Architettura degli Elaboratori, 2006-07

Terza Esercitazione di Verifica Intermedia

Consegna: lezione di venerdì 15 dicembre, ore 11

L'elaborato, da presentare in una forma leggibile agevolmente, deve contenere spiegazioni chiare ed esaurienti, utilizzando la corretta terminologia ed i concetti del corso.
Insieme a nome e cognome indicare l'anno di corso di ogni studente.

Domanda 1

Si consideri il programma della Domanda 1 della Seconda Esercitazione. Il programma viene eseguito da un sistema che, oltre alle caratteristiche di quello della Domanda 2 della Seconda Esercitazione, possiede una memoria cache con le seguenti caratteristiche:

- operante su domanda,
- capacità di 64K parole e blocchi di 8 parole,
- indirizzamento con il metodo associativo su insiemi, con 2 blocchi per insieme,
- scritture gestite con il metodo Write Through.

La memoria principale è interallacciata con 4 moduli. Non esiste cache secondaria.

a) Valutare il tempo di completamento del programma, la performance, e l'efficienza relativa nell'uso della cache. Spiegare il procedimento seguito.

b) Ricavare la Parte Operativa dell'unità cache di questo sistema, spiegando come è stata ottenuta, e verificare che il tempo di accesso in cache, in assenza di fault, sia quello ipotizzato al punto *a).*

Domanda 2

a) Si consideri il programma corrispondente alla seguente computazione:

$$i = 0 \,..\, N - 1 : A[i] = F\,(A[i], B)$$

con A e B array di N = 32K interi, ed F funzione data (libreria). Spiegare come, per questo programma, l'uso della cache operante su domanda può essere ottimizzato, e determinare il corrispondente numero di fault. Nel far questo si consideri che il set di istruzioni è arricchito da opzioni associate alle istruzioni LOAD e STORE per indicare che il blocco contenente la parola riferita, una volta allocato in cache, può essere deallocato dalla cache in qualunque momento, oppure che non deve essere deallocato (finché non viene eseguita una istruzione che indica che il blocco può essere deallocato in qualunque momento).

b) Dimostrare che dei tre metodi di indirizzamento, visti a proposito della memoria cache, solo il metodo completamente associativo può essere utilizzato (e di fatto lo è) anche nella gerarchia memoria virtuale - memoria principale.

Domanda 3

Il messaggio di interruzione di un certo sottoinsieme di unità di I/O è costituito dalla coppia di parole (codice evento = sveglia processo, nome di processo). L'Handler associato a questo evento provvede a svegliare il processo il cui nome unico è dato dalla seconda parola. Il sistema prevede 4 code dei processi pronti, ognuna delle quali corrisponde ad una diversa classe a cui che i processi possono appartenere. La funzionalità di sveglia inserisce il PCB del processo da svegliare in fondo alla coda pronti che compete a quel processo.

a) Scrivere l'Handler, spiegando accuratamente come si è ragionato, le strutture dati utilizzate, ed il codice assembler.

b) Valutare l'intervallo di tempo necessario a trattare questo tipo di interruzione, a partire dall'istante in cui l'interruzione viene rilevata fino all'istante in cui si conclude l'elaborazione dell'Handler.

Domanda 4

Il programma della Domanda 1 è completato in modo che gli array *A, B, C* siano letti, uno alla volta, da un certo dispositivo USER_DEV e che l'array *C* modificato sia scritto sullo stesso dispositivo.

Si suppone che, a livello del linguaggio delle applicazioni, USER_DEV sia visto attraverso i comandi *leggi(n, x)* e *scrivi (n, x),* dove *x* è il nome di una variabile rappresentata da *n* byte.

Il sistema operativo è a processi cooperanti a scambio di messaggi.

a) Mostrare e spiegare la *compilazione* in linguaggio concorrente del programma *in un processo* MY_PROG, supponendo che all'unità I/O_USER_DEV sia associato un processo Driver_USER_DEV. In seguito all'eventuale esito anomalo dei comandi *leggi* e *scrivi*, il processo MY_PROG invia un messaggio "errore" al dispositivo SYSTEM_DEV dopo di che termina. *Mostrare e spiegare tutti gli oggetti contenuti nella memoria virtuale del processo MY_PROG*.

b) Spiegare in seguito a quali eventi il processo MY_PROG può *transire in stato di attesa*, ed in seguito a quali eventi può essere *risvegliato*.

c) Spiegare se la versione *assembler* di MY_PROG va modificata o meno, ed eventualmente come, nel caso che sia eliminato il processo Driver_USER_DEV,

d) Spiegare se, nella versione *c),* è visibile a MY_PROG quale modello, DMA o Memory Mapped I/O, è adottato per i trasferimenti di I/O.

Soluzione

Domanda 1

a) Come nella soluzione della Domanda 2 della Seconda Esercitazione, ha senso valutare le prestazioni considerando solo il loop della procedura:

```
LOOP:   LOAD   RA, Ri, Ra
        IF>=   Ra, CONT1
        SUB    0, Ra, Ra
CONT1:  MOD    Ra, RN, Rj
        LOAD   RC, Ri, Rc
        LOAD   RB, Rj, Rb
        IF>=   Rc, Rb, CONT2
        STORE  RC, Ri, Rb
CONT2:  INCR   Ri
        IF<    Ri, RN, LOOP
```

Il tempo di completamento si valuta come:

$$T_c = T_{c\text{-}id} + T_{fault}$$

dove $T_{c\text{-}id}$ è il tempo medio di completamento in assenza di fault di cache, e T_{fault} è il tempo medio speso per gestire tutti i fault di cache che hanno luogo durante l'esecuzione del programma.

Nella valutazione di $T_{c\text{-}id}$ per il tempo di accesso in memoria va, quindi, preso il tempo di accesso alla cache t_c (come "visto" dal processore), nel nostro caso (*metodo associativo su insiemi*) dato da:

$$t_c = 3\tau$$

Riprendendo, dalla soluzione dell'Esercitazione 2, l'espressione del tempo di completamento, e sostituendo t_c a t_a, si ottiene:

$$T_{c\text{-}id} = N\,(\,82{,}5\ \tau + 12{,}5\ t_a\,) = N\,(\,82{,}5\ \tau + 12{,}5\ t_c\,) = 120\ N\ \tau$$

che risulta di un ordine di grandezza inferiore rispetto all'architettura senza cache.

