Architettura degli Elaboratori, 2007-08

Esercitazione 6 di verifica

Soluzione: entro giovedì 13 dicembre

Domanda 1

Si consideri il programma P2 dell'Esercitazione 5:

- P2:opera su tre array A[N], B[N], C[N][N] di interi, con $N = 4K$, e calcola la seguente funzione:

$$\forall i, j = 0 .. N-1 : C_{i,j} = min (A_i, B_j)$$

I valori degli array A, B sono *letti da un dispositivo DIN* ed il valore di C è *inviato ad un dispositivo DOUT*. Il linguaggio applicativo dispone di opportuni comandi per richiedere tali trasferimenti.

Oltre alle caratteristiche dell'architettura assembler-firmware, come nell'Esercitazione 5, sono note le caratteristiche del livello dei processi:

1. il linguaggio concorrente è *LC*;
2. il supporto alle primitive di scambio messaggi è realizzato sotto forma di procedure con parametri passati attraverso registri generali;
3. ogni unità di ingresso-uscita è interfacciata da un processo driver nei confronti delle applicazioni;
4. qualunque esito anomalo dei trasferimenti di ingresso-uscita viene segnalato ad un processo ECC, identificando il dispositivo e il processo che ha tentato il trasferimento.

a) Compilare P2 in un processo PROC2, mostrandone in dettaglio la memoria virtuale.

b) Compilare P2 in PROC2 nel caso che vengano eliminati i processi driver delle unità di I/O di DIN e DOUT, spiegando sotto quali condizioni la compilazione rimane inalterata rispetto al caso *a)*.

Domanda 2

a) Spiegare quali modifiche devono essere apportate al supporto di una *send* o di una *receive*, eseguita da un processo interno, nel caso che il partner sia un processo esterno. Spiegare se le modifiche dipendono dal fatto che il processo esterno sia eseguito su una unità di I/O operante in DMA e/o in Memory Mapped I/O.

b) Individuare una implementazione delle *send* e delle *receive* che rende la compilazione di un processo indipendente dal fatto che i processi con cui comunica siano interni o esterni.

c) Si consideri una unità di I/O in due versioni, una capace e l'altra non capace di eseguire primitive *send* e *receive*. Spiegare, nei due casi, quali funzionalità sono svolte dall'handler delle interruzioni inviate dall'unità.

Domanda 3

Si consideri un sistema che faccia uso di metodi *statici* con indirizzi logici per realizzare la condivisione mediante riferimenti indiretti.

a) Spiegare se le seguenti affermazioni sono vere, false, o vere sotto determinate condizioni:

1. istante per istante, deve essere noto il numero dei processi creati;
2. deve essere noto il massimo numero dei processi che possono creati;
3. i PCB dei processi creati devono essere allocati nelle stesse posizioni nelle memorie virtuali di qualunque processo;
4. l'allocazione della memoria principale ai PCB deve essere nota a tempo di compilazione;
5. l'allocazione della memoria principale ai PCB deve essere nota a tempo di creazione-caricamento.

b) Nei confronti dei punti precedenti, spiegare come si caratterizza un metodo *dinamico* per realizzare la condivisione mediante riferimenti indiretti.

Soluzione

Domanda 1

a) Lo pseudo codice del programma P2 è del tipo:

int A[N], B[N], C[N][N];

{ *leggi* (N, A);

leggi (N, B);

