master è in attesa nella `proc_wait()`; `paused_slaves` è una coda che contiene tutti gli slave attualmente in pausa. I secondi tre campi sono relativi agli slave, con il seguente significato: `master` punta al master dello slave; `next_slave` è usato per creare la lista di tutti gli slave dello stesso master (lista la cui testa è il puntatore `slaves` nel master); `last_exception` contiene il numero dell'ultima eccezione ricevuta dallo slave (o 32 se lo slave aveva invocato `terminate_p()`).

Si modifichino i file `sistema/sistema.s` e `sistema/sistema.cpp` per implementare il meccanismo e le seguenti primitive (abortiscono il processo in caso di errore):

- `bool proc_attach(nat1 id)`: (tipo 0x59, già realizzata) La primitiva restituisce `false` se il processo che la invoca è uno slave, oppure se il processo *id* non esiste oppure è già un master. È un errore se il processo *P* è già master o cerca di diventare master di se stesso. Altrimenti fa in modo che *P* diventi il master di *id* e restituisce `true`.

- **natl proc_wait()**: (tipo 0x5a, da realizzare): attende che almeno un processo slave vada in pausa per la ricezione di una eccezione (nota: si trascurino i page faulti, tipo 14, e le interruzioni non mascherabili, tipo 2) e restituisce l'identificatore dello slave in pausa a priorità maggiore; restituisce 0xFFFFFFFF se non ci sono slave (o se sono tutti terminati);
- **void proc_cont(nat1 id, bool terminate)**: (tipo 0x5c, da realizzare): termina la pausa del processo slave di identificatore **id**. Se **terminate** vale **true** il processo viene distrutto, altrimenti riparte dallo stato salvato alla ricezione dell'interruzione. È un errore invocare questa primitiva se il processo non è master, oppure se il processo di identificatore **id** non esiste, o non è uno slave in pausa del processo che invoca la primitiva.

```

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


// ( SOLUZIONE 2020-02-26
    des_proc *master = esecuzione->master;

    esecuzione->last_exception = tipo;
    if (master) {
        if (master->is_waiting) {
            // il master ha chiamato proc_wait() e si è bloccato,
            // quindi questo è il primo slave: risvegliamo il
            // master facendo in modo che proc_wait() gli
            // restituisca l'identificatore dello slave
            master->contesto[I_RAX] = esecuzione->id;
            inserimento_lista(pronti, master);
            master->is_waiting = false;
        }
        // lo slave deve essere messo in pausa in ogni caso.
        // Ne uscirà se il master chiama proc_cont() o termina.
        inserimento_lista(master->paused_slaves, esecuzione);
        schedulatore();
        // saltiamo la c_abort_p()
        return;
    }
    // il processo non è uno slave:
    // procediamo con la distruzione del processo
// SOLUZIONE 2020-02-26 )
// ( SOLUZIONE 2020-02-26
    // scolleghiamo tutti gli eventuali slave
    for (des_proc *slave = p->slaves; slave; slave = slave->next_slave)
{
    slave->master = nullptr;
}

// distruggiamo tutti gli eventuali slave in pausa
while (p->paused_slaves) {
    des_proc *slave = rimozione_lista(p->paused_slaves);
    distruggi_processo(slave);
    flog(sev, "Processo slave %d distrutto", p->id);
    processi--;
}

des_proc *master = p->master;
if (master) {
    sgancia_slave(p);
    // se non ci sono piu' slave e il master è bloccato nella
    proc_wait,
        // il master si deve risvegliare e la proc_wait deve
    restituirgli 0xFFFFFFFF
    if (master->is_waiting && !master->slaves) {
        master->is_waiting = false;
        master->contesto[I_RAX] = 0xFFFFFFFF;
        inserimento_lista(pronti, master);
    }
}

// SOLUZIONE 2020-02-26 )
// ( SOLUZIONE 2020-02-26
extern "C" void c_proc_wait()

```

```

{
    if (!esecuzione->slaves) {
        esecuzione->contesto[I_RAX] = 0xFFFFFFFF;
        return;
    }

    if (esecuzione->paused_slaves) {
        esecuzione->contesto[I_RAX] = esecuzione->paused_slaves->id;
    } else {
        esecuzione->is_waiting = true;
        schedulatore();
    }
}

extern "C" void c_proc_cont(natl id, bool terminate)
{
    if (id >= MAX_PROC_ID) {
        flog(LOG_WARN, "id non valido: %d", id);
        c_abort_p();
        return;
    }

    des_proc* slave = rimozione_lista_id(esecuzione->paused_slaves,
id);
    if (!slave) {
        flog(LOG_WARN, "proc %d non e' uno slave", id);
        c_abort_p();
        return;
    }
    if (terminate) {
        distruggi_processo(slave);
        processi--;
    } else {
        inspronti();
        inserimento_lista(pronti, slave);
        schedulatore();
    }
}
// SOLUZIONE 2020-02-26 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

18 giugno 2020

1. Vogliamo aggiungere al nucleo il meccanismo dei *gruppi di processi*. Un qualunque processo può attivare un nuovo gruppo e altri processi possono unirsi. Un processo può poi mettersi in attesa che uno qualunque dei processi di un gruppo termini.

I gruppi sono identificati da un numero tra 0 e MAX_GRP – 1. Un gruppo è *vuoto* se non vi appartiene nessun processo e *attivo* altrimenti. Al più MAX_GRP gruppi possono essere attivi in ogni istante.

Ogni processo può appartenere al più ad un gruppo alla volta e, alla creazione, non appartiene a nessun gruppo. Un processo entra a far parte di un gruppo o attivando un gruppo precedentemente vuoto (primitiva `newgrp()`, che restituisce l'identificatore del gruppo appena attivato) o aggiungendosi ad un gruppo già attivo (primitiva `joingroup(gid)`, dove `gid` deve essere l'identificatore del gruppo). Un processo abbandona il gruppo a cui appartiene se invoca la primitiva `leavegrp()`, se termina/abortisce, oppure se attiva un altro gruppo invocando `newgrp()` (non si può invece abbandonare un gruppo invocando direttamente `joingroup()` su un altro gruppo).

Un qualunque processo può invocare `waitgrp(gid)` e sospendersi in attesa della terminazione/aborto di uno qualunque dei processi che ancora fanno parte del gruppo `gid`. La primitiva restituisce l'identificatore del processo terminato o abortito. Se il gruppo `gid` è inizialmente vuoto, o si svuota in seguito, la primitiva restituisce invece 0xFFFFFFFF. Si noti che un gruppo può svuotarsi perché i processi che ne fanno parte possono abbandonarlo senza terminare o abortire. Più processi possono aver invocato `waitgrp(gid)` sullo stesso gruppo ed essere tutti contemporaneamente in attesa. Tutti i processi in questa condizione si risveglieranno insieme e riceveranno lo stesso risultato.

Per realizzare il meccanismo aggiungiamo le seguenti primitive (abortiscono il processo in caso di errore).

- `natl newgrp()`: attiva un gruppo precedentemente vuoto e ne restituisce l'identificatore. Il processo invocante entra nel nuovo gruppo e abbandona l'eventuale gruppo precedente. La primitiva restituisce 0xFFFFFFFF e non ha altri effetti se, e solo se, ci sono già MAX_GRP gruppi attivi.
- `bool joingroup(natl gid)`: tenta di entrare nel gruppo `gid`. La primitiva fallisce, e restituisce `false`, se il gruppo `gid` non è attivo, altrimenti restituisce `true`. È un errore invocare la primitiva se il processo appartiene già ad un gruppo (anche se si tratta dello stesso gruppo `gid`), o se `gid` non è un identificatore valido.
- `void leavegrp()`: il processo abbandona il gruppo corrente torna a non appartenere ad alcun gruppo. È un errore invocare la primitiva se il processo non appartiene ad un gruppo.
- `natl waitgrp(natl gid)`: sospende il processo in attesa che uno qualunque dei processi del gruppo `gid` termini/abortisca. Restituisce l'identificatore del processo terminato/abortito, oppure 0xFFFFFFFF se il gruppo è vuoto o si svuota prima che qualunque processo del gruppo termini/abortisca. È un errore se `gid` non è un identificatore valido.

Modificare i file `sistema.s` e `sistema.cpp` per realizzare il meccanismo. Gestire correttamente i casi di *preemption*.

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



// ( SOLUZIONE 2020-06-18
    carica_gateTIPO_NEWGRPa_newgrp    LIV_UTENTE
    carica_gateTIPO_LEAVEGRP        a_leavegrp  LIV_UTENTE
    carica_gateTIPO_JOINGRP       a_joingrp   LIV_UTENTE
    carica_gateTIPO_WAITGRP      a_waitgrp   LIV_UTENTE
//  SOLUZIONE 2020-06-18 )
// ( SOLUZIONE 2020-06-18
    .extern c_newgrp

a_newgrp:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_newgrp
    call carica_stato
    iretq
    .cfi_endproc

    .extern c_leavegrp
a_leavegrp:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_leavegrp
    call carica_stato
    iretq
    .cfi_endproc

    .extern c_joingrp
a_joingrp:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_joingrp
    call carica_stato
    iretq
    .cfi_endproc

    .extern a_waitgrp
a_waitgrp:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_waitgrp
    call carica_stato
    iretq
    .cfi_endproc
//  SOLUZIONE 2020-06-18 )
```

```

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

// ( SOLUZIONE 2020-06-18 #1
//   SOLUZIONE 2020-06-18 )
// ( SOLUZIONE 2020-06-18 #2
//   // gruppo a cui appartiene il processo, 0xFFFFFFFF se nessuno
//   natl grp_id;
//   SOLUZIONE 2020-06-18 )
// ( SOLUZIONE 2020-06-18 #3
//   // inizialmente il proceso non appartiene a nessun gruppo
//   p->grp_id = 0xFFFFFFFF;
//   SOLUZIONE 2020-06-18 )
// ( SOLUZIONE 2020-06-18 #4
// definita più avanti
void leavegrp(bool terminating);
// SOLUZIONE 2020-06-18 )
// ( SOLUZIONE 2020-06-18 #5
//   leavegrp(true /* terminating */);
//   // schedulatore() viene chiamato poco piu' avanti
//   SOLUZIONE 2020-06-18 )
// ( SOLUZIONE 2020-06-18 #6
struct grp_des {
    // numero di processi che appartengono al gruppo
    natl nproc;
    // processi in attesa che uno dei processi del gruppo termini
    des_proc *waiting;
};
grp_des array_grp[MAX_GRP];

// restuisce l'indice del primo gruppo vuoto, o 0xFFFFFFFF se sono tutti
pieni
natl first_empty_grp()
{
    natl gid;

    for (gid = 0; gid < MAX_GRP; gid++)
        if (!array_grp[gid].nproc)
            break;
    if (gid == MAX_GRP)
        return 0xFFFFFFFF;
    return gid;
}

// funzione che rimuove il processo in esecuzione del suo gruppo.
// Se 'terminating' è true la rimozione è dovuta alla terminazione o
aborto
// del processo, altrimenti all'invocazione di newgrp() o leavegrp().
// Se il processo non appartiene a nessun gruppo, la funzione non fa
niente.
void leavegrp(bool terminating)
{
    if (esecuzione->grp_id == 0xFFFFFFFF) {
        // nessun gruppo
        return;
    }

    grp_des *g = &array_grp[esecuzione->grp_id];

