Architettura degli Elaboratori - Corso A

Seconda Prova di Verifica Intermedia, a.a. 2004-05, 22 dicembre 2004

Correzione

Queste note, oltre che documentare la soluzione della prova, hanno lo scopo di mostrare come organizzare un elaborato, ed in particolare come apportare le necessarie spiegazioni.

Domanda 1

Si consideri il programma assembler Risc ottenuto dalla compilazione di un programma sorgente che, dato un vettore A di 256K interi, modifica A in modo che ogni elemento sia sostituito dal suo quadrato.

a) Con riferimento a tale programma, spiegare le proprietà di località e riuso.

b) Valutarne il tempo di completamento, il tempo medio di elaborazione, la performance e l'efficienza relativa per un calcolatore con le seguenti caratteristiche:

i) processore con ciclo di clock $\tau = 0{,}3$ nsec;

ii) cache primaria operante su domanda, con indirizzamento completamente associativo, blocchi di 8 parole, scritture con il metodo Write-Through;

iii) cache secondaria (con probabilità di fault trascurabile) organizzata con 4 moduli interallacciati collegati direttamente alla CPU, ciclo di clock di 10τ e latenza di trasmissione dei collegamenti inter-chip di 10τ.

Spiegare esplicitamente come si è tenuto conto di ognuna delle caratteristiche del punto *ii)*.

c) Spiegare, in generale, le motivazioni dell'inserimento della cache secondaria.

Domanda 2

Si consideri un calcolatore avente sistema operativo a processi cooperanti secondo il modello a scambio di messaggi. L'architettura include 16 unità di I/O. Il massimo numero di processi nel sistema è 2000. Non viene adottata alcuna forma di prerilascio del processore. L'allocazione dinamica della memoria principale è effettuata con il metodo della paginazione.

a) Il programma applicativo di cui alla Domanda 1 è modificato nel seguente modo: all'inizio l'array A viene letto da un certo dispositivo D1 e alla fine viene scritto su un certo dispositivo D2. D1 è un dispositivo solo d'ingresso, D2 solo di uscita.

Si suppone che il linguaggio sorgente dell'applicazione metta a disposizione le seguenti operazioni per operare su D1 e D2 rispettivamente:

- *get* (n, v) : richiesta a D1 di inviare *n* byte che verranno assegnati alla variabile *v*;
- *put* (n, v) : invio a D2 del valore di una variabile *v* di *n* byte.

Compilare il programma in un processo, detto QUAD, e mostrare lo schema della computazione a processi comunicanti comprendente QUAD.

I dispositivi D1 e D2 sono interfacciati attraverso i rispettivi gestori Driver_D1 e Driver_D2.

Non sono previste informazioni sull'esito delle operazioni richieste ai dispositivi.

b) Spiegare in seguito a quali eventi il processo QUAD può subire transizioni di stato di avanzamento, e dare una sintetica descrizione a parole delle azioni di scheduling a basso livello associate a tali transizioni.

c) Mostrare e spiegare la memoria virtuale del processo QUAD, indicando quali oggetti, o parti di oggetti, sono inizializzati a tempo di compilazione.

d) Spiegare, in generale, perché l'inizializzazione di variabili, decisa a tempo di compilazione, ha proprio l'effetto desiderato.

Domanda 1

Il programma da compilare è:

```
int A[256K];
for (i = 0; i < N; i++)
        A[i] = A[i] * A[i];
```

Il compilatore provvede a inizializzare RG[RA] all'indirizzo logico base di A, RG[Ri] a zero, RG[RN] alla costante N = 256K, ed a riservare RG[Ra] come temporaneo non inizializzato. Adottando la regola di compilazione di un *for* con almeno una iterazione, il programma assembler richiesto è il seguente:

```
LOOP: LOAD  RA, Ri, Ra
      MUL  Ra, Ra, Ra
      STORE  RA, Ri, Ra
      INCR  Ri
      IF<  Ri, RN, LOOP
      END
```

a) Località e riuso

Queste proprietà possono essere spiegate osservando una *traccia di indirizzi logici generati dal programma* in esecuzione. Supponendo, ad esempio, che, nella memoria virtuale, il codice sia allocato a partire dall'indirizzo 0 e l'array A sia allocato a partire dall'indirizzo 1024, la seguente è la traccia della fase iniziale dell'esecuzione del programma (in assenza di interruzioni ed eccezioni):

0, 1024, *1, 2,* 1024, *3, 4, 0,* 1025, *1, 2,* 1025, *3, 4, 0,* 1026, *1, 2,* 1026, *3, 4, 0,* 1027, …

Si osserva che, all'interno di una finestra temporale sufficientemente ampia, si intrecciano più (due nel caso specifico) sequenze di indirizzi logici ([0, 1, 2, 3, 4] e [1024, 1024, 1205, 1025, 1026, 1026, 1027, …]) dove in ogni sequenza gli indirizzi sono tra loro relativamente vicini rispetto all'ampiezza della finestra temporale (consecutivi e/o coincidenti nel caso specifico): questa proprietà è detta *località* dei riferimenti. Inoltre, alcune sequenze si ripetono molte volte all'interno della finestra temporale (la prima nel caso specifico), oppure alcuni riferimenti si ripetono all'interno della stessa sequenza (come nel secondo caso), dando luogo a *riuso* delle informazioni.

Organizzando le informazioni a pagine (blocchi), risulta relativamente elevata la probabilità che, per un periodo di tempo relativamente lungo, le informazioni riferite appartengano ad un numero limitato di pagine e che le pagine riferite variino lentamente: nel caso specifico, la stessa pagina di codice è usata per tutta la durata del programma, mentre la stessa pagina di dati è usata per σ riferimenti consecutivi all'array A, con σ ampiezza della pagina.

Quanto ora detto vale qualitativamente *per qualunque* (sotto-)gerarchia di memoria. Passando da una sotto-gerarchia (ad esempio, Memoria Virtuale – Memoria Principale) ad un'altra (ad esempio, Memoria Principale – Memoria Cache), cambiano quantitativamente i valori di σ e della probabilità di fault *h* (ad esempio, $s \sim 10^3$ parole e $h \sim 10^{-4} \div 10^{-5}$ nel primo caso, $s \sim 10^0 \div 10^1$ parole e $h \sim 10^{-2}$ nel secondo caso).

b) Valutazione delle prestazioni

Essendo la cache operante su domanda, il trasferimento di un blocco dal supporto superiore della gerarchia viene effettuato ad ogni fault sequenzialmente all'elaborazione. Il tempo di completamento si valuta come:

$$T_c = T_{c-id} + T_{fault} = T_{c-id} + N_{fault} * T_{trasf}$$

dove

- $T_{c\text{-}id}$ è il tempo di completamento ideale, in assenza di fault. Esso è quindi valutando assumendo che il tempo di accesso alla memoria sia uguale al tempo di accesso alla cache primaria; nel nostro caso, essendo usato il metodo completamente associativo, $t_c = 3\tau$;
- N_{fault} è il numero di fault che si verificano durante l'esecuzione del programma. Le istruzioni provocano un solo fault iniziale, dopo di che, grazie alla proprietà del riuso, esse sono sempre presenti in un blocco di cache; si può dunque trascurare la probabilità di fault delle istruzioni. Per i dati (elementi dell'array A), si verifica un fault ogni σ iterazioni, quindi

$$N_{fault} = \frac{N}{\sigma} = \frac{N}{8}$$

- T_{trasf} è il tempo per trasferire un blocco dalla cache secondaria C2 alla cache primaria C1. Essendo C2 interallacciata con $m = 4$ moduli, occorrono due letture da C2 per un blocco di $\sigma = 8$ parole, delle quali la seconda lettura si sovrappone alla scrittura in C1 delle 4 parole ottenute con la prima lettura:

$$T_{trasf} = \frac{\sigma}{m} t_{C2} + m\tau = 2t_{C2} + 4\tau = 2(\tau_{C2} + 2T_{tr}) + 4\tau = 64\tau$$