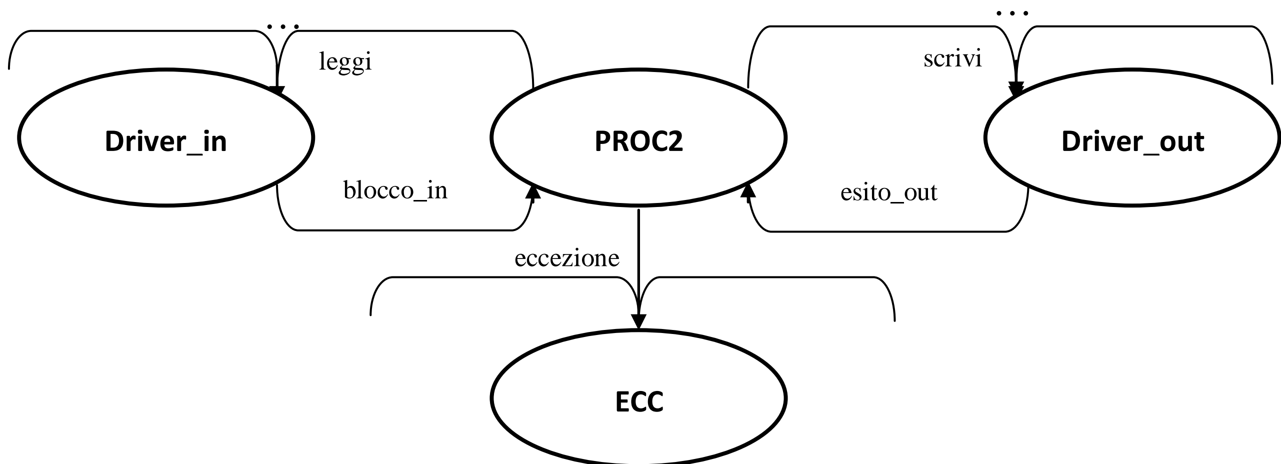
< calcolo >

scrivi (N∗N, C)

}

dove le funzioni leggi e scrivi sono disponibili a livello applicativo ed operano su dati di tipo intero.

P2 viene compilato in un processo PROC2 facente parte della seguente configurazione di processi:



Si suppone che i processi-servizi *Driver_in* e *Driver_out* virtualizzino, rispettivamente, le unità I/O-DIN e I/O-DOUT, e quindi i dispositivi DIN e DOUT, in modo da trasferire blocchi di *b* parole (ad esempio, *b* = 1K). L'esito dei trasferimenti è segnalato con un intero non negativo *tipoesito*, il cui valore zero indica che il trasferimento è avvenuto con successo. Vanno aggiunti i canali *from_gmem*, *to_gmem* con il processo *Gestore della Memoria*, per il trattamento dell'eccezione di fault di pagina.

PROC2 è compilato in LC nel seguente modo:

PROC2:: **int** A[N], B[N], C[N][N]; **const** b = …, Din = …, DOUT = …; **int** tipoesito, disp;

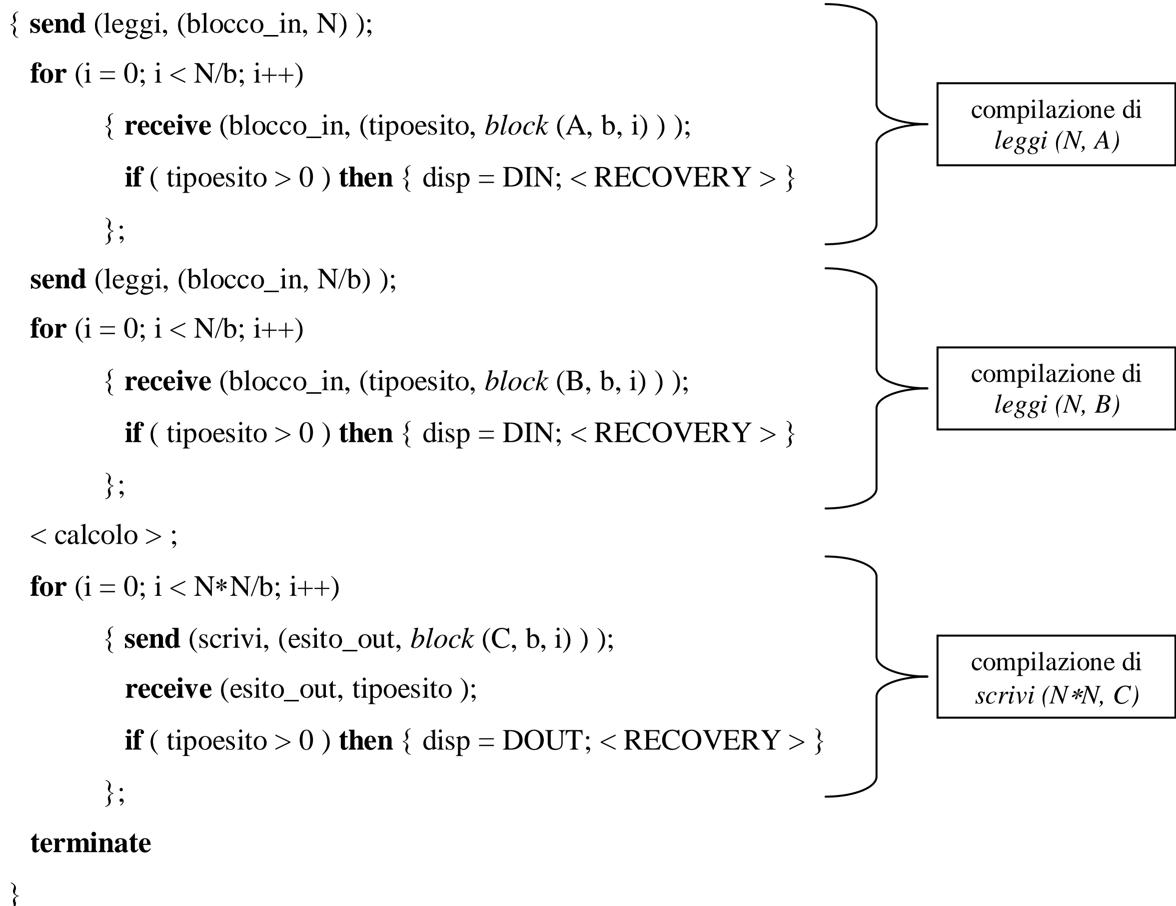
channel in blocco_in (N/b); esito_out (N/b), from_gmem (1);

