Corso di Laurea in Informatica, A.A. 2022/2023

Architetture degli Elaboratori e Sistemi Operativi (AESO corso A e B)

Secondo Appello – 9 giugno 2023

Scrivere in modo comprensibile e chiaro rispondendo alle domande/esercizi riportati in questo foglio. Sviluppare le soluzioni dei primi due esercizi (prima parte) in un foglio separato rispetto alla soluzione degli ultimi due esercizi (seconda parte), in modo da facilitare la correzione da parte dei docenti. Riportare su tutti i fogli consegnati nome, cognome, numero di matricola e corso (A o B).

Parte 1

Esercizio 1

Si fornisca il codice assembler ARMv7 della procedura int compute_row(int *r, int n) che restituisce il minimo di un vettore di interi di n>0 posizioni il cui indirizzo è contenuto nella variabile r. La compute_row non deve utilizzare altri registri che quelli temporanei. Un possibile pseudocodice della funzione è il seguente:

```
int compute_row(int *r, int n) {
 poni min = primo elemento del vettore;
 per tutti gli altri elementi del vettore:
       se minore del min corrente assegna a min il valore dell'elemento
 restituisci min;
}
```

Esercizio 2

Si consideri il codice assembler ARMv7 di seguito (ogni istruzione è marcata con un identificatore univoco progressivo)

1. LDR R0, [R1, #4]
2. ADDS R3, R0, R2
3. SUB R2, R2, R4
4. LDR R5, [R2]
5. BEQ cont
6. STR R7, [R3]

Si indichino quali dipendenze logiche (dati e di controllo) sono presenti nel frammento di codice di cui sopra. Si valuti il CPI medio nei seguenti scenari: *a)* considerando l'esecuzione sia sulla micro-architettura pipeline ottimizzata (con canali di forwarding e anticipazione di salto) che sulla micro-architettura pipeline "pura" (in cui non ci sono canali di forwarding e nessuna anticipazione di salto, e in cui le dipendenze possono essere risolte soltanto tramite l'introduzione di stalli/NOP); *b)* in ambo i casi precedenti, considerare le situazioni in cui l'istruzione 5 di BEQ sia un salto preso o meno.

Parte 2

Esercizio 3

a) Si consideri uno schedulatore in cui un processo possa trovarsi solo in stato di esecuzione (EX), di pronto (RDY) o di attesa (W). Si indichino brevemente quali possono essere i motivi che determinano una transizione dallo stato di esecuzione (EX) in ciascuno degli altri due stati (RDY e W) e viceversa.

b) Quattro processi A, B, C e D vengono mandati in esecuzione uno dopo l'altro a distanza di 0.5s l'uno dall'altro. I processi A, B e C hanno priorità uguale e più bassa di quella del processo D. La durata necessaria per l'esecuzione dei 4 processi risulta essere: 1s (A), 0.75s (B), 1s (C) e 2s (D). Nessuno dei quattro processi si blocca durante la sua esecuzione, se non per decisione dello schedulatore. Lo schedulatore lavora a quanti di tempo di durata 0.25s e con prerilascio. Il tempo impiegato per la commutazione di contesto si considera trascurabile. Infine, se allo scadere del quanto di tempo arrivano nuovi processi di uguale priorità, il processo in esecuzione viene messo in fondo alla coda pronti. Si indichi il tempo di permanenza nel sistema di ogni processo ed il tempo medio di permanenza dei processi.

Esercizio 4

a) Si descriva brevemente cos'è un semaforo, quali operazioni supporta e quale sia la semantica implementata dalle diverse operazioni.

b) N thread si comportano come produttori utilizzando un buffer in grado di contenere un solo messaggio. Un ulteriore thread si comporta da consumatore dei messaggi presenti nel buffer. Si fornisca lo pseudo codice C del metodo put(msg) usato del generico thread produttore e del metodo get(&msg) usato dal thread consumatore considerando in alternativa i seguenti casi (tra parentesi sono indicati i metodi che possono essere utilizzati per la sincronizzazione):

1. utilizzando solamente i semafori (P, V)
2. utilizzando una variabile di spin-lock (SPINLOCK (*), UNLOCK)
3. utilizzando *una sola* variabile di condizione (LOCK, UNLOCK, COND_WAIT, COND_SIGNAL/BROADCAST)

(*) SPINLOCK ritorna *true* se la lock è stata acquisita, *false* altrimenti. Per tutte le altre funzioni si assuma che vengano eseguite sempre con successo.

SOLUZIONE

Eserizio 1

```
            .text
            .global compute_row
            .type compute_row,%function
compute_row: mov r3, r0         @ sposto indirizzo in R3
            ldr r0, [r3], #4    @ min = r[0]
       loop: subs r1, r1, #1    @ decremento N
            beq end             @ se diventa 0 ho finito
            ldr r2, [r3], #4    @ carico r[i]
            cmp r2, r0      @ confronto con min
            movlt r0, r2    @ in caso lo metto al posto di min
            b loop          @ e vado alla iterazione successiva
        end: mov pc, lr      @ il risultato e' gia' in r0
```

Esercizio 2

Ci sono dipendenze RAW tra le istruzioni 1->2, 2->6 e 3->4. La 2->6 non conta, dal momento che le due istruzioni sono a distanza 4. Come sappiamo nella microarchitettura pipeline qualsiasi dipendenza di distanza superiore a due è risolta automaticamente. Per esempio in questo caso quando la 6 viene decodificata, la 2 è completata e quindi il valore di R3 modificato dalla ADDS può essere letto dalla STR.

La dipendenza 3->4 è di distanza 1 indotta da una istruzione operativa (SUB). Nel pipeline ottimizzato si risolve con forwarding dallo stadio di Memoria a quello di Execute senza degradazioni. Nel caso di pipeline puro (senza canali di forwarding), bisognerebbe mettere in stallo il pipeline per due cicli (equivalenti a due NOP che aumenterebbero la distanza a 3), in modo che quando la LDR è in fase di Decode la SUB sia in fase di WB. Ricordiamo che il Register File della microarchitettura pipeline consente di scrivere un registro nella parte iniziale del ciclo e di leggerne il valore scritto nella seconda parte del ciclo (vedi libro).

La dipendenza 1->2 è di distanza 1 indotta dalla LDR. E' il caso peggiore nel pipeline ottimizzato: si risolve con stallo di un ciclo e forwarding dallo stadio di WB a quello di Execute. Nel caso del pipeline puro invece, si richiede uno stallo anche qui di 2 cicli (equivalente a due NOP) in modo da far sì che quando la ADDS è in fase di Decode la LDR sia in fase di WB.

Per le dipendenze di controllo la BEQ non provoca degradazioni se il salto è non preso. Se il salto è preso, nel pipeline con anticipazione di salto si richiede di annullare due istruzioni (quella correntemente in fase di Fetch e quella in fase di Decode) e quindi una bolla di due cicli, mentre nel pipeline "puro" bisogna annullare tutte le istruzioni nel pipeline precedenti alla BEQ quando questa ha raggiunto la fase di WB, quindi quattro istruzioni (bolla di quattro cicli).

Risultato:

- pipeline ottimizzato con BEQ non presa: CPI medio 1.1666
- pipeline ottimizzato con BEQ presa: CPI medio 1.6
- pipeline puro con BEQ non presa: CPI medio 1.6666
- pipeline puro con BEQ presa: CPI medio 2.4

Esercizio 3

La sequenza di scheduling è:

Esecuzione:	**A**	**A**	**B**	**A**	**B**	**C**	**D**	**D**	**D**	**D**	**D**	**D**	**D**	**A**	**B**	**C**	**C**	**C**
Coda pronti:			**A**	**B**	**C,A**	**A,B**	**A,B,C**	**…A,B,C…**						**B,C**	**C**			

PROCESSO (tempo di esecuzione)	TEMPO DI ARRIVO	INIZIO ESECUZIONE	TERMINA ESECUZIONE	TEMPO DI PERMANENZA NEL SISTEMA
A (1)	**0**	**0**	**3.75 (15° QdT)**	**3.75 (15 QdT)**
B (0.75)	**0.5**	**0.5**	**4 (16° QdT)**	**3.5 (14 QdT)**
C (1)	**1.0**	**1.25**	**4.75 (19° QdT)**	**3.75 (15 QdT)**
D (2)	**1.5**	**1.5**	**3.5 (14° QdT)**	**2.00 (8 QdT)**

Il tempo medio di permanenza nel sistema dei processi è 3.25 (13 QdT)

Esercizio 4

1) Si utilizzano 2 semafori condivisi: **prod** inizializzato ad 1, **cons** inizializzato a zero.

```
put(msg) {
   P(&prod);
   inserisci(msg);
   V(&cons);
}
```

```
get(&msg) {
  P(&cons};
  msg= estrai();
  V(&prod);
}
```

2) Si utilizza una variabile di spin-lock **sl** ed un contatore **cnt** inizializzato a 0, condivisi tra i thread.

```
put(msg) {
  bool ok=false;
  do {
   if ( SPIN_LOCK(&sl) == true) {
      if (cnt==1) UNLOCK(&sl);
      else ok=true; // entro in sezione critica
   }
  } while(!ok);
  inserisci(msg);
  cnt=1;
  UNLOCK(&sl);
}
```

```
get(&msg) {
     bool ok=false;
     do {
      if (SPIN_LOCK(&sl) == true) {
        if (cnt==0) UNLOCK(&sl);
        else ok=true; // entro in sezione critica
     } while(!ok);
    msg=estrai();
    cnt=0;
    UNLOCK(&mtx);
 }
```

3) Si utilizzano i seguenti dati condivisi tra i thread: una variabile di mutex **mtx,** una variabile di condizione **cond** (come da richiesta, una sola CV), ed un contatore **cnt** inizializzato a 0.

```
 put(msg) {
    LOCK(&mtx);
    while(cnt==1)
      COND_WAIT(&cond, &mtx);
    inserisci(msg);
    cnt=1;
    COND_BROADCAST(&cond);  (*)
    UNLOCK(&mtx);
  }
```

```
 get(msg) {
   LOCK(&mtx);
   while(cnt==0)
    COND_WAIT(&cond, &mtx);
   msg=estrai();
   cnt=0;
   COND_SIGNAL(&cond);
   UNLOCK(&mtx);
 }
```

NOTA: In (*) va necessariamente utilizzata COND_BROADCAST altrimenti ci può essere deadlock. Dato l'utilizzo di una sola variabile di condizione, la SIGNAL in (*) potrebbe svegliare un produttore in attesa invece che il consumatore in attesa dato che non possiamo supporre un ordinamento FIFO per i risvegli.