Quindi:

$$T_{fault} = N_{fault} * T_{trasf} = 8N\tau$$

Si osservi che l'ipotesi delle scritture effettuate con il metodo *Write-Through* permette, in generale, di non considerare i casi in cui si vada a sostituire un blocco di A precedentemente modificato. Con il metodo Write-Back, invece, occorrerebbe considerare che, con una certa probabilità (non semplice da stimare in generale, ma semplice nell'esempio specifico: il 100% dei casi), il trasferimento di un blocco richiede un tempo doppio rispetto a quello calcolato sopra (riscrittura del blocco sostituito, lettura del blocco richiesto). Affinché il metodo Write-Through non provochi degradazione di prestazioni, occorre verificare che la scrittura in C2 sia effettivamente sovrapposta al calcolo nella CPU fino all'esecuzione della prossima istruzione di STORE: nel nostro caso che sia sovrapposta all'elaborazione di una iterazione (vedi in seguito).

Con i simboli noti per i tempi medi di chiamata istruzione e decodifica (T_{ch}) e della fase di esecuzione (T_{ex}), e ricordando le prestazioni dell'interprete microprogrammato del linguaggio assembler Risc, il tempo di completamento ideale è dato da:

$$\begin{aligned} T_{c-id} &= N\ [5T_{ch} + 2T_{ex-LD/ST} + T_{ex-MUL} + T_{ex-INCR} + T_{ex-IF}\ ] = \\ &= N\ [5(2\tau + t_c) + 2(2\tau + t_c) + 50\tau + 1\tau + 2\tau\ ] = 88N\tau \end{aligned}$$

Quindi:

$$T_c = T_{c-id} + T_{fault} = 96N\tau = 7{,}55 \text{ msec}$$

Una singola iterazione è eseguita in 96τ, quindi la scrittura nella cache secondaria (che richiede un tempo $t_{C2} = 30\tau$) è mascherata completamente.

L'efficienza relativa della CPU con cache è data da:

$$\varepsilon = \frac{T_{c-id}}{T_c} = \frac{88}{96} = 0{,}92$$

che è un buon risultato (cache sfruttata al 92% delle proprie potenzialità), dovuto anche al fatto che il programma è "rallentato" dall'istruzione MUL.

Essendo $5N$ il numero di istruzioni complessivamente eseguite, il tempo medio di elaborazione per istruzione è dato da:

$$T = \frac{T_c}{5N} = \frac{96}{5}\tau = 19{,}2\tau = 5{,}76 \text{ nsec}$$

La performance vale:

$$\wp = \frac{1}{T} = 173{,}6\,MIPS$$

c) Motivazioni dell'inserimento della cache secondaria

Nonostante si ricorra ad una organizzazione interallacciata, essendo la probabilità di fault dell'ordine di 10^{-2} nel caso più favorevole, l'efficienza relativa sarebbe ugualmente piuttosto bassa se il supporto di memoria immediatamente superiore alla cache (primaria) fosse direttamente la memoria principale, a causa del tempo di accesso, e quindi del valore di T_{trasf}, troppo elevato.

Ad esempio, nel nostro caso, conservando $m = 4$, ma passando da $t_{C2} = 30\tau$ a $t_M \sim 300\tau$ (anche a causa dell'aumento della latenza dei collegamenti), avremmo $T_{fault} = 75{,}5N\tau$, $T_c = 163{,}5N\tau$, $\varepsilon = 0{,}54$.

Domanda 2

Con il modello di sistema operativo indicato nel testo, la compilazione di qualunque programma applicativo deve realizzare, mediante comunicazioni attraverso canali predefiniti con i processi di sistema, l'aggancio delle funzionalità di gestione delle risorse al processo applicativo.

a) Compilazione a processi comunicanti

Il programma applicativo sorgente è il seguente:

```
int A[256K];
{       get (1024K, A);
        for (i = 0; i < N; i++)
                A[i] = A[i] * A[i];
        put (1024K, A)
}
```

Il processo Driver_D1 dispone di un canale d'ingresso CH_D11 sul quale riceve la dimensione del blocco di dati (è implicita la richiesta di inviare un blocco di dati di tale dimensione), ed un canale di uscita CH_D12 sul quale invia il blocco di valori che viene richiesto. Il processo Driver_D2 dispone di un canale di ingresso CH_D2 sul quale riceve la dimensione del blocco dei dati che gli vengono inviati ed il valore di tali dati.

La compilazione del processo QUAD nel linguaggio concorrente di sistema a scambio di messaggi è:

```
{   send (CH_D11, 1024K );
    receive (CH_D12, A );

    for (i = 0; i < N; i++)
            A[i] = A[i] * A[i];

    send (CH_D2, (1024K, A) );

    procedure per trattamento interruzioni (CH_I/O_0, …, CH_I/O_15);
    procedure per trattamento eccezioni (CH_G_MEM_1, CH_G_MEM_2)
}
```

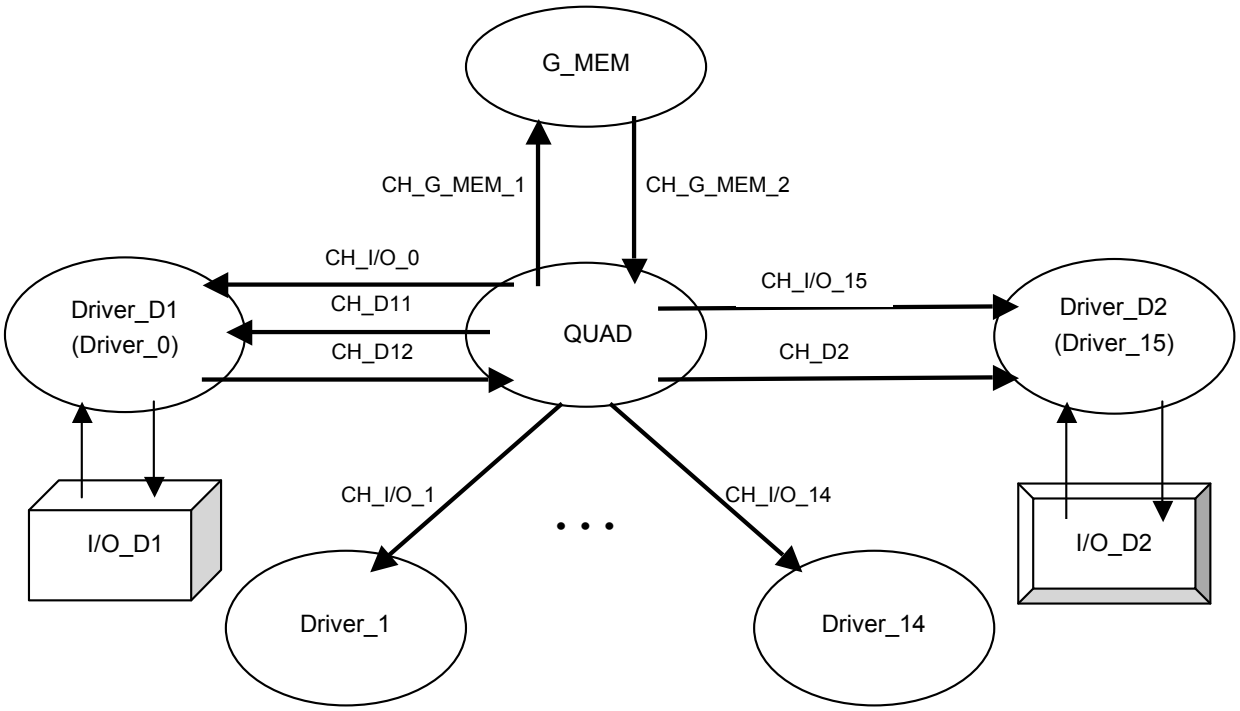
Aggancio alla gestione di D1 per ricevere 256K parole

Codice dell'applicazione isolata

Aggancio alla gestione di D2 per inviare 256K parole

La compilazione in assembler produce, oltre al codice visto nella Domanda 1, il codice del nucleo, corrispondente alle primitive *send* e *receive* (chiamate di procedura oppure codice espanso in loco), incluso lo scheduling a basso livello, ed al trattamento di interruzioni ed eccezioni, oltre ad inizializzate tutte le strutture dati riferite dal nucleo stesso.

Lo schema della computazione a processi comunicanti, relativamente alla parte che interessa QUAD, è mostrato nella figura seguente:



Oltre ai canali di comunicazione con Driver_D1 e Driver_D2, utilizzati per effettuare l'aggancio alle funzionalità di gestione delle risorse richieste esplicitamente dall'applicazione, QUAD utilizza:

- canali per il *trattamento delle interruzioni* (CH_I/O_0, …, CH_I/O_15), mediante i quali virtualizza l'invio dei messaggi che ogni unità di I/O intende inviare al rispettivo processo Driver e che vengono trasmessi al processore all'accettazione delle interruzioni (a meno che l'unità di I/O non sia capace di eseguire essa stessa le primitive di comunicazione). Si è supposto che D1 sia associato all'unità I/O_0 e D2 ad I/O_{15};
- canali per il *trattamento dell'eccezione di fault di pagina* (CH_G_MEM_1, CH_G_MEM_2), 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 memoria principale e che la Tabella di Rilocazione sia stata aggiornata.

b) Transizioni di stato di avanzamento

Non essendo prevista alcuna forma di prerilascio, non si applica la gestione della CPU a quanti di tempo, ed i processi hanno tutti la stessa priorità; la coda pronti è quindi FIFO. Le possibili transizioni di stato di avanzamento sono quindi da Pronto a Esecuzione, da Esecuzione ad Attesa, e da Attesa a Pronto.

Una volta che QUAD è stato *creato* (e quindi portato per la prima volta in stato di Pronto), gli unici eventi che possono provocare tali transizioni di stato di avanzamento sono legati all'esecuzione di primitive *send* e *receive* da parte di QUAD stesso o di altri processi o alla terminazione di processi:

- *transizione da Pronto a Esecuzione*: quando il processore viene liberato da un altro processo (passaggio in Attesa o terminazione) e QUAD si trova in testa alla Lista Pronti. Il processo che libera il processore effettua la commutazione di contesto (vedi il punto successivo per la spiegazione);

- *transizione da Esecuzione ad Attesa*: quando QUAD tenta di eseguire una *send* su un canale (CH_a) che già contiene un numero di messaggi uguale al grado di asincronia, oppure una *receive* su un canale (CH_b) che non contiene alcun messaggio.

  Canali come CH_a (sui quali QUAD si può sospendere con una *send*) non possono essere CH_D11, CH_2 e CH_G_MEM_1, se vengono realizzati in modo asincrono (è sufficiente grado di asincronia uguale a uno), mentre possono esserlo CH_I/O_0, …, CH_I/O_15. Canali come CH_b (sui quali QUAD si può sospendere con una *receive*) sono tutti i canali di ingresso di QUAD.

  Quando passa in Attesa, QUAD esegue la *commutazione di contesto*: salva il proprio stato interno (RG, IC) nel proprio PCB, stacca il primo PCB dalla Lista Pronti, ripristina in RG e IC i valori contenuti in tale PCB, ed esegue come ultima istruzione la *START_PROCESS;*

- *transizione da Attesa a Pronto*: quando si verifica l'evento atteso (ricezione di un messaggio su CH_a, o invio di un messaggio su CH_b) in seguito all'esecuzione della primitive corrispondente da parte del processo partner. È questo processo a inserire il PCB di QUAD in fondo alla Lista Pronti.