// i canali d'ingresso hanno *grado di asincronia* uguale al numero di blocchi di A e B (ad esempio, 4); in tale modo i processi Driver non potranno mai bloccarsi nell'eseguire la send su tali canali //

channel out leggi, scrivi, eccezione, to_gmem;

// i canali d'ingresso dei processi-servizi sono asimmetrici; i loro canali di uscita sono simmetrici e identificati dal valore di una variabile channelname a cui, in seguito all'interazione con PROC2, verrà, rispettivamente, assegnato il valore *blocco_in* in Driver_in e *esito_out* in Driver_out //

// verrà fatto uso della funzione *block (X, size, j)* che, applicata ad un array *X*, denota il valore del *j*-esimo blocco di X, con blocchi di ampiezza *size* //



```
{ send (leggi, (blocco_in, N) );
  for (i = 0; i < N/b; i++)
        { receive (blocco_in, (tipoesito, block (A, b, i) ) );
          if ( tipoesito > 0 ) then { disp = DIN; < RECOVERY > }
        };
  send (leggi, (blocco_in, N/b) );
  for (i = 0; i < N/b; i++)
        { receive (blocco_in, (tipoesito, block (B, b, i) ) );
          if ( tipoesito > 0 ) then { disp = DIN; < RECOVERY > }
        };
  < calcolo > ;
  for (i = 0; i < N*N/b; i++)
        { send (scrivi, (esito_out, block (C, b, i) ) );
          receive (esito_out, tipoesito );
          if ( tipoesito > 0 ) then { disp = DOUT; < RECOVERY > }
        };
  terminate
}
```


dove:

RECOVERY :: { **send** (eccezione, (tipoesito, disp, PROC2));
terminate }

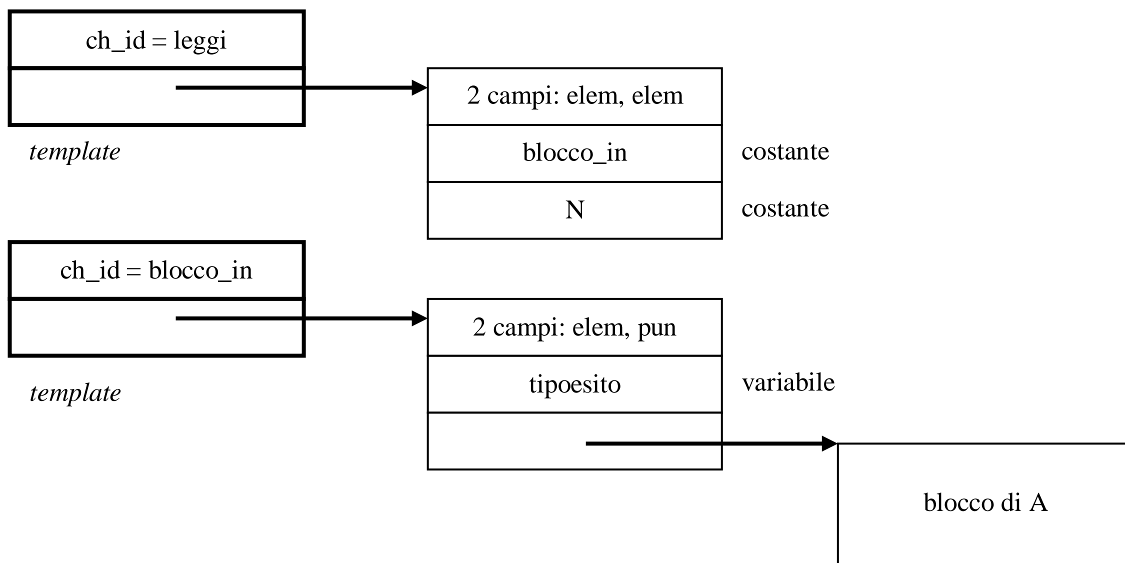
Per la compilazione, verranno usate le due procedure D-RISC:

SEND (Rch, Rmsg) RECEIVE (Rch, Rvtg)

dove il registro generale di indirizzo *Rch* contiene l'identificatore unico del descrittore di canale, *Rmsg* l'indirizzo logico della variabile messaggio nello spazio di indirizzamento del processo mittente, e *Rvtg* l'indirizzo logico della variabile targa nello spazio di indirizzamento del processo destinatario.

La compilazione in D-RISC utilizza le seguenti tecniche:

1) *comando send con valore del messaggio fornito da una espressione*: in tale caso, prima viene valutata l'espressione ed assegnato il valore ad una variabile *msg*, quindi viene chiamata la procedura SEND;
2) *nel caso di tuple,* gli elementi sono tutti consecutivi nella memoria virtuale;
3) *template per il passaggio dei parametri alle procedure SEND e RECEIVE:* per ogni istanza di comando *send* o *receive*, il compilatore prepara una struttura dati di due parole (*template*), la prima delle quali contiene il valore dell'identificatore di canale e la seconda l'indirizzo del messaggio o della variabile targa, ad esempio:



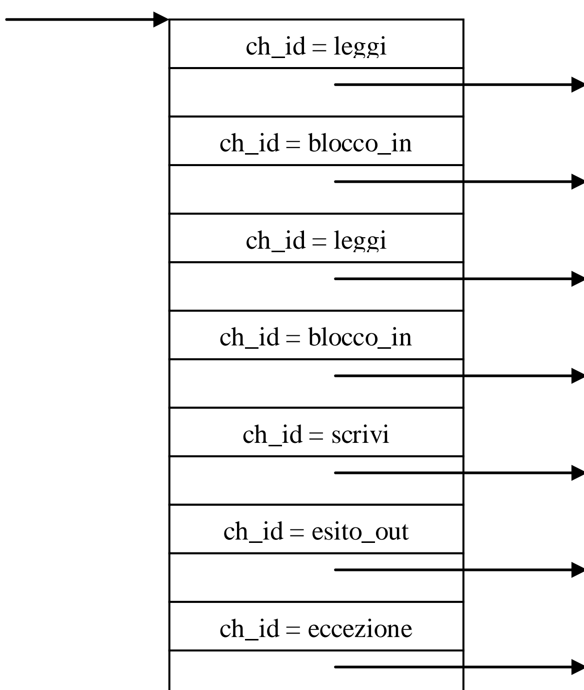
con la convenzione che, nella struttura (messaggio o variabile targa) puntata dal secondo campo del template, le costanti (come *N*) e i tipi elementari (come per *tipoesito*) vengono rappresentate direttamente, mentre per le strutture più complesse (come i blocchi di A) viene inserito il puntatore.

Per distinguere tutte le situazioni di interesse, la prima parola contiene un *descrittore* indicante:

- quanti campo seguiranno (ad esempio: 2),
- una maschera di bit tale che se l'i-esimo è = 0 allora l'i-esimo campo contiene un valore elementare (*elem*), altrimenti l'i-esimo campo contiene il puntatore ad una struttura complessa (*pun*).

Le informazioni del descrittore sono utilizzate dal compilatore per distinguere i vari tipi di oggetti in tuple, e (tranne casi speciali qui non contemplati, ad esempio per controlli) non saranno utilizzate nel codice compilato.

Nella memoria virtuale, tutti i template sono *allocati consecutivamente* secondo l'ordine più utile alla compilazione, ad esempio nell'ordine con cui sono incontrate le *send* e *receive* nel codice LC:



L'indirizzo base di questo vettore di template è contenuto nel registro generale di indirizzo *Rtemplate*.

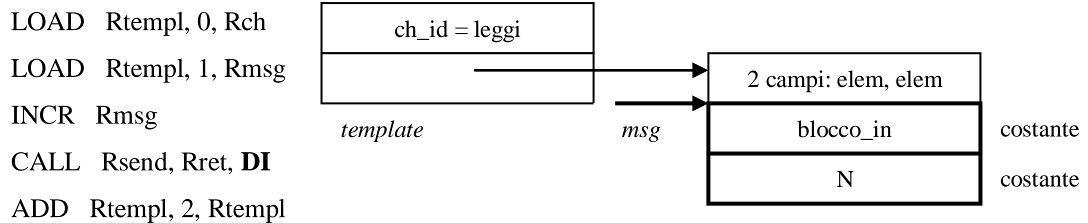
Segue il codice D-RISC in cui, al solito, il significato dei registri è auto esplicativo.

```
        MOV Rtemplate, Rtempl          // Rtempl verrà modificato via via //
```