La penalità T_{fault} si valuta come:

$$T_{fault} = N_{fault} * T_{trasf}$$

dove N_{fault} è il numero medio di fault di cache che hanno luogo durante tutta l'esecuzione del programma, e T_{trasf} è il tempo necessario per effettuare il trasferimento di un blocco di cache dal livello superiore (memoria principale nel nostro caso) alla cache.

Nel seguito, assumeremo che

a) per questo programma, il metodo associativo su insiemi non comporti un aumento apprezzabile della probabilità di fault rispetto al caso completamente associativo: specie se il numero di blocchi per insieme viene portato a 4;

b) le scritture in memoria principale (metodo *Write Through*) non costituiscano "collo di bottiglia" (cioè, non comportino attese ulteriori da parte della CPU), sia perché la memoria interallacciata permette fino a 4 scritture contemporaneamente su richiesta della MMU, sia perché è possibile pensare di aumentare ulteriormente il grado di asincronia dei messaggi dalla CPU alla memoria.

L'espressione per il tempo di trasferimento del blocco:

$$T_{trasf} = \frac{\sigma}{m} t_a + m\tau$$

vale nel caso che l'unità cache effettui due richieste distinte alla memoria interallacciata per i due trasferimenti di $m = 4$ parole. Le prestazioni possono essere un po' migliorate assumendo che l'unità cache richieda con uno stesso messaggio il trasferimento di 8 parole (intero blocco) e delegando alla memoria il compito di effettuare due trasferimenti consecutivi di 4 parole ciascuno. Di conseguenza si ha:

$$T_{trasf} = \frac{\sigma}{m}\tau_M + 2T_{tr} + m\tau = 124\tau$$

in quanto

- la memoria lavora per σ/m cicli di clock τ_M consecutivi,
- un ritardo T_{tr} (= $20\ \tau$) e gli m cicli di clock τ per la scrittura di m parole in cache si sovrappongono al ciclo di clock della memoria (= $40\ \tau$), eccetto che per le ultime m parole,
- occorre considerare due ritardi T_{tr}, uno per il messaggio iniziale di richiesta dall'unità cache ed uno per quello di risposta dalla memoria delle ultime m parole.

Per valutare il numero medio di fault, consideriamo intanto che il numero di fault generato dalle istruzioni (programma chiamante e procedura) ed il numero di fault per dati al di fuori del loop della procedura sono decisamente trascurabili rispetto al numero di fault provocato dagli accessi agli array A, B, C durante il loop della procedura.

Consideriamo che gli array A e C sono scanditi per intero; quindi ogni loro blocco, una volta in cache, viene utilizzato per 8 accessi in lettura. Inoltre, su ogni blocco di C si ha anche riuso per altri 8 accessi in scrittura: accessi che, quindi, non generano fault. Relativamente ad A e C, il numero di fault è dunque:

$$N_{fault,A-C} = 2\frac{N}{\sigma} = \frac{N}{4}$$

Ben diversa è la situazione per l'array B, in quanto l'indice j di ogni suo elemento è calcolato, per ogni valore di i, in funzione del valore di $A[i]$, quindi in maniera *impredicibile* (in assenza di informazioni addizionali sul contenuto di A). Anche imponendo il riuso dei blocchi di B finché è possibile mantenerli in cache (usando l'opzione "non deallocare se possibile" nelle istruzioni di tipo LOAD B[j], vedi Domanda 2), *non è possibili quantificare analiticamente la località ed il riuso negli accessi a B.*

In una valutazione del caso peggiore, tutti gli accessi a B danno luogo a fault di cache:

$$N_{fault,B} = N$$

da cui:

$$N_{fault} = N_{fault,A-C} + N_{fault,B} = \frac{N}{4} + N = \frac{5}{4}N$$

Quindi:

$$T_{fault} = N_{fault} * T_{trasf} = 155N\tau$$

che risulta addirittura superiore al tempo di completamento ideale, a causa dell'assenza di cache secondaria e della scarsa località e riuso delle informazioni del programma.

Il tempo di completamento e l'efficienza relativa nell'uso della cache sono dati da:

$$T_c = T_{c\text{-}id} + T_{fault} = 275\ N\ \tau$$

$$\varepsilon = \frac{T_{c-id}}{T_c} = 0{,}44$$

Il tempo medio di elaborazione e la performance sono dati da :

$$T = \frac{T_c}{k} = \frac{T_c}{9N} = 30{,}5\tau = 7{,}6\ n\text{sec}$$

$$\wp = \frac{1}{T} = 131\ MIPS$$

Il miglioramento delle prestazioni rispetto al caso dell'architettura senza cache è "solo" di un fattore 4 o, equivalentemente, del 75%.

Si noti che, anche se il programma valutato pone problemi dal punto di vista della "predicibilità" delle prestazioni, le prestazioni che verranno *misurate* di volta in volta saranno probabilmente superiori a quelle calcolate analiticamente riferendosi al caso peggiore nello sfruttamento della cache.

Una valutazione analitica più favorevole può riferirsi al caso in cui il numero di fault di cache, per gli accessi a B, sia uguale a

$$N_{fault,B} = \frac{N}{2}$$

ottenendo:

$$T_{fault} = 93N\tau$$

$$T_c = 217N\tau$$

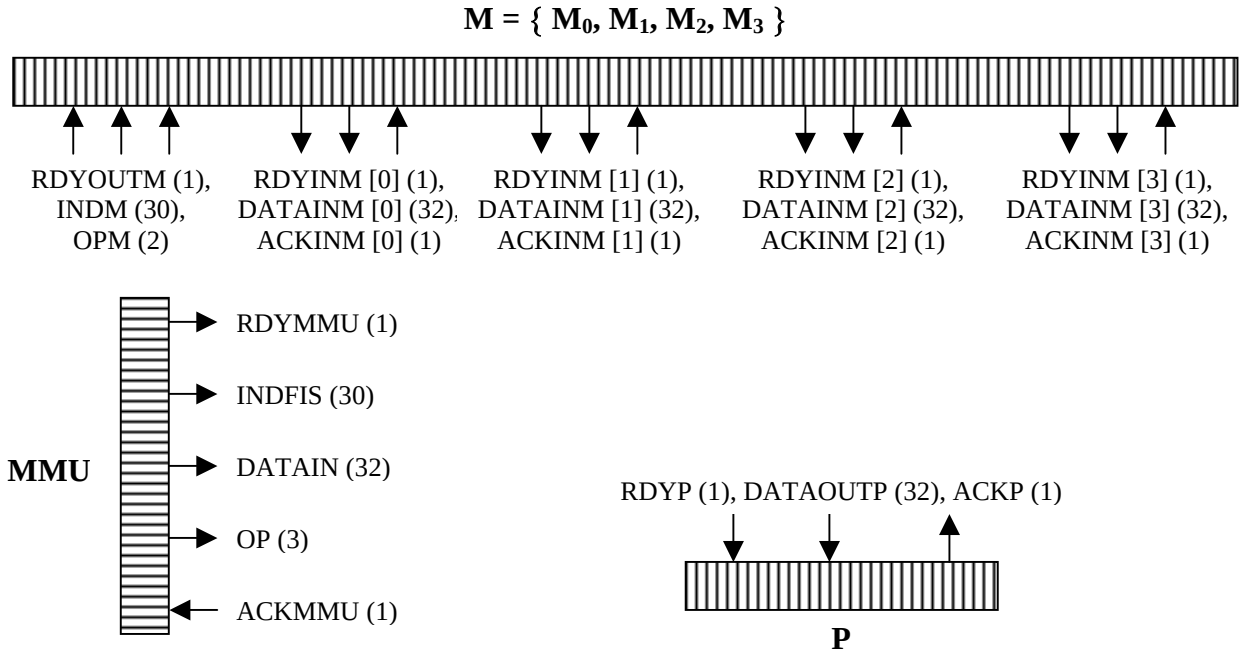
$$\varepsilon = 0{,}57$$

$$T = 24\tau = 6\ n\text{sec}$$

$$\wp = 166\ MIPS$$

b) Per questa domanda faremo riferimento al contenuto delle *Note "Parte Terza: guida alla studio e integrazioni", sez. 2.3*, dove sono spiegati alcuni aspetti implementativi di un'unità cache con il metodo associativo si insiemi, con due blocchi per insieme. In particolare, in tali Note è descritto come rilevare la condizione di fault (F = 1) e come ricavare il bit B da concatenare in mezzo ai campi SET e D dell'indirizzo di memoria principale per ottenere l'indirizzo di cache in assenza di fault.

Lo schema delle interfacce, tutte a transizione di livello, è schematicamente il seguente:



Nonostante siano interfacce generali, il protocollo sarà complessivamente a domanda-risposta (non useremo gli ACK).

Useremo i seguenti simboli:

- un funzione *operazione* applicata a OP, e realizzata come rete combinatoria, produce il valore booleano RW tale che: RW = 0 corrisponde alla richiesta di lettura e RW = 1 a quella di scrittura;
- una funzione *riuso* applicata a OP, e realizzata come rete combinatoria, produce un valore booleano che, se vero, sta a significare che il blocco è soggetto alla politica "non deallocare se possibile";
- MC = componente logico memoria cache di 64K parole;
- TABC = componente logico memoria di 8K entrate, corrispondente alla *Tab d*elle Note;
- *riloca* = funzione, realizzata come rete combinatoria, per il calcolo dei bit F e di B, come definito nella Note;
- *aggiorna* = funzione, realizzata come rete combinatoria, per aggiornare l'entrata di TABC riferita; riguarda l'aggiornamento del bit di presenza (messo ad 1 ad ogni accesso al blocco), del bit LRU (messo ad 1ad ogni accesso al blocco), e del bit RIUSO;
- *alloca:* funzione, realizzata come rete combinatoria, che, in seguito a fault, genera il valore del bit B per la scelta del blocco da sostituire ed aggiorna TABC. Tale funzione opera in maniera banale sui bit LRU e RIUSO associati ai blocchi dell'insieme riferito.

Vediamo il microprogramma dell'unità cache, relativamente alla fase di traduzione dell'indirizzo e di accesso alla memoria cache:

0. (RDYMMU = 0) nop, 0;

   (= 1) reset RDYMMU, INDFIS → IND, DATAIN → DATA, *operazione* (OP) → RW,

   riloca (TABC [INDFIS.SET], INDFIS.TAG) → (F, B),

   aggiorna (TABC [INDFIS.SET], *riuso* (OP)) → TABC [INDFIS.SET], 1

   // oltre a valutare i bit F, B come spiegato nelle Note, è stata aggiornata l'entrata di TABC come indicato sopra //

1. (F, RW = 0 0) MC[IND.SET • B • IND.D] → DATAOUTP, set RDYOUTP, 0;

   (= 0 1) DATA → MC [IND.SET • B • IND.D], set RDYOUTP, 0;

   (= 1 0) IND → INDM, 'trasferisci blocco' → OPM, set RDYOUTM,

   alloca (TABC [INDFIS.SET]) → (B, TABC [INDFIS.SET]),

   0 → I, 0 → D, 3;

   (= 1 1) *alloca* (TABC [INDFIS.SET]) → (B, TABC [INDFIS.SET]), 2

2. // questa microistruzione viene eseguita in caso di fault in seguito a richiesta di scrittura; si tratta di allocare il blocco in cache, senza provocare trasferimento dalla memoria; si ricordi che questa CPU adotta il metodo Write-Through //

   DATA → MC [IND.SET • B • IND.D], set RDYOUTP, 0;

Segue la parte di microprogramma per effettuare la sostituzione del blocco. La richiesta è già stata effettuata nella terza frase della microistruzione 1. La memoria interallacciata è costituita da 4 unità operanti in parallelo, capaci di

- assegnare il valore corretto del campo displacement (D) dell'indirizzo;
- effettuare due accessi consecutivi, come spiegato nella parte *a).*

In altre parole, tutte le unità ricevono il valore dell'indirizzo fisico; ognuna assegna al campo D il valore del proprio indice (da 0 a 3); viene effettuato il primo accesso in lettura di 4 parole in parallelo; effettuato il primo accesso, ogni unità incrementa nuovamente il campo D dell'indirizzo ed effettua un secondo accesso.