La Lista Pronti è definita da due variabili condivise contenenti rispettivamente il puntatore di testa e quello di coda. Ogni processo reperisce gli indirizzi di tali variabili in posizioni note del proprio PCB. Inoltre, per permettere la strutturazione a lista "linkata", ogni PCB contiene il puntatore all'elemento successivo ed il puntatore all'elemento precedente della coda FIFO. Tutti i puntatori contenuti in variabili condivise hanno lo stesso indirizzo logico nello spazio logico di tutti i processi.

c) Memoria virtuale

Nella seguente tabella sono indicati gli oggetti che fanno parte della memoria virtuale di QUAD e se/come sono eventualmente inizializzati a compilazione:

Codice visibile al programmatore (for … A[i] = …)	Completamente inizializzato
Dati visibili al programmatore (array A)	Non inizializzato
Codice di send e receive	Completamente inizializzato
Codice dello scheduling a basso livello	Completamente inizializzato
Codice del trattamento interruzioni	Completamente inizializzato
Tabella degli indirizzi degli Handler Interruzioni	Completamente inizializzato
Codice del trattamento eccezioni	Completamente inizializzato
Tabella degli indirizzi degli Handler Eccezioni	Completamente inizializzato
Canale di comunicazione CH_D11	Inizializzazione di indicatori di attesa, variabili per il controllo della coda di messaggi, tipo e lunghezza del messaggio, indirizzi dei PCB dei processi partner (possibile anche la soluzione con inserimento dinamico del PCB di un processo solo quando questo si sospende)
Canale di comunicazione CH_D12	Come sopra
Canale di comunicazione CH_D2	Come sopra
Canale di comunicazione CH_G_MEM_1	Come sopra
Canale di comunicazione CH_G_MEM_2	Come sopra
Canale di comunicazione CH_I/O_0	Come sopra
… …	… …
Canale di comunicazione CH_I/O_15	Come sopra

Tabella di Rilocazione	Tutte le entrate inizializzate a "pagina non allocata" (bit di presenza pagina uguale a zero); bit di modifica e contatore LRU inizializzati a zero; inizializzati i campi diritti di accesso e limite di pagina.
PCB_QUAD	Inizializzati il valore di IC (0), le immagine di alcuni registri generali (RA, Ri, RN, Rpcb, ed altri eventualmente contenenti parametri delle funzionalità di nucleo, come indirizzi di routine handler e indirizzi base di tabelle handler), alcuni parametri delle funzionalità di nucleo (se non contenuti in registri), puntatore alla Tabella di Rilocazione, indirizzi delle locazioni che contengono i puntatori di testa e di coda della Lista Pronti.
Puntatori di testa e di coda della Lista Pronti	Non inizializzato
PCB_0	Non inizializzato
… …	… …
PCB_1999	Non inizializzato

Per la protezione delle informazioni, ogni campo occupa una pagina o un multiplo intero di una pagina.

d) Significato dell'inizializzazione di variabili a tempo di compilazione

L'immagine della memoria virtuale di un processo, come risultante dalla compilazione, rappresenta il contenuto di un file (file oggetto) mantenuto in una memoria permanente.

Quando il processo viene creato, una parte di tale file ("Working Set" del processo contenente tipicamente il PCB del processo stesso, codice e dati del nucleo, parte del codice e dei dati visibili al programmatore) viene trasferito in memoria principale. Di conseguenza, tutte le variabili le, che erano state inizializzate a tempo di compilazione in memoria virtuale, lo sono anche a tempo di esecuzione in memoria principale esattamente con gli stessi valori. Lo stesso dicasi per le variabili in registri generali ed il contatore istruzioni, i quali assumono i valori corretti quando il processo entra nello stato di Esecuzione a seguito di una commutazione di contesto.

Analogamente, ogni volta che una pagina del processo viene trasferita da memoria secondaria in memoria principale (in seguito a fault di tale pagina), ha luogo automaticamente l'inizializzazione delle variabili in memoria principale con i valori presenti in memoria secondaria. Ciò può avvenire anche più volte per la stessa pagina; a questo scopo, ogni volta che una pagina modificata viene scaricata dalla memoria principale, essa viene riscritta in memoria secondaria.