// **send** (leggi, (blocco_in, N)); //

// la struttura puntata dal template è tutta fatta di costanti (vedi figura in precedenza), inizializzate nella memoria virtuale a tempo di compilazione; di conseguenza, il passaggio dei parametri alla procedura SEND è molto semplice:

```
        LOAD   Rtempl, 0, Rch
        LOAD   Rtempl, 1, Rmsg
        INCR   Rmsg
        CALL   Rsend, Rret, DI
        ADD    Rtempl, 2, Rtempl
```



// la riabilitazione delle interruzioni avverrà contestualmente al ritorno da procedura, eventualmente alla fine della commutazione di contesto //

// si noti che la procedura SEND, quando copierà il messaggio, ne conoscerà la lunghezza (in questo caso *L = 2*) attraverso l'apposito campo del descrittore di canale, inizializzato a compilazione //

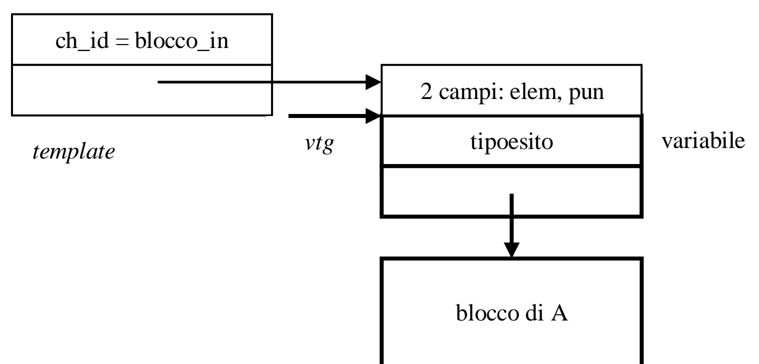
// **for** (i = 0; i < N/b; i++)

{ **receive** (blocco_in, (tipoesito, *block* (A, b, i)));

if (tipoesito > 0) **then** { disp = DIN; < RECOVERY > } }; //

// nel terzo campo della struttura puntata dal template va inizialmente scritto l'indirizzo base dell'array A; nelle iterazioni successive questo campo (il cui indirizzo è consecutivo a quello di *tipoesito*, e quindi consecutivo all'indirizzo di *vtg*) deve essere modificato per puntare ai blocchi successivi di A //

```
        CLEAR  Ri
        DIV    RN, Rb, Rnumblocchi
        LOAD   Rtempl, 0, Rch
        LOAD   Rtempl, 1, Rvtg
        INCR   Rvtg
LOOP_A: MUL    Rb, Ri, Roffset
        ADD    RA, Roffset, RAblocco
        STORE  Rvtg, 1, RAblocco
        CALL   Rreceive, Rret, , DI
        LOAD   Rvtg, 0, Resito
        MOVE   RDIN, Rdisp
        IF > 0   Resito, RECOVERY
        INCR   Ri
        IF <   Ri, Rnumblocchi, LOOP_A
        ADD    Rtempl, 2, Rtempl
```



// analogamente per la lettura dell’array B //

```
            LOAD   Rtempl, 0, Rch
            LOAD   Rtempl, 1, Rmsg
            INCR   Rmsg
            CALL   Rsend, Rret, DI
            ADD    Rtempl, 2, Rtempl

            CLEAR  Ri
            DIV    RN, Rb, Rnumblocchi
            LOAD   Rtempl, 0, Rch
            LOAD   Rtempl, 1, Rvtg
            INCR   Rvtg
LOOP_B:     MUL    Rb, Ri, Roffset
            ADD    RB, Roffset, RBblocco
            STORE  Rvtg, 1, RBblocco
            CALL   Rreceive, Rret, , DI
            LOAD   Rvtg, 0, Resito
            MOVE   RDIN, Rdisp
            IF > 0 Resito, RECOVERY
            INCR   Ri
            IF <   Ri, Rnumblocchi, LOOP_B
            ADD    Rtempl, 2, Rtempl
```

// compilazione della fase di calcolo : vedi esercitazioni precedenti //

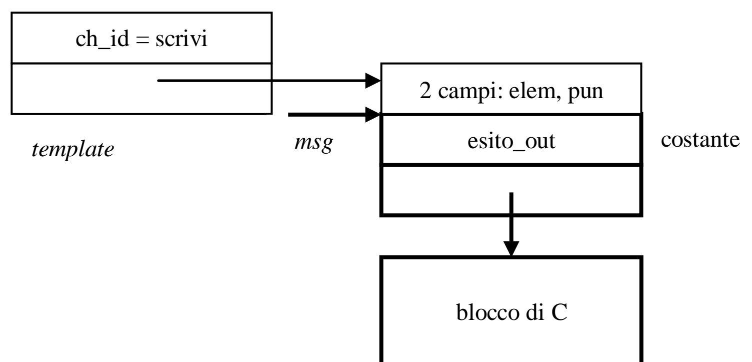
… … …

// **for** (i = 0; i < N*N/b; i++)

{ **send** (scrivi, (esito_out, *block* (C, b, i))); //

// la seconda parola del messaggio (puntato dal template) è la costante data dall’identificatore *esito_out,* inizializzata a tempo di compilazione; la terza parola del messaggio deve puntare al generico blocco di C //

```
            CLEAR  Ri
            MUL    RN, RN, RM
            DIV    RM, Rb, Rnumblocchi
LOOP_C:     LOAD   Rtempl, 0, Rch
            LOAD   Rtempl, 1, Rmsg
            INCR   Rmsg
            MUL    Rb, Ri, Roffset
            ADD    RC, Roffset, RCblocco
            STORE  Rmsg, 1, RCblocco
            CALL   Rsend, Rret, DI
```



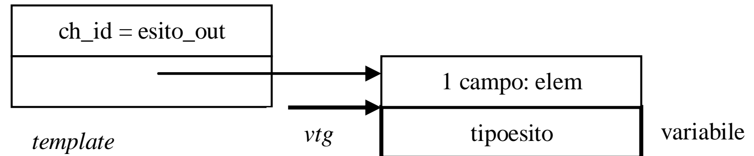
// **receive** (esito_out, tipoesito);

if (tipoesito > 0) **then** { disp = DOUT; < RECOVERY > } };

terminate } //

// la struttura puntata dal template è costituita dal descrittore e da un'altra sola parola relativa alla variabile *tipoesito* //

```
LOAD   Rtempl, 2, Rch
LOAD   Rtempl, 3, Rvtg
INCR   Rvtg
CALL   Rreceive, Rret, DI
LOAD   Rvtg, 0, Resito
MOVE   RDOUT, Rdisp
IF > 0   Resito, RECOVERY
INCR   Ri
IF <   Ri, Rnumblocchi, LOOP_C
ADD   Rtempl, 4, Rtempl
GOTO   FINE
```

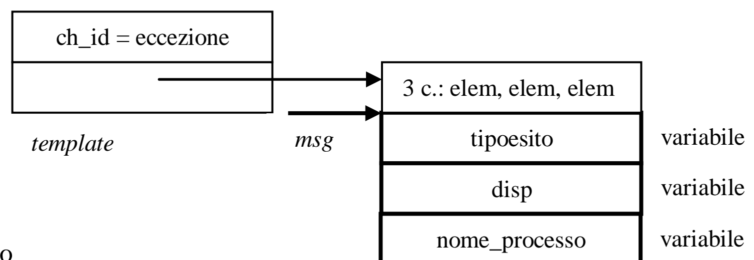


// RECOVERY :: { **send** (eccezione, (tipoesito, disp, PROC2);

terminate } //

// va formato il valore del messaggio, scrivendo nella seconda, terza e quarta parola della struttura puntata da template, rispettivamente l'esito negativo ricevuto in precedenza, il nome del dispositivo, e il nome del processo PROC2; in quest'ultimo caso, il nome viene letto dalla prima parola del PCB, il cui indirizzo è presente in un registro generale//

```
RECOVERY:  LOAD   Rtempl, 0, Rch
           LOAD   Rtempl, 1, Rmsg
           INCR   Rmsg
           STORE   Rmsg, 0, Resito
           STORE   Rmsg, 1, Rdisp
           LOAD   Rpcb, 0, Rnome:processo
           STORE   Rmsg, 2, Rnome_processo
           CALL   Rsend, Rret, DI
FINE:      END
```



La memoria virtuale di PROC2 contiene gli oggetti elencati di seguito; va considerato che, per ragioni di protezione, ogni oggetto deve essere contenuto in un numero intero di pagine:

- codice derivato dalla compilazione vista sopra;
- strutture A, B, C, ed altre variabili locali;
- codice delle procedure SEND, RECEIVE;
- tabella dei template;
- strutture dati puntate dai template (descrittori, messaggi, variabili targa);

- tabella dei canali;
- descrittori dei canali *blocco_in, esito_out, leggi, scrivi, eccezione, to_gmem, from_gmem*;
- codice delle procedure per scheduling a basso livello, handler eccezioni, handler interruzioni;
- strutture dati per trattamento eccezioni e trattamento interruzioni;
- testa della lista pronti;
- PCB;
- Tabella di Rilocazione.