L'unità cache utilizza la tecnica del *controllo residuo* per la scrittura in cache delle 4 parole ricevute dalla memoria. Allo scopo, nella microistruzione 1 è stato inizializzato a zero un registro I di 3 bit usato come contatore; l'uscita I_m di tale registro (i due bit meno significativi) comandano un commutatore la cui uscita è il valore corrente della variabile di condizionamento RDYINM [I_m] e il valore del dato DATAINM [I_m] da scrivere nella memoria cache MC. Un registro D di 3 bit, inizializzato a zero nella microistruzione 1, contiene il displacement dell'indirizzo di cache.

Nella microistruzione 3 verranno ricevute e scritte le prime 4 parole del blocco, nella microistruzione 4 le seconde 4 parole:

3. (I_0, RDYINM [I_m] = 0 0) nop, 3;

   (= 0 1) reset RDYINM [I_m], DATAINM [I_m] $\rightarrow$ MC [IND.SET • B • D], I + 1 $\rightarrow$ I, D + 1 $\rightarrow$ D, 3;

   (= 1 –) 0 $\rightarrow$ I, 4

4. (I_0, RDYINM[I_m] = 0 0) nop, 3;

   (= 0 1) reset RDYINM [I_m], DATAINM [I_m] $\rightarrow$ MC [IND.SET • B • D], I + 1 $\rightarrow$ I, D + 1 $\rightarrow$ D, 3;

   (= 1 –) nop, 0 // nop eliminabile //

Lo schema della Parte Operativa è ottenuto direttamente dal microprogramma applicando la metodologia generale del Cap. III e utilizzando le reti combinatorie *operazione, riuso, riloca, aggiorna* e *alloca* in aggiunta a componenti standard.

Dal microprogramma si verifica che, come ipotizzato al punto *a,)* il tempo di accesso in cache in assenza di fault è uguale a 3τ, dei quali 1τ speso nella MMU e 2τ spesi nell'unità cache (microistruzioni 0 e 1). Nel caso di fault e accesso in scrittura, la penalità è 1τ, quindi trascurabile.

Si osservi che, in base alla teoria del Cap. III, sarebbe stato possibile spendere solo 1τ nell'unità cache, fondendo le microistruzioni 0 e 1 in una sola microistruzione; questo avrebbe comportato la stabilizzazione in cascata di due memorie (TABC e MC), oltre a reti combinatorie "veloci", all'interno della funzione σ_{PO}, e quindi un valore del ciclo di clock quasi doppio. Questo non sarebbe stato accettabile, in quanto l'unità cache è implementata sullo stesso chip del processore, e quest'ultimo ha un ciclo di clock che, relativamente alla funzione σ_{PO}, prevede la stabilizzazione di una sola memoria di registri (RG) e di una ALU in cascata.

Domanda 2

a) É lecito assumere che

1. la cache sia quella della Domanda 1,
2. la procedura *F* sia contenuta in “pochi” blocchi,
3. in assenza di altre informazioni, “tutte” le componenti dell’array B siano utilizzate in lettura per ogni applicazione di *F* (per ogni valore di *i*).

Il punto 3 è alla base dell’ottimizzazione che è possibile introdurre nell’uso della memoria cache per questo programma.

Intanto notiamo che il tempo di completamento del programma, e quindi il tempo di completamento ideale, ha ordine di grandezza proporzionale a N^2.

Per ogni valore di *i*, viene utilizzato un blocco diverso dell’array A. Quindi, non essendoci riuso per A, le istruzioni tipo LOAD A[i] contengono l’opzione “dealloca pure”.

Poiché gli array A e B sono ampi 32K parole e la cache ha capacità 64K parole, è possibile fare in modo che, dopo l’iterazione con $i = 0$, tutti i blocchi di B siano allocati in cache e non vengano più rimossi: le istruzioni tipo LOAD B[j] hanno l’opzione “non deallocare se possibile”. Viene così fatto il miglior uso possibile della proprietà del *riuso* che caratterizza l’array B in questo programma.

Ne consegue che il numero di fault per l’esecuzione di tutto il programma è (trascurando, al solito, pochi fault per le istruzioni):

$$N_{fault} = 2\frac{N}{\sigma}$$

Quindi la penalità T_{fault}, avendo ordine di grandezza proporzionale a *N*, è trascurabile rispetto al tempo di completamento ideale, e la cache risulta sfruttata praticamente al 100%.

b) La gerarchia di memoria “memoria virtuale (MV) – memoria principale (M)” deve essere vista applicata *ad ogni singolo programma (processo):* ogni processo ha la propria MV e tutte le MV sono allocate dinamicamente nella stessa M.

Ne consegue che non è possibile utilizzare alcun metodo di indirizzamento che preveda una associazione a priori tra un certo identificatore di pagina logica IPL ed un certo identificatore di pagina fisica IPF, in quanto pagine logiche con lo stesso IPL in processi *diversi* verrebbero ad essere messe in corrispondenza con la stessa pagina fisica. Ciò esclude il metodo diretto, ma esclude anche qualunque altro metodo, come quello associativo su insiemi, che metta in corrispondenza fissa pagine logiche con insiemi di pagine fisiche (di cardinalità minore del numero dei processi).

L’unica possibilità è che la corrispondenza tra pagine logiche di ogni MV e pagine fisiche di M sia *completamente dinamica* e stabilita volta per volta all’atto dell’allocazione di una pagina logica.

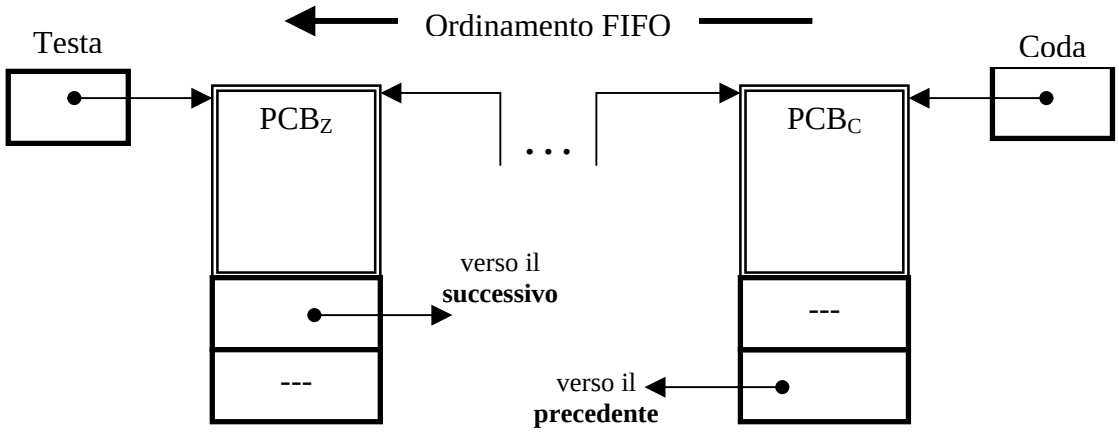
Detta μ la funzione di rilocazione di un processo, ed A e B due distinti processi, la coincidenza

$$\mu_A\,(IPL_A) = \mu_B\,(IPL_B)$$

si ha solo per le pagine *condivise* tra A e B, *ma senza imporre vincoli a priori* sul risultato di tali funzioni, purché, a tempo di creazione e di esecuzione, si garantisca che tale risultato sia uguale (compito affidato alla funzionalità di gestione della memoria principale, facendo uso di opportune annotazioni presenti nel file di configurazione di ogni processo).

Domanda 3

a) La struttura di ogni coda pronti (condivisa) è realizzata come lista "linkata" bidirezionale, come mostrato in figura:



I **puntatori condivisi** sono realizzati mediante due parole:

- la prima implementa un booleano "ultimo" che, se uguale a vero, sta a significare fine lista;
- la seconda contiene un *identificatore unico* (nome, di tipo intero) di processo, e precisamente il nome del processo il cui PCB è puntato.

Per i *puntatori condivisi* viene usata la tecnica degli *indirizzi logici distinti:* ogni processo A ha, nella propria MV, una tabella (Tabella dei Processi) che, per ogni identificatore di processo B, contiene l'indirizzo logico del PCB$_B$ nello spazio di indirizzamento di A.

Il PCB di ogni processo contiene, tra le altre, le seguenti informazioni utili per il nostro scopo:

- posizione L0: classe (da 0 a 3)
- posizione L1: indirizzo base della *Tabella dei Processi*;
- posizione L2: indirizzo base della *Tabella Interruzioni*, contenente, per ogni codice di evento, l'indirizzo della procedura Handler relativa a quell'evento;
- posizione L3+0: indirizzo della locazione condivisa Coda per la classe 0;
- …
- posizione L3+3: indirizzo della locazione condivisa Coda per la classe 3;
- posizione L4: puntatore al PCB successivo nella coda della propria classe;
- posizione L5: puntatore al PCB precedente nella coda della propria classe;

Si assume che le costanti L0, L1, L2, L3, L4, L5 abbiano valore compreso tra 0 e 31.

L'indirizzo del PCB del processo in esecuzione è contenuto in un registro generale *Rpcb*.

Alla fine della fase firmware del trattamento interruzioni, il processo in esecuzione ha ricevuto nei registri *Revento* e *Rnome* la doppia parola del messaggio di interruzione, ed ha chiamato la Routine Interfacciamento Interruzioni con indirizzo di ritorno in *Rret.*

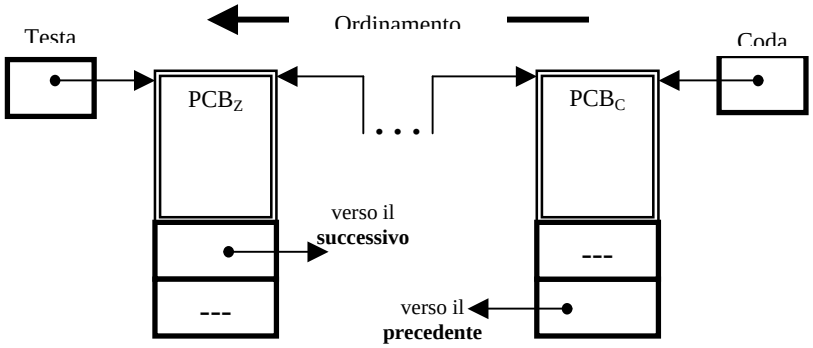
La procedura Handler_Sveglia viene chiamata dalla Routine Interfacciamento Interruzioni, nel modo seguente:

```
LOAD   Rpcb, L2, Rtemp, DI
LOAD   Rtemp, Revento, Rhandler
CALL   Rhandler, Rret1
GOTO   Rret, EI
```

Tutta la fase assembler del trattamento interruzioni è così eseguita a interruzioni disabilitate.

Indichiamo con **A** il processo in esecuzione, **B** il processo da svegliare, **C** il processo ultimo in coda nella lista della stessa classe di B.

L’implementazione è mostrata a pagina seguente.



Handler_Sveglia ha il seguente comportamento:

1. dalla Tabella dei Processi (PCB$_A$, L1) e da RG[Rnome] ricava l'indirizzo del PCB$_B$ del processo da svegliare;
2. da PCB$_B$ ricava la classe di B (L0): sia la classe *i*;
3. da PCB$_A$ (campo L3+*i*) ricava l'indirizzo della locazione Coda per la classe di B;
4. salva il nome C attualmente presente nella locazione Coda;
5. scrivi nella locazione Coda il nome di B (RG[Rnome]);
6. nel PCB$_B$, campo L4 ("successivo"), scrivi "fine lista";
7. nel PCB$_B$, campo L5 ("precedente"), scrivi il nome C salvato al passo 4;
8. dalla Tabella dei Processi (PCB$_A$, L1) e dal nome C ricava l'indirizzo del PCB$_C$ e scrivi il nome B (RG[Rnome]) nel campo "successivo" (L4) del PCB$_C$;
9. ritorna da procedura alla Routine Interfacciamento Interruzioni.

La sua implementazione in assembler è la seguente:

```
1.  LOAD   Rpcb, L1, Rtab_proc              / letto dal PCB_A l'indirizzo della Tabella dei Processi /
    LOAD   Rtab_proc, Rnome, RpcbB          / letto dalla Tabella dei Processi l'indirizzo del PCB_B /

2.  LOAD   RpcbB, L0, Rclasse               / letto da PCB_B il valore della classe di B /

3.  ADD    L3, Rclasse, RL3                 / calcolato l'indice della locazione Coda per la classe di B /
    LOAD   RpcB, RL3, RCoda                 / letto l'indirizzo della locazione Coda per la classe di B /

4.  LOAD   RCoda, 0, RC_ultimo              / salvata in RC_ultimo la prima parola del puntatore a PCB_C /
    LOAD   RCoda, 1, RC_nome                / salvata in RC_nome la seconda parola del puntatore a PCB_C /

5.  STORE  Rcoda, 0, 0                      / nella prima parole di Coda scrivi "non è l'ultimo" /
    STORE  Rcoda, 1, Rnome                  / ora la locazione Coda contiene il nome di B /

6.  STORE  RpcbB, L4, 1                     / scrivi "fine lista" nel campo del PCB_B che punta al successivo /

7.  STORE  RpcbB, L5, RC_ultimo             / scrivi nel campo precedente del PCB_B il puntatore C: parola 1. /
    ADD    L5, 1, RL5_seconda               / indice alla seconda parola del puntatore nel campo L5 /
    STORE  RcbB, RL5_seconda, RC_nome       / scrivi nel campo precedente del PCB_B il puntatore C: parola 2 /

8.  LOAD   Rtab_proc, RC_nome, RpcbC        / letto dalla Tabella dei Processi l'indirizzo del PCB_C /
    STORE  RpcbC, L4, 0                     / scrivi nel campo al successivo di C che non è più l'ultimo /
    ADD    L4, 1, RL4_seconda               / indice alla seconda parola del puntatore nel campo L4 /
    STORE  RpcbC, RL4_seconda, Rnome        / scritto il nome di B nel campo puntatore di C al successivo /

9.  GOTO   Rret1                            / ritorna alla Routine Interfacciamento Interruzioni /
```

b) La fase firmware del trattamento interruzioni è descritta dal seguente microprogramma:

tratt_int. reset INT, set ACKINT, int1

int1. (RDYIN, or (ESITO), = 0-) nop, int1;

(= 11) …, tratt_ecc ;

(= 10) reset RDYIN, set ACKIN, DATAIN → RG [Revento], int2

// ricevuta la seconda parola del messaggio di interruzione; viene passata alla Routine Interfacciamento Interruzione via Revento //

int2. (RDYIN, or (ESITO), = 0-) nop, int2;

(= 11) …, tratt_ecc ;

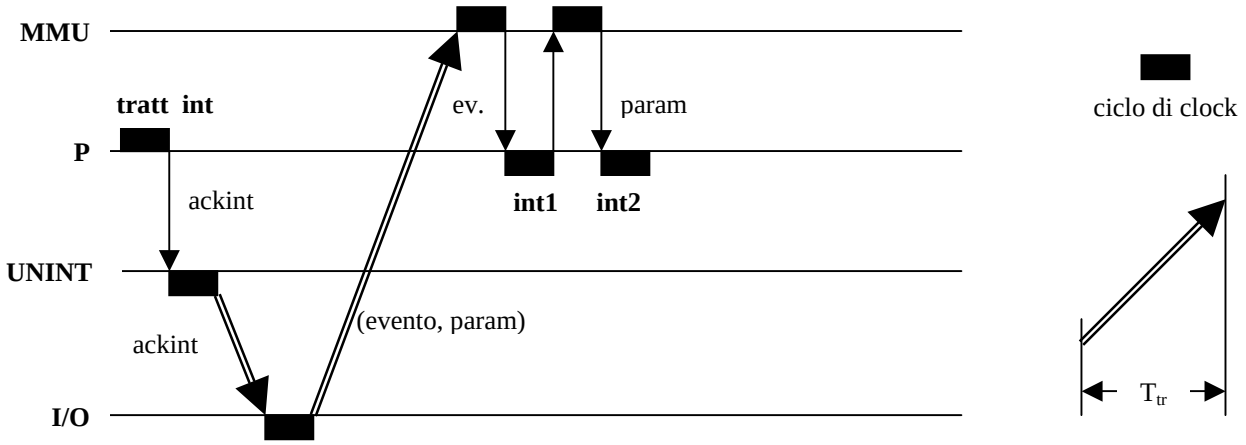
(= 10) reset RDYIN, set ACKIN, DATAIN → RG [Rparam], RG [Rroutine] → IC, IC → RG [Rret], ch0

// ricevuta la seconda parola del messaggio di interruzione; viene passata alla Routine Interfacciamento Interruzione via Rparam, viene chiamata tale Routine e viene salvato l'indirizzo di ritorno //

Il tempo di elaborazione di questa fase è dato da:

$$T_{int\text{-}FW} = 3\,\tau + T_{I/O}$$

dove $T_{I/O}$ è la latenza delle comunicazioni che hanno luogo nel percorso da P, a UNINT, all'unità di I/O, alla MMU via Bus I/O e, per due parole, da MMU a P:



Supponendo che le unità di I/O abbiano lo stesso ciclo di clock della CPU, e che tutti i collegamenti inter-chip abbiano la stessa latenza di trasmissione T_{tr}, si ha:

$$T_{I/O} = 4\,\tau + 2\,T_{tr}$$

da cui:

$$T_{int\text{-}FW} = 7\,\tau + 2\,T_{tr}$$

Assumendo ancora $T_{tr} = 20\,\tau$:

$$T_{int\text{-}FW} = 47\,\tau$$

Per la fase assembler occorre valutare l'uso della cache.

Complessivamente, considerando sia la Routine Interfacciamento Interruzioni che Handler, il tempo di completamento ideale è dato da ($t_c = 3\tau$):

$$T_{int\text{-}Assembler\text{-}id} = 22\,T_{ch} + 16\,T_{ex\text{-}LD/ST} + 3\,T_{ex\text{-}ADD} + 3\,T_{ex\text{-}GOTO/CALL} = 22\,(2\,\tau + t_c) + 16\,(2\,\tau + t_c) + 3\,\tau + 3\,\tau =$$

$$= 196\,\tau$$

Si può supporre che, con alta probabilità:

- tutte le istruzioni siano già in cache;
- di tutte le strutture dati utilizzate, il PCB del processo in esecuzione (PCB_A) e le strutture da esso puntate siano già in cache, eccetto i puntatori di Coda di qualcuna delle quattro liste pronti,.

Per tutti questi oggetti (che sono comunque in numero limitato) viene sfruttato il riuso imponendo che non vengano deallocati.

I fault di cache riguardano invece, con alta probabilità, i riferimenti a blocchi di oggetti come PCB_B e PCB_C e la locazione Coda (per quest'ultima si è considerato un caso sfavorevole). Supponendo che i campi L0, L4, L5 del PCB siano contigui, il numero di fault è valutabile come

$$N_{fault} = 3$$

Quindi, con i dati della Domanda 1:

$$T_{int\text{-}Assembler} = T_{int\text{-}Assembler\text{-}id} + 3 * T_{trasf} = 196\ \tau + 3 * 124\ \tau = 568\ \tau$$

Complessivamente:

$$T_{int} = T_{int\text{-}FW} + T_{int\text{-}Assembler} = 615\ \tau$$

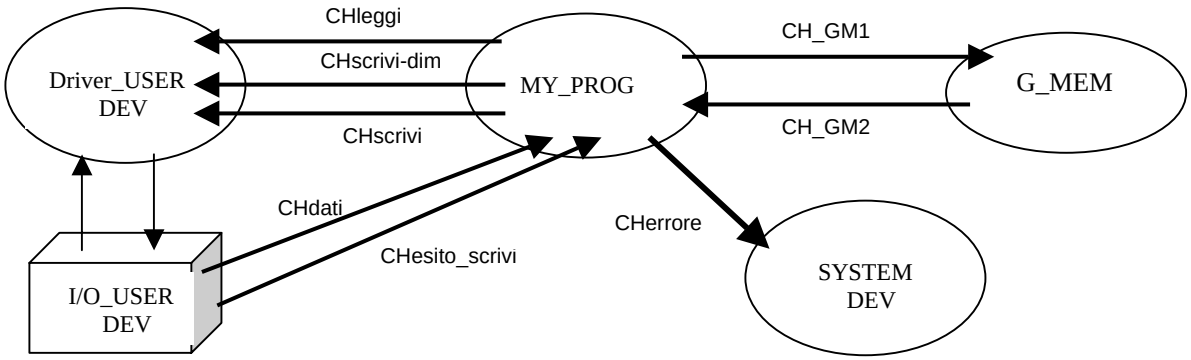
Domanda 4

a) Il programma applicativo sorgente è il seguente:

```
int A[N], B[N], C[N]; < altre dichiarazioni >
{   leggi (4*N, A);
    leggi (4*N, B);
    leggi (4*N, C);
            < corpo del programma >
    scrivi (4*N, C)
}
```

Il processo Driver_USER_DEV dispone di tre canali *asimmetrici* d'ingresso, di identificatori *CHleggi*, *CHscrivi-dim e CHscrivi*, corrispondenti alle due operazioni da esso implementate per virtualizzare USER_DEV: attraverso *CHleggi* riceve la dimensione del blocco di dati e l'identificatore del canale *CHdati* su cui l'applicazione si aspetta di ricevere il blocco di dati; attraverso *CHscrivi-dim* riceve la dimensione del blocco di dati, e attraverso *CHscrivi* il valore del blocco dei dati.

Lo schema a processi comunicanti della computazione è mostrato nella figura seguente:



Per minimizzare il tempo di comunicazione dei valori degli array *A*, *B* e *C*, il canale *CHdati* ha come mittente direttamente il *processo esterno* I/O_USER_DEV. A questo scopo, il valore dell'identificatore *CHdati*, che Driver_USER_DEV ha ricevuto in una variabile di tipo *channelname*, viene inoltrato da Driver_USER_DEV ad I/O_USER_DEV con il messaggio di richiesta di lettura del blocco.

Oltre ai canali di comunicazione con Driver_USER_DEV, I/O_USER_DEV e SYSTEM_DEV, MY_PROG utilizza i canali per il *trattamento dell'eccezione di fault di pagina* (*CH_GM1, CH_GM2*), mediante i quali comunica al processo gestore della memoria principale i parametri relativi ad una situazione di fault di pagina, ed attende la segnalazione che la pagina richiesta sia stata allocata e trasferita in M e che la Tabella di Rilocazione sia stata aggiornata.

Tutti i canali di ingresso di MY_PROG sono *asincroni*, per assicurare che i mittenti delle comunicazioni su tali canali si sospendano il meno possibile nell'esecuzione delle *send* relative. Pur non essendo visibile in MY_PROG, anche i canali di ingresso degli altri processi conviene che siano asincroni con grado di asincronia opportunamente alto.