```

```

g->nproc--;
esecuzione->grp_id = 0xFFFFFFFF;
// sia se il processo sta terminando, sia se il gruppo si è
// svuotato, bisogna fare la stessa cosa: risvegliare tutti
// i processi in attesa sul gruppo. Cambia solo il valore
// da restituirci: esecuzione->id in caso di terminating e
// 0xFFFFFFFF in caso di gruppo svuotato.
if (terminating || !g->nproc) {
    while (g->waiting) {
        des_proc *p = rimozione_lista(g->waiting);
        p->contesto[I_RAX] = (terminating ? esecuzione->id :
0xFFFFFFFF);
        inserimento_lista(pronti, p);
    }
}
// il chiamante dovrà pensare a chiamare schedulatore()
}

extern "C" void c_newgrp()
{
    natl gid = first_empty_grp();

    // dobbiamo abbandonare l'eventuale gruppo corrente,
    // ma solo se l'allocazione ha avuto successo
    if (gid != 0xFFFFFFFF) {

        // leavegrp potrebbe spostare dei processi in coda pronti.
        // Per gestire la preemption inseriamo anche esecuzione
        // e poi chiamiamo schedulatore()
        inspronti();
        leavegrp(false /* !terminating */);
        schedulatore();

        // entriamo nel nuovo gruppo
        esecuzione->grp_id = gid;
        array_grp[gid].nproc++;
    }
    // in ogni caso restituiamo gid, anche se vale 0xFFFFFFFF
    esecuzione->contesto[I_RAX] = gid;
}

extern "C" void c_leavegrp()
{
    // errore se il processo non appartiene a un gruppo
    if (esecuzione->grp_id == 0xFFFFFFFF) {
        flog(LOG_WARN, "no group to leave");
        c_abort_p();
        return;
    }

    // come sopra
    inspronti();
    leavegrp(false /* !terminating */);
    schedulatore();
}

extern "C" void c_joingroup(natl gid)
{
    // errore se gid non è valido
    if (gid >= MAX_GRP) {

```

```

        flog(LOG_WARN, "invalid group id: %d", gid);
        c_abort_p();
        return;
    }
    // errore se il processo appartiene già ad un gruppo
    if (esecuzione->grp_id != 0xFFFFFFFF) {
        flog(LOG_WARN, "already in group: %d", esecuzione->grp_id);
        c_abort_p();
        return;
    }
    grp_des *g = &array_grp[gid];
    // La primitiva fallisce se il gruppo non è attivo
    if (!g->nproc) {
        esecuzione->contesto[I_RAX] = (natq)false;
        return;
    }
    // ok, entriamo nel gruppo
    esecuzione->grp_id = gid;
    g->nproc++;
    esecuzione->contesto[I_RAX] = (natq)true;
}

extern "C" void c_waitgrp(natl gid)
{
    // errore se gid non è valido
    if (gid >= MAX_GRP) {
        flog(LOG_WARN, "invalid group id: %d", gid);
        c_abort_p();
        return;
    }

    grp_des *g = &array_grp[gid];

    // se il gruppo è vuoto restituiamo subito 0xFFFFFFFF
    if (!g->nproc) {
        esecuzione->contesto[I_RAX] = 0xFFFFFFFF;
        return;
    }

    // altrimenti blocchiamo il processo
    inserimento_lista(g->waiting, esecuzione);
    schedulatore();
}
// SOLUZIONE 2020-06-18 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

8 luglio 2020

1. Colleghiamo al sistema una periferica PCI di tipo **ce**, con vendorID **0xedce** e deviceID **0x1234**. Le periferiche **ce** sono simili a piccoli hard disk e sono in grado di operare in PCI Bus Mastering autonomamente.

Ogni periferica contiene un certo numero di settori, ciascuno grande **DIM_SECT**, numerati a partire da 0, ed è in grado di trasferire un settore alla volta da o verso la memoria.

Mentre è in corso una operazione la periferica non è in grado di avviarne un'altra. Al termine dell'operazione in corso la periferica invia una richiesta di interruzione. La lettura del registro di stato fa da risposta alla richiesta di interruzione. Le interruzioni sono sempre abilitate, ma la periferica non invia ulteriori richieste finché non ha ricevuto risposta all'ultima inviata.

Ogni periferica **ce** usa 16 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione BAR0, sia *b*. I registri accessibili al programmatore sono i seguenti:

1. **SNR** (indirizzo *b*, 4 byte): Sector NumberR; il numero del settore che si vuole leggere o scrivere;
2. **CMD** (indirizzo *b* + 4, 4 byte): CoMmanD; scrivendo 1 in questo registro si avvia una operazione di scrittura (trasferimento dall'indirizzo ADR della memoria al settore SNR); scrivendo 2 si avvia una operazione di lettura (trasferimento dal settore SNR verso l'indirizzo ADR della memoria);
3. **STS** (indirizzo *b* + 8, 4 byte): STatuS, di sola lettura; nei 16 bit meno significativi contiene il numero di settori che la periferica implementa;
4. **ADR** (indirizzo *b* + 12, 4 byte): ADdRess; indirizzo di memoria a partire dal quale la periferica deve leggere (se CMD vale 1) o scrivere (se CMD vale 2);

Vogliamo permettere all'utente di leggere o scrivere dalle periferiche **ce** come se contenessero un'unica sequenza di byte, numerati a partire da 0, ignorando la suddivisione in settori. Per esempio, assumiamo che **DIM_SECT** sia 32 e che l'utente chieda di leggere 30 byte a partire dal byte numero 42: la primitiva di lettura restituirà gli ultimi 22 (cioè $32 - (42 \% 32)$) byte del settore numero 1 e i primi 8 (cioè $(42+30) \% 8$) del settore numero 2. Si noti che, per fare questo, la primitiva dovrà necessariamente trasferire i settori in un suo buffer interno, da cui poi estrarrà solo i byte richiesti dall'utente.

Tutte le periferiche di tipo **ce** presenti nel sistema sono identificate durante la fase di inizializzazione e vengono numerate a partire da 0.

Aggiungiamo dunque le seguenti primitive (abortiscono il processo in caso di errore):

- **bool ceread(nat1 id, nat1 off, char *buf, nat1 quanti)**: legge **quanti** byte dalla periferica **ce** numero **id** e li copia nel buffer puntato da **buf**; non è richiesto che tale buffer sia residente o condiviso; è un errore se la periferica **id** non esiste; la primitiva restituisce **false** se qualcuno dei byte [**off**, **off** + **quanti**] cade al di fuori di quelli effettivamente contenuti nella periferica;
- **bool cewrite(nat1 id, nat1 off, char *buf, nat1 quanti)**: scrive **quanti** byte nella periferica **ce** numero **id**, prendendoli dal buffer puntato da **buf**; non è richiesto che tale buffer sia residente o condiviso; *solo* i byte indicati devono essere modificati all'interno della periferica; è un errore se la periferica **id** non esiste; la primitiva restituisce **false** se qualcuno dei byte [**off**, **off** + **quanti**] cade al di fuori di quelli effettivamente contenuti nella periferica;

Modificare i file **io.s** e **io.cpp** in modo da realizzare la primitiva come descritto. Controllare eventuali problemi di Cavallo di Troia.

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


// ( SOLUZIONE 2020-07-08

struct des_ce {
    ioaddr iSNR, iCMD, iSTS, iADR;
    natl mutex;
    natl sync;
    // numero di settori contenuti nella periferica
    natl nsect;
    // Buffer di appoggio per letture/scritture.
    // Dobbiamo assicurarci che sia interamente contenuto
    // in un frame. Assumendo che DIM_SECT sia un sottomultiplo
    // di 4096, e' sufficiente allinearla a DIM_SECT
    char sector[DIM_SECT] __attribute__((aligned(DIM_SECT)));
};

des_ce array_ce[MAX_CE];
natl next_ce = 0;

extern "C" bool c_ceread(natl id, natl offset, char *buf, natl quanti)
{
    if (id >= next_ce) {
        flog(LOG_WARN, "dispositivo ce%d non esistente", id);
        abort_p();
    }

    if (!access(buf, quanti, true)) {
        flog(LOG_WARN, "parametri non validi");
        abort_p();
    }

    /* se quanti e' zero e' inutile procedere */
    if (!quanti)
        return true;

    des_ce *ce = &array_ce[id];

    // primo settore
    natl snum = offset / DIM_SECT;

    if (snum >= ce->nsect) {
        flog(LOG_INFO, "settore %d fuori dai limiti [0, %d]", snum,
ce->nsect);
        return false;
    }
    // convertiamo a natq per evitare overflow
    if ((natq)offset + quanti > (natq)ce->nsect * DIM_SECT) {
        flog(LOG_INFO, "offset + quanti = %d supera il limite %d",
(natq)offset + quanti, (natq)ce->nsect *
DIM_SECT);
        return false;
    }

    // offset del primo byte nel primo settore
    natl soff = offset % DIM_SECT;
    // quanti byte copiare dal primo settore
    natl tocopy = DIM_SECT - soff;
    if (quanti < tocopy) {
```

```

        tocopy = quanti;
    }

    // Visto che il buffer dell'utente puo' non essere residente, non
    // possiamo avviare il bus mastering direttamente verso di esso.
    // Faremo dunque sempre bus mastering verso il nostro buffer di
appoggio,
    // e da questo copieremo i byte nel buffer dell'utente.
    // Questo semplifica anche la gestione dei casi di lettura di
settori
    // incompleti.
    // Si noti che, trovandoci nel modulo I/O, non dobbiamo
preoccuparci di
    // eventuali page fault durante la copia, in quanto non abbiamo
vincoli
    // di atomicita'.
    //
    // Visto che il buffer dell'utente puo' anche essere privato, la
copia
    // non potra' essere effettuata dal processo esterno. Svolgeremo
dunque
    // tutte le operazioni all'interno della primitiva stessa (visto
che
    // questa gira sempre nel contesto del processo che la ha
invocata).
    // Il processo esterno si limitera' a segnalarc ci che il
trasferimento
    // e' completo.

    sem_wait(ce->mutex);
    // leggiamo un settore alla volta e copiamo i byte richiesti dentro
buf;
    // 'quanti' e' il numero di byte ancora da copiare
    // Nota: l'indirizzo fisico di 'ce->sector' e' stato scritto dentro
ADR
    // in fase di inizializzazione, dal momento che non cambia mai.
    while (quanti) {
        outputl(snum, ce->iSNR);
        outputl(2, ce->iCMD);
        sem_wait(ce->sync);
        memcpy(buf, ce->sector + soff, tocopy);
        snum++;
        buf += tocopy;
        quanti -= tocopy;

        // dal secondo ciclo in poi copieremo tutti i byte del
settore,
        // tranne eventualmente per l'ultimo
        soff = 0;
        tocopy = DIM_SECT;
        if (quanti < tocopy)
            tocopy = quanti;
    }
    sem_signal(ce->mutex);
    return true;
}

extern "C" bool c_cewrite(natl id, natl offset, char *buf, natl quanti)
{
    if (id >= next_ce) {

```

```

        flog(LOG_WARN, "dispositivo ce%d non esistente", id);
        abort_p();
    }

    if (!access(buf, quanti, false)) {
        flog(LOG_WARN, "parametri non validi");
        abort_p();
    }

    /* se quanti e' zero e' inutile procedere */
    if (!quanti)
        return true;

    des_ce *ce = &array_ce[id];

    natl snum = offset / DIM_SECT;

    if (snum >= ce->nsect) {
        flog(LOG_INFO, "settore %d fuori dai limiti [0, %d]", snum,
ce->nsect);
        return false;
    }
    if ((natq)offset + quanti > (natq)ce->nsect * DIM_SECT) {
        flog(LOG_INFO, "offset + quanti = %d supera il limite %d",
(natq)offset + quanti, (natq)ce->nsect *
DIM_SECT);
        return false;
    }

    natl soff = offset % DIM_SECT;
    natl tocopy = DIM_SECT - soff;
    if (quanti < tocopy) {
        tocopy = quanti;
    }
    sem_wait(ce->mutex);
    while (quanti) {
        outputl(snum, ce->iSNR);
        // nel caso di scrittura dobbiamo stare attenti a non
        // modificare byte non richiesti. Cio' potrebbe accadere nel
        // primo settore, se soff != 0, o nell'ultimo, se tocopy <
DIM_SECT.
        // In questi casi dobbiamo prima leggere il contenuto
        // attuale del settore
        if (soff || tocopy < DIM_SECT) {
            // SNR contiene gia' il valore corretto
            outputl(2, ce->iCMD);
            sem_wait(ce->sync);
        }
        // ora modifichiamo solo i byte richiesti
        memcpy(ce->sector + soff, buf, tocopy);
        // (ri)scriviamo il settore
        outputl(1, ce->iCMD);
        sem_wait(ce->sync);
        snum++;
        buf += tocopy;
        quanti -= tocopy;

        soff = 0;
        tocopy = DIM_SECT;
        if (quanti < tocopy)