Inoltre, le seguenti strutture dati sono allocate staticamente nella memoria virtuale di PROC2 solo se vengono usati *metodi statici*, con indirizzi logici, per risolvere il problema della condivisione mediante riferimenti indiretti:

- PCB dei processi Driver_in, Driver_out, ECC, Gestore della Memoria,
- PCB di tutti gli altri processi creabili,
- variabili targa dei *canali leggi, scrivi, eccezione*, *to_gmem.*

Altrimenti, usando il *metodo a capability*, tutti questi oggetti non sono presenti staticamente nella memoria virtuale di PROC2 e vi vengono, invece, allocati dinamicamente se e quando necessario.

b) Nel grafo dei processi disegnato all'inizio del punto *a)* i canali *blocco_in* ed *esito_out* potrebbero anche avere come mittente le unità di ingresso-uscita I/O-DIN e I/O-OUT, così da abbassare il numero di comunicazioni tra processi.

In generale, poiché I/O-DIN e I/O-OUT possono essere implementate come *processi esterni*, la presenza esplicita di *Driver_in* e *Driver_out* non è necessaria, in quanto le loro funzionalità possono essere affidate alle rispettive unità di I/O.

Il grafo dei processi rimane quindi inalterato, a patto di sostituire Driver_in e Driver_out con I/O-DIN e I/O-DOUT rispettivamente. I canali di comunicazione conservano gli stessi nomi.

Potenzialmente siamo nelle condizioni di *non* dover ricompilare PROC2.

Affinché questo sia vero occorre, però, verificare anche un'altra condizione, e cioè che *le procedure SEND e RECEIVE, con partner processi esterni, rimangano le stesse*.

<u>Lo studio di queste condizioni è oggetto della Domanda 2</u>.

Domanda 2

a) Nelle sue linee essenziali, il supporto a tempo di esecuzione di *send* e *receive*, con partner entrambi processi interni, rimane lo stesso anche quando uno dei partner è un processo esterno.

Ciò è vero per la parte di *comunicazione vera e propria*, in quanto il supporto alle comunicazioni con i processi esterni sfrutta ancora la *memoria condivisa* (descrittori di canali, variabili targa), e nel codice è invisibile il fatto che la condivisione di memoria sia realizzata con la tecnica del Memory Mapped I/O o con DMA o entrambe. Questo aspetto necessita di essere approfondito come segue.

> Il supporto delle primitive, *eseguite da processi esterni*, differisce sostanzialmente da quello delle primitive eseguite da processi interni, sia per ragioni di ottimizzazioni (una certa unità di I/O può essere specializzata solo per certe forme di comunicazione e certi canali), che per ragioni algoritmiche e di correttezza di funzionamento: in particolare
>
> 1. per un processo esterno, *l'attesa è di tipo attivo*,

2. l'unità di I/O realizza *l'indivisibilità* delle sequenze di letture-scritture per la manipolazione del descrittore di canale,
3. la realizzazione delle primitive dipende dal grado di "intelligenza" delle unità di I/O, secondo quanto vedremo al *punto c).*

Riguardo al punto 2, notiamo che la sola disabilitazione delle interruzioni da parte del processore non è sufficiente a garantire l'indivisibilità, in quanto siamo in presenza di due processori (quello centrale e quello di I/O) che possono tentare di manipolare il descrittore di canale contemporaneamente con il rischio di portarlo in uno stato inconsistente (vedi Cap. VIII, sez. 2.5).

Nel supporto di una *send*, una possibile sequenza da rendere indivisibile è la seguente(si veda l'algoritmo del Cap. VIII, sez. 7.2): lettura di *wait*, scrittura di *wait*; un'altra è: lettura di *wait*, letture e scritture per la copia del messaggio nel buffer, eventuale scrittura di *wait* e di PCB_rif.

Se non si dispone di meccanismi generali validi per architetture multiprocessor (lock, unlock), nel nostro caso la soluzione consiste nel *delegare interamente all'unità di I/O il compito di garantire l'indivisibilità di certe sequenze, allocando fisicamente il descrittore di canale nella memoria di I/O*. L'unità di I/O è progettata in modo che, una volta che essa stessa inizi una sequenza di accessi indivisibile, provvede a completarla *senza ascoltare le richieste del processore* di accesso alla memoria di I/O; oppure in modo che, una volta che ascolti la prima richiesta da parte del processore relativamente ad una sequenza indivisibile (il riconoscimento può essere effettuato in funzione dell'indirizzo associato alla richiesta), provvede a *servire solo le richieste da parte del processore* finché la sequenza non sarà completata.

Nessun problema di indivisibilità si ha su variabili targa, mentre la lista pronti non è utilizzata dai processi esterni.

Con questa realizzazione dei processi esterni, nessuna modifica viene apportata al codice del supporto di *send/receive*, *eseguite da processi interni*, per quanto riguarda la *fase di comunicazione vera e propria*.