A questo riguardo, va notato che, nel formalismo a scambio di messaggi adottato, i canali di comunicazione sono *tipati* affinché il compilatore possa effettuare controlli forti e affinché tutte le strutture dati del supporto siano allocate staticamente nelle memorie virtuali dei processi. Nel nostro caso, questo implica che i canali su cui trasmettere blocchi di dati devono essere dimensionati in modo costante. Ad esempio, supponiamo che sul canale *CHdati* possano essere trasmessi messaggi di 128K parole; quindi, ognuno degli array A, B, C deve essere trasmesso mediante una sequenza di 8 messaggi da 128K ciascuno. Questo vale anche per il canale *CHscrivi*; in questo caso, però, occorre anche un ulteriore canale, *CHscrivi-dim*, per comunicare preventivamente la dimensione del messaggio.

La compilazione nel linguaggio $\mathcal{L}_c$ è mostrata nel seguito. La notazione **block(A, i, j)** sta a significare l'*i*-esimo blocco dell'array A di dimensione *j* interi. Il comando **end** può essere inserito in qualunque punto del processo, provocandone la terminazione.

MY_PROG:: **channel in** CHdati (8), CHesito_scrivi (1), CH_GM2 (1); **channel out** CHleggi, CHscrivi, Chscrivi-dim, CHerrore, CH_GM1; <altre dichiarazioni>;

```
{   send ( CHleggi, (CHdati, 4*N) );
    for (i = 0; i < N; i++)
        { receive ( CHdati, ( esito, block (A, i, N/8) );
          if not (esito)  then { send (CHerrore, "errore"); end }
        };

    send ( CHleggi, (CHdati, 4*N) );
    for (i = 0; i < N; i++)
        { receive ( CHdati, ( esito, block (B, i, N/8) );
          if not (esito)  then { send (CHerrore, "errore"); end }
    };

    send ( CHleggi, (CHdati, 4*N) );
    for (i = 0; i < N; i++)
        { receive ( CHdati, ( esito, block (C, i, N/8) );
          if not (esito)  then { send (CHerrore, "errore"); end }
        };

              < corpo del programma >;

    send ( CHscrivi-dim, 4*N );
    for (i = 0; i < N; i++)
        { send ( CHscrivi, (CHesito_scrivi, block (C, i, N/8) );
          receive (CHesito_scrivi, esito)
          if not (esito)  then { send (CHerrore, "errore"); end }
        };

     end
};
procedure
{   Handler per trattamento interruzioni;
    Handler per trattamento eccezioni (CH_GM1, CH_GM2)
}
```

La compilazione *in assembler* produce, oltre al codice visto nell'Esercitazione 2, il codice del supporto a tempo di esecuzione di $\mathcal{L}_c$: supporto delle primitive *send* e *receive*, incluso lo scheduling a basso livello, trattamento di interruzioni ed eccezioni. Il compilatore inizializza tutte le strutture dati del supporto di $\mathcal{L}_c$.

b) Il processo MY_PROG non può mai sospendersi sulla primitiva *send(CHleggi, …)*, a condizione che *CHleggi* sia asincrono (è sufficiente il grado di asincronia uguale a uno), visto che il colloquio per implementare il comando *leggi* è a domanda e risposta. Lo stesso vale per la *send(CH_GM1, …)* e per *send (CHscrivi-dim, …).*

MY_PROG può sospendersi (si sospende) sulle primitive *receive(CHdati, …), receive(CHesito_scrivi, …)* e *receive(CH_GM2, …)*, proprio in quanto il colloquio è a domanda e risposta.

MY_PROG può inoltre sospendersi sulla primitiva *send(CHscrivi, …),* se il grado di asincronia di *CHscrivi* è minore di 8, e MY_PROG tenta di inviare un numero di messaggi maggiore del grado di asincronia senza che il partner abbia ancora eseguito una *receive* corrispondente. Analogamente per *send(CHerrore, …),* se questo canale è sincrono.

MY_PROG verrà risvegliato dai partner delle comunicazioni, su cui si è sospeso, quando questi eseguiranno le primitive corrispondenti. Nel caso che il partner sia il processo esterno I/O_USER_DEV, la sveglia di MY_PROG viene effettuata *in seguito all'interruzione* che I/O_USER_DEV invia quando, *e se*, rileva (attraverso informazioni nel descrittore di canale) che MY_PROG si trova in attesa. Il processo attualmente in esecuzione sul processore, una volta completata la fase firmware del trattamento interruzioni, esegue l'Handler che consiste proprio nella procedura di sveglia processo, come visto nella Domanda 3.

c) Eliminando il processo Driver_USER_DEV, *non va apportata alcuna modifica* alla versione assembler di MY_PROG, in quanto valgono *entrambe* le seguenti condizioni:

1) l'unità di I/O è vista come un processo;

2) tutti i dettagli dell'implementazione dei trasferimenti di dati e della sincronizzazione tra processi, *inclusi i trasferimenti di I/O*, sono completamente trattati nel supporto delle primitive *send* e *receive*.

Ora, i canali *CHleggi* e *CHscrivi* saranno in ingresso al processo esterno I/O_ USER_DEV, senza che ciò sia minimamente visibile a MY_PROG: le primitive *send e receive* lavorano su descrittori di canale, senza alcuna conoscenza del nome del processo partner né tanto meno della sua implementazione. Quando si tratti di eseguire operazioni di "sveglia partner", queste riferiscono indirizzi di PCB presenti nel descrittore di canale.

d) Nel caso *c),* il modello dei trasferimenti di I/O (DMA o Memory Mapped I/O) è *completamente invisibile* nel codice di MY_PROG, in quanto valgono *entrambe* le seguenti condizioni:

1) tutti i dettagli dell'implementazione dei trasferimenti di dati e della sincronizzazione tra processi, *inclusi i trasferimenti di I/O*, sono completamente trattati nel supporto delle primitive *send* e *receive*;

2) le primitive *send* e *receive* utilizzano indirizzi logici, senza fare alcuna assunzione sul fatto che gli oggetti *condivisi* riferiti (descrittori di canali, messaggi, variabili targa, ecc.) siano allocati fisicamente in memoria principale o nelle memorie di I/O.