```

```

        tocopy = quanti;
    }
    sem_signal(ce->mutex);
    return true;
}

extern "C" void estern_ce(int id)
{
    des_ce *ce = &array_ce[id];

    for (;;) {
        // risposta alla richiesta di interruzione
        inputl(ce->iSTS);
        sem_signal(ce->sync);
        wfi();
    }
}

bool ce_init()
{
    for (natb bus = 0, dev = 0, fun = 0;
         pci_find_dev(bus, dev, fun, 0xedce, 0x1234);
         pci_next(bus, dev, fun))
    {
        if (next_ce >= MAX_CE) {
            flog(LOG_WARN, "troppi dispositivi ce");
            break;
        }
        des_ce *ce = &array_ce[next_ce];
        natw base = pci_read_conf1(bus, dev, fun, 0x10);
        base &= ~0x1;
        ce->iSNR = base;
        ce->iCMD = base + 4;
        ce->iSTS = base + 8;
        ce->iADR = base + 12;
        ce->nsect = inputl(ce->iSTS);
        ce->nsect &= 0xFFFF; /* solo 16 bit sono significativi */
        // visto che faremo i trasferimenti sempre e solo verso
        // il buffer 'sector', possiamo scriverene l'indirizzo fisico
        // dentro ADR una volta per tutte
        outputl(trasforma(ce->sector), ce->iADR);

        ce->mutex = sem_ini(1);
        if (ce->mutex == 0xFFFFFFFF) {
            flog(LOG_ERR, "semafori insufficienti");
            return false;
        }
        ce->sync = sem_ini(0);
        if (ce->sync == 0xFFFFFFFF) {
            flog(LOG_ERR, "semafori insufficienti");
            return false;
        }
        natb irq = pci_read_conf1(bus, dev, fun, 0x3c);
        activate_pe(estern_ce, next_ce, MIN_EXT_PRIO + 0x70 + next_ce,
LIV, irq);
        flog(LOG_INFO, "ce%d: %2x:%1x:%1x base=%4x IRQ=%d size=%d
mutex=%d sync=%d",
                                         next_ce, bus, dev, fun, base, irq, ce->nsect, ce-
>mutex, ce->sync);
        next_ce++;
    }
}
```

```

        }
        return true;
    }
// SOLUZIONE 2020-07-08 )
// ( SOLUZIONE 2020-07-08
    if (!ce_init())
        panic("inizializzazione CE fallita");
// SOLUZIONE 2020-07-08 )

*****  

* io/io.s  

*****  

// ( SOLUZIONE 2020-07-08
    fill_io_gate    IO_TIPO_CEREAD    a_ceread
    fill_io_gate    IO_TIPO_CEWRITE   a_cewrite
// SOLUZIONE 2020-07-08 )
// ( SOLUZIONE 2020-07-08
    .extern          c_ceread
a_ceread:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_ceread
    iretq
    .cfi_endproc

    .extern          c_cewrite
a_cewrite:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_cewrite
    iretq
    .cfi_endproc
// SOLUZIONE 2020-07-08 )

```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

29 luglio 2020

1. Vogliamo permettere ad ogni processo di definire ciò che deve succedere quanto esso stesso causa una eccezione che, normalmente, ne causerebbe la terminazione forzata. Forniamo dunque una primitiva `signal(tipo, sighandler)`, tramite la quale un processo può associare un proprio *gestore sighandler* alle eccezioni di tipo `tipo`. Da quel momento in poi, quando il processo in questione causa una eccezione di questo tipo, il nucleo fa in modo che il processo stesso passi ad eseguire il gestore (ovviamente, al livello di privilegio del processo). Il gestore può poi terminare o abortire il processo (invocando le normali primitive già esistenti), oppure può decidere di farne proseguire l'esecuzione invocando la primitiva `signal_return(rip)`, che termina il gestore e fa proseguire il processo dall'indirizzo `rip`.

Ogni processo può associare un gestore, anche diverso, ad ogni eccezione (tranne la numero 2, associata agli interrupt non mascherabili). Le eccezioni che non sono associate a gestori si comportano come sempre, causando la terminazione forzata del processo che le ha causate. L'eccezione di tipo page-fault deve essere sempre gestita normalmente e passare il controllo al gestore solo nel caso in cui la normale gestione ha causato un errore.

I gestori hanno il seguente tipo:

```
typedef void (*sighandler)(int tipo, natq errore, natq rip)
```

I gestori devono dunque ricevere tre parametri: il `tipo` dell'eccezione (utile nel caso in cui lo stesso gestore sia stato associato a più eccezioni), l'`errore` che specifica ulteriormente la causa dell'eccezione, e il `rip` salvato in pila dal meccanismo delle eccezioni. Nel caso dell'eccezione di page-fault, `errore` contiene l'indirizzo virtuale che ha causato il fault.

I gestori non devono a loro volta causare eccezioni: se ciò accade, il processo deve essere abortito.

Aggiungiamo le seguenti primitive (abortiscono il processo in caso di errore):

- `void signal(int tipo, sighandler sigh)`: associa il signal handler `sigh` all'eccezione di tipo `tipo`, sostituendo un eventuale signal handler precedente. Se `sigh` vale zero viene ripristinato il funzionamento normale dell'eccezione. È un errore se `tipo` non corrisponde a nessun tipo di eccezione, oppure se si tenta di associare il gestore all'eccezione numero 2.
- `void signal_return(natq rip)`: termina il gestore e fa ripartire il processo dall'istruzione `rip`. I registri generali del processore devono avere il contenuto che avevano al momento dell'eccezione. È un errore invocare questa primitiva se non è in corso la gestione di una eccezione.

Modificare i file `sistema.s` e `sistema.cpp` in modo da realizzare il meccanismo descritto.

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

```
// ( SOLUZIONE 2020-07-29
    carica_gateTIPO_SIGNALa_signal    LIV_UTENTE
    carica_gateTIPO_SIGRETa_signal_return  LIV_UTENTE
//  SOLUZIONE 2020-07-29 )
// ( SOLUZIONE 2020-07-29
    .extern c_signal
a_signal:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_signal
    call carica_stato
    iretq
    .cfi_endproc

    .extern c_signal_return
a_signal_return:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_signal_return
    call carica_stato
    iretq
    .cfi_endproc
//  SOLUZIONE 2020-07-29 )
```

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

```
// ( SOLUZIONE 2020-07-29 #1
//  SOLUZIONE 2020-07-29 )
// ( SOLUZIONE 2020-07-29 #2
    /* puntatori ai vari signal handler */
    sighandler sigh[32];
    /* copia di contesto */
    natq csave[N_REG];
    /* true sse il processo ha ricevuto un segnale e non ha ancora
       * invocato signal_return */
    bool in_sighandler;
//  SOLUZIONE 2020-07-29 )
// ( SOLUZIONE 2020-07-29 #3
    if (esecuzione->sigh[tip] && !esecuzione->in_sighandler) {
        /* dobbiamo fare in modo che, al ritorno da questa funzione,
si
            * salti al signal handler. Per far questo e' sufficiente
            * modificare il rip che e' stato salvato in pila dal
            * meccanismo delle interruzioni e che sara' letto dalla iret
            * al ritorno da questa funzione. La salva_stato ha salvato
in
            * contesto[I_RSP] il puntatore alla pila subito dopo il
```

```

        * salvataggio delle 5 parole lunghe, e dunque contesto[I_RSP]
        * punta proprio al rip che vogliamo modificare.
        */
    natq *saved_rip = reinterpret_cast<natq*>(esecuzione-
>contesto[I_RSP]);
    /* si noti che contesto[I_RSP] contiene un indirizzo virtuale,
     * ma possiamo usarlo direttamente in quanto e' ancora attivo
     * il giusto albero di traduzione
     */
    *saved_rip = reinterpret_cast<natq>(esecuzione->sigh[tipos]);
    /* salviamo tutto il contesto, sia perche' dovremo modificarlo
     * per poter passare i parametri al signal handler, sia
perche'
     * il signal handler stesso potrebbe modificarlo se viene
     * interrotto.
     */
    for (int i = 0; i < N_REG; i++)
        esecuzione->csave[i] = esecuzione->contesto[i];
    /* passiamo i parametri al signal handler */
    esecuzione->contesto[I_RDI] = tipo;
    esecuzione->contesto[I_RSI] = (tipo == 14 ? readCR2() :
errore);
    esecuzione->contesto[I_RDX] = reinterpret_cast<natq>(rip);
    /* ci ricordiamo che il signal handler e' attivo, per
     * controllare che non vi siano annidamenti
     */
    esecuzione->in_sighandler = true;
    return;
}
// SOLUZIONE 2020-07-29 )
// ( SOLUZIONE 2020-07-29 #5
// SOLUZIONE 2020-07-29 )
// ( SOLUZIONE 2020-07-29 #6
extern "C" void c_signal(int t, sighandler h)
{
    if (t < 0 || t >= 32 || t == 2) {
        flog(LOG_WARN, "illegal type %d", t);
        c_abort_p();
        return;
    }
    esecuzione->sigh[t] = h;
}

extern "C" void c_signal_return(vaddr rip)
{
    /* controllo sull'annidamento delle eccezioni */
    if (!esecuzione->in_sighandler) {
        flog(LOG_WARN, "illegal call to signal_return");
        c_abort_p();
        return;
    }
    esecuzione->in_sighandler = false;

    /* ripristino del contesto e del rip */
    natq *saved_rip = reinterpret_cast<natq*>(esecuzione-
>contesto[I_RSP]);
    *saved_rip = rip;
    for (int i = 0; i < N_REG; i++)

```

```
    esecuzione->contesto[i] = esecuzione->csave[i];
    return;
}
// SOLUZIONE 2020-07-29 )
```

Prova pratica di Calcolatori Elettronici (nucleo v6.*)

C.d.L. in Ingegneria Informatica, Ordinamento DM 270

23 settembre 2020

- Vogliamo permettere ai processi di modificare il permesso di scrittura per alcuni indirizzi del proprio spazio di indirizzamento. Per farlo definiamo una nuova primitiva (tipo 0x5a):

```
bool mprotect(vaddr v, natq n, bool w);
```

La primitiva riceve un indirizzo virtuale di partenza, **v**, un numero di byte **n** e un flag **w**. La primitiva deve impostare i bit R/W nel percorso di traduzione di tutti gli indirizzi dell'intervallo [v, v+n) in modo che le scritture da parte del processo siano permesse (se **w** vale **true**) o vietate (se **w** vale **false**).

Per semplicità, la primitiva permette di eseguire questa operazione solo per intervalli che cadono interamente all'interno della parte utente condivisa, che va da **ini_utn_c**, incluso, a **fin_utn_c**, escluso. Se l'intervallo richiesto non è interamente contenuto tra questi indirizzi, il processo che ha invocato la primitiva deve essere abortito. Si ricordi che, per come abbiamo realizzato il sistema, questa parte della memoria virtuale di ogni processo contiene solo pagine residenti (e dunque anche le relative tabelle sono residenti). Inoltre, tutte le tabelle dal livello 3 a scendere sono condivise tra tutti i processi.

Non ci sono altri vincoli su **v** e **n**. In particolare, sia **v** che **v+n** potrebbero non essere allineati alla pagina. Resta però inteso che la primitiva aggiusterà i permessi del minimo insieme di pagine che contengono l'intervallo richiesto.

Attenzione: la primitiva deve cambiare i permessi solo per il processo che la invoca, anche se gli indirizzi si trovano nella parte condivisa tra tutti i processi. Questo vuol dire che la primitiva deve prima creare un percorso di traduzione privato per il processo (ma solo il percorso di traduzione deve diventare privato: le pagine vere e proprie devono restare condivise).

La primitiva potrebbe fallire perché deve duplicare delle tabelle e non ci sono più frame liberi. In tal caso restituisce **false**. Altrimenti restituisce **true**.

Poiché la memoria è limitata, è necessario non duplicare tabelle più del necessario. Inoltre, alla terminazione del processo, è anche necessario rendere di nuovo liberi i frame eventualmente usati per contenere le tabelle duplicate.

Modificare i file **sistema.s** e **sistema.cpp** in modo da realizzare la primitiva **mprotect()**, scrivendo esclusivamente negli spazi contrassegnati con **SOLUZIONE** (non è necessario però usarli tutti).

Il file **utente/prog/pmpmprotect.in** contiene i test, con dei commenti che spiegano cosa ci aspetta da ciascuno di essi.

Suggerimento 1: per duplicare un percorso di traduzione ci si può ispirare alla funzione **duplica()** dell'appello del 23 Settembre 2019.

Suggerimento 2: non preoccuparsi di ottimizzare troppo il tempo di esecuzione: è accettabile visitare il percorso di traduzione daccapo, anche più volte, se questo semplifica la soluzione. In particolare, nel caso in cui **mprotect()** debba lavorare su più pagine, è del tutto accettabile visitare il percorso di traduzione daccapo per ogni pagina, anche se questo comporta di visitare più volte le stesse tabelle di livello superiore.

```
*****
* util/question.tex
*****
```



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

```
// ( SOLUZIONE 2020-09-23 #1
    carica_gateTIPO_MPROTECT      a_mprotect LIV_UTENTE
//   SOLUZIONE 2020-09-23 )
// ( SOLUZIONE 2020-09-23 #2
a_mprotect:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_mprotect
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2020-09-23 )
```



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

```
// ( SOLUZIONE 2020-09-23 #1
//   SOLUZIONE 2020-09-23 )
// ( SOLUZIONE 2020-09-23 #3
    tab_iter it(root_tab, ini_utn_c, fin_utn_c - ini_utn_c);
    for (it.post(); it; it.next_post()) {
        tab_entry& e = it.get_e();

        if (e & BIT_P && !it.is_leaf()) {
            paddr tab = extr_IND_FISICO(e);
            if (vdf[tab/DIM_PAGINA].owner == p->id) {
                rilascia_frame(tab);
                e = 0;
            }
        }
    }
//   SOLUZIONE 2020-09-23 )
// ( SOLUZIONE 2020-09-23 #6
extern "C" void c_mprotect(vaddr v, natq n, bool wen)
{
    if (v < ini_utn_c || v + n < v || v + n >= fin_utn_c) {
        flog(LOG_WARN, "mprotect() su intervallo non consentito");
        c_abort_p();
        return;
    }

    for (tab_iter it(esecuzione->cr3, v, n); it; it.next()) {
        tab_entry &e = it.get_e();

        if (!(e & BIT_P))
```

```

        continue;

bool cwen = e & BIT_RW;

if (!it.is_leaf()) {
    paddr p = extr_IND_FISICO(e);
    des_frame *df = &vdf[p/DIM_PAGINA];

    if (df->owner != esecuzione->id) {
        // dobbiamo clonare la tabella
        paddr new_f = alloca_tab();
        if (!new_f)
            esecuzione->contesto[I_RAX] = false;
        return;
    }
    copy_des(p, new_f, 0, 512);
    set_IND_FISICO(e, new_f);
    df->owner = esecuzione->id;
}

if (wen && !cwen)
    e |= BIT_RW;
} else if (wen != cwen) {
    if (wen)
        e |= BIT_RW;
    else
        e &= ~BIT_RW;
    invalida_entrata_TLB(it.get_v());
}
esecuzione->contesto[I_RAX] = true;
}

// SOLUZIONE 2020-09-23 )

```