Quello che cambia nel supporto di una primitiva, *eseguita da un processo interno*, è la fase di *scheduling a basso livello*, ed in particolare la *sveglia* del partner: nel caso che il partner sia esterno, la sveglia consiste semplicemente in una comunicazione di I/O (una STORE in Memory Mapped I/O) per segnalare l'evento all'unità di I/O (ovviamente non c'è nessuna manipolazione di PCB).

b) Le differenze di cui sopra, circa il supporto di una primitiva eseguita da un processo interno, possono essere rese visibili prevedendo *procedure diverse* nel caso di partner interno e nel caso di partner esterno: in tal caso, con riferimento alla Domanda 1b), il processo applicativo deve essere ricompilato.

Alternativamente, *le differenze possono essere contemplate all'interno di una stessa procedura*, distinguendovi le azioni da svolgere a seconda che il partner sia interno o esterno: questa informazione può essere data attraverso il descrittore di canale. In tale realizzazione, nella Domanda 1b) il processo applicativo non deve essere ricompilato.

c) Questo aspetto completa la differenza nell'implementazione del supporto a primitive *eseguite da processi esterni*, quindi con partner processi interni.

Nel caso "capace", la fase di comunicazione vera e propria è effettuata normalmente via memoria condivisa, mentre la sveglia del partner interno è delegata alla CPU (al processo attualmente in esecuzione) mediante una interruzione, il cui messaggio associato è del tipo (evento = sveglia processo, dato = riferimento a PCB). L'Handler esegue la funzionalità di sveglia.

Nel caso "non capace", tutta l'implementazione di *send* o *receive* è delegata al processo in esecuzione sulla CPU mediante una interruzione, il cui messaggio associato è del tipo (evento = esegui *send* / *receive*, dato = riferimento ad una struttura dati condivisa "template" per i parametri della primitiva). L'Handler esegue l'intera funzionalità di *send* / *receive*, inclusa la sveglia del processo interno partner dell'unità di I/O.

Domanda 3

a) Poiché siamo nel caso di metodo *statici*, con indirizzi logici, per realizzare la condivisione mediante riferimenti indiretti, tutti gli oggetti del supporto sono *allocati staticamente nelle memorie virtuali* dei processi.

1. Falso. Il numero dei processi attivi ha impatto su aspetti come il numero di PCB presenti nella memoria virtuale di ogni processo o il numero di partner di un processo-servizio (numero di processi mittenti di un canale asimmetrico in ingresso, numero e tipo di canali simmetrici in uscita). Dovendo allocare le memorie virtuali staticamente, non ha senso un dimensionamento dinamico.
2. Vero. Una strategia di allocazione statica deve prevedere un "dimensionamento sul massimo" (numero di PCB, canali dei servizi), ben sapendo che per i processi applicativi è previsto un numero massimo di "contenitori" che verranno occupati di volta in volta dai processi effettivamente creati.
3. Vero *se* il metodo statico adottato è quello degli indirizzi logici coincidenti. Con il metodo degli indirizzi logici distinti, il compilatore può allocare ogni memoria virtuale senza vincoli sulla posizione degli oggetti condivisi.
4. Vero, ma si tratta di una caso con implicazioni molto pesanti. Questo aspetto è in relazione con il problema di *inizializzare le Tabelle di Rilocazione dei processi*. Se l'allocazione della memoria *principale* ai PCB fosse *statica*, allora tale inizializzazione può avvenire a tempo di compilazione.
5. Vero, sotto condizioni molto più favorevoli rispetto al caso precedente. Se l'allocazione della memoria *principale* ai PCB è (anch'essa) *dinamica*, allora l'inizializzazione delle Tabelle di Rilocazione dei processi non può (ovviamente) avvenire a tempo di compilazione, mentre può senz'altro avvenire a tempo di creazione-caricamento da parte del gestore della memoria, il quale ha una visione aggiornata (dinamica) dello stato di allocazione della memoria principale ed è quindi in grado di riempire in modo corretto i campi di tali Tabelle.

b) Con il metodo *a capability*, molti oggetti, in particolare oggetti condivisi, sono allocati dinamicamente nelle memorie virtuali dei processi. Questo si applica a PCB di altri processi, variabili targa, canali dei servizi, ecc.

L'unico caso, nello specifico supporto di LC, di conoscenza del massimo numero di processi (punto 2) si ha nei canali asimmetrici d'ingresso dei processi-servizi, problema che può comunque essere minimizzato (ma non annullato) facendo uso di channelname anche per i canali d'ingresso.

Tutti i punti relativi ai PCB (3, 4, 5) non hanno più significato: ogni PCB di un altro processo è allocato dinamicamente se e quando necessario e nella posizione ritenuta più conveniente in quell'istante.