Politecnico di Milano
Dipartimento di Elettronica e Informazione

prof.ssa Anna Antola **prof.ssa Donatella Sciuto**
prof. Luca Breveglieri **prof. Roberto Negrini**
prof. Giuseppe Pelagatti **prof.ssa Cristina Silvano**

AXO – Architettura dei Calcolatori e Sistemi Operativi

esame di martedì 5 luglio 2016

CON SOLUZIONI

Cognome _____ **Nome** _____

Matricola _____ **Firma** _____

Istruzioni

Scrivere solo sui fogli distribuiti. Non separare questi fogli.

È vietato portare all'esame libri, eserciziari, appunti, calcolatrici e telefoni cellulari. Chiunque fosse trovato in possesso di documentazione relativa al corso – anche se non strettamente attinente alle domande proposte – vedrà annullata la propria prova.

Non è possibile lasciare l'aula conservando il tema della prova in corso.

Tempo a disposizione: 1h:30m (una parte) 3h:00m (completo).

	punteggio approssimativo	I parte ☐	II parte ☐	completo ☐
esercizio 1	5			
esercizio 2	7			
esercizio 3	4			
esercizio 4	5			
esercizio 5	5			
esercizio 6	6			
voto finale				

ATTENZIONE: alcuni esercizi sono suddivisi in parti.

esercizio n. 1 – linguaggio macchina

Si deve tradurre in linguaggio macchina simbolico (assemblatore) *MIPS* il frammento di programma C riportato sotto. Il modello di memoria è quello **standard *MIPS*** e le variabili intere sono da **32 bit**. Non si tenti di accorpare od ottimizzare insieme istruzioni C indipendenti. Si facciano le ipotesi seguenti:

- il registro "frame pointer" *fp* **non è in uso**
- le variabili locali sono allocate **nei registri o in pila**, secondo le convenzioni note
- vanno salvati (a cura del chiamante o del chiamato, secondo il caso) **solo i registri necessari**

Si chiede di svolgere i tre punti seguenti (usando le varie tabelle predisposte nel seguito):

1. **Si descriva** il segmento dei dati statici, dando anche gli spiazzamenti delle variabili globali rispetto al registro global pointer *gp*, e si traducano in linguaggio macchina le dichiarazioni delle variabili globali.
2. **Si descrivano** l'area di attivazione della funzione *inizializza* e l'allocazione delle variabili locali nei registri del processore.
3. **Si traduca** in linguaggio macchina il codice della funzione *inizializza*.

```
/* costanti e variabili globali */
#define DIM  10
int vout [DIM]
int soglia

/* funzione inizializza */
int inizializza (int * point, int i, int err) {

   /* variabili locali */
   int * soglia_rel
   int temp

   /* parte esecutiva */
   soglia_rel = &soglia
   temp = *soglia_rel – i
   if (err < temp) {
      return i
   } else {
      err = err / 2
      vout [i] = *point + i
      return inizializza (point, i - err, err)
   } /* end if */

} /* inizializza */
```

punto 1 – codice MIPS della sezione dichiarativa globale (num. righe non signif.)		
	.data	// seg. dati statici – 0x 1000 0000
	.eqv *DIM, 10*	*// dichiarazione della costante DIM*
VOUT:	***.space*** *40*	*// spazio per varglob VOUT (non iniz.)*
SOGLIA:	***.space*** *4*	*// spazio per varglob SOGLIA (non iniz.)*

La costante simbolica DIM è dichiarata per comodità espressiva, ma è sempre sostituibile con il relativo valore numerico. Le variabili globali sono allocate in memoria in ordine di dichiarazione.

punto 1 – segmento dati statici (numero di righe non significativo)

contenuto simbolico	indirizzo	spiazzamento rispetto a *gp*	
			indirizzi alti
SOGLIA	*0x 1000 0028*	*0x 8028*	
VOUT [9]	*0x 1000 0024*	*0x 8024*	
…….	……	……	
VOUT [1]	*0x 1000 0004*	*0x 8004*	
VOUT [0]	0x 1000 0000	0x 8000	indirizzi bassi

Gli spiazzamenti delle variabili (e degli elementi del vettore) rispetto al global pointer gp, che vale 0x 1000 8000, sono numeri negativi a 16 bit, a partire da 0x 8000 (negativo poiché ha bit più significativo pari a 1).

punto 2 – area e registri di ***INIZIALIZZA*** (numero di righe non significativo)

area di attivazione di *INIZIALIZZA*		
contenuto simbolico	**spiazz. rispetto a stack pointer**	
ra	*8*	indirizzi alti
s0	*4*	
s1	*0*	← *sp*
		indirizzi bassi

Il registro ra DEVE essere salvato nell'area di attivazione in pila, poiché la funzione INIZIALIZZA è ricorsiva (e quindi NON è di tipo foglia). Le variabili locali SOGLIA_REL (puntatore) e TEMP (intero) vengono allocate nei registri s0 e s1, rispettivamente, poiché non occorre che abbiano un indirizzo esplicito.

allocazione delle variabili locali di *inizializza* nei registri	
variabile locale	**registro**
SOGLIA_REL (indirizzo, è un puntatore a intero)	*s0*
TEMP (intero)	*s1*

La funzione INIZIALIZZA utilizza i registri "callee-saved" s0 e s1 per le sue variabili locali, e dunque tali registri vanno salvati (da parte della funzione) all'inizio e ripristinati (da parte della funzione) alla fine.

punto 3 – codice MIPS di ***INIZIALIZZA*** (num. righe non signif.)			
INIZ:	***addiu***	*$sp, $sp, -12*	*// crea area*
	.eqv	*RA, 8*	*// spi di reg ra salvato*
	.eqv	*S0, 4*	*// spi di reg s0 salvato*
	.eqv	*S1, 0*	*// spi di reg s1 salvato*
	sw	*$ra, RA($sp)*	*// salva reg ra*
	sw	*$s0, S0($sp)*	*// salva reg s0*
	sw	*$s1, S1($sp)*	*// salva reg s1*
	la	*$s0, SOGLIA*	*// assegna SOGLIA_REL = &SOGLIA*
	lw	*$t0, ($s0)*	*// carica oggetto *SOGLIA_REL in t0*
	sub	*$s1, $t0, $a1*	*// assegna TEMP = *SOGLIA_REL – I*
IF:	***bge***	*$a2, $s1, ELSE*	*// se ERR >= TEMP va' a ELSE*
THEN:	***move***	*$v0, $a1*	*// predisponi valore uscita*
	j	*END_IF*	*// va' a END_IF*
ELSE:	***srl***	*$a2, $a2, 1*	*// assegna ERR = ERR / 2*
	lw	*$t1, ($a0)*	*// carica oggetto *POINT in t1*
	add	*$t2, $t1, $a1*	*// calcola espr *POINT + I in t2*
	la	*$t3, VOUT*	*// carica ind di elem VOUT [0] in t3*
	sll	*$t4, $a1, 2*	*// calcola spi di elem VOUT [I] in t4*
	addu	*$t5, $t3, $t4*	*// calcola ind di elem VOUT [I] in t5*
	sw	*$t2, ($t5)*	*// assegna VOUT [I] = *POINT + I*
	sub	*$a1, $a1, $a2*	*// calcola espr I – ERR in a2 (param I)*
	jal	*INIZ*	*// chiamata ricorsiva*
END_IF:	***lw***	*$s1, S1($sp)*	*// ripristina reg s1*
	lw	*$s0, S0($sp)*	*// ripristina reg s0*
	lw	*$ra, RA($sp)*	*// ripristina reg ra*
	addiu	*$sp, $sp, 12*	*// elimina area*
	jr	*$ra*	*// ritorna al chiamante*

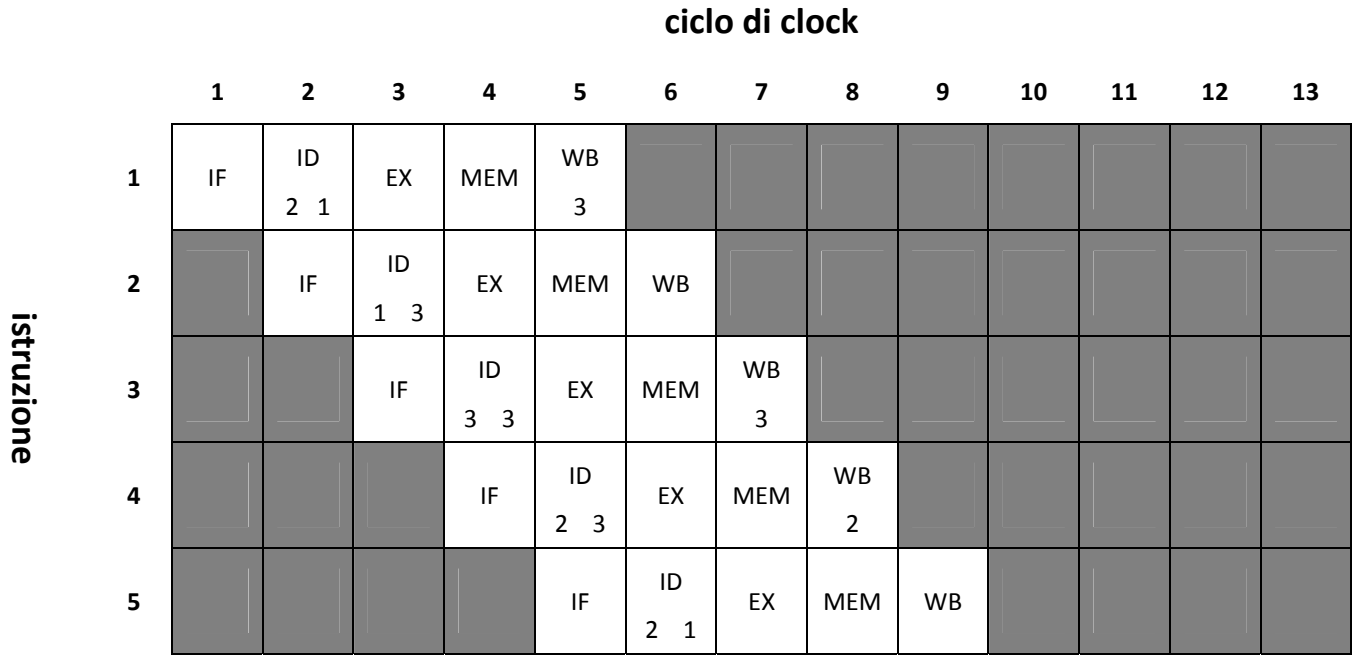
Il codice è sostanzialmente conforme alla convenzioni. Né i registri di argomento $a0-2, né i registri temporanei $t0-5 vengono salvati, poiché al rientro dalla chiamata ricorsiva di INIZIALIZZA essi non vengono più usati. Si potrebbe ottimizzare un poco riusando alcuni registri temporanei e così limitandone il consumo.

esercizio n. 2 – struttura e controllo del processore

prima parte – gestione di conflitti e stalli

Si consideri la sequenza di istruzioni sotto riportata (di cui viene fornito per comodità di analisi anche il diagramma multiciclo teorico):

```
1.  add  $3, $2, $1
2.  sw   $3, ($1)
3.  lw   $3, 40($3)
4.  sub  $2, $2, $3
5.  beq  $2, $1, ADDR
```



Si risponda alle domande seguenti.

punto 1 – Si definiscano tutte le dipendenze di dato completando la **Tabella 1**.

Tabella 1 (domanda 1)
(numero di righe non significativo)

istruzione	istruzione da cui dipende	registro coinvolto
2	*1*	*$3*
3	*1*	*$3*
4	*1*	*$3*
4	*3*	*$3*
5	*4*	*$2*

Tabella 2 (domanda 2.a)
(num. di righe non signif.)

genera conflitto (sì / no)	numero di stalli
si	*2*
si	*1*
no	*0*
si	*2*
si	*2*

punto 2 – Si faccia l'ipotesi che la pipeline **non** sia dotata di percorsi di propagazione.

a) Considerando le dipendenze di dato definite in Tabella 1, **si indichino** in **Tabella 2** quelle che creano un **conflitto**, e per ognuna di queste quanti **stalli** sono necessari per risolvere il conflitto.

b) **Si disegni** il diagramma temporale della pipeline e gli stalli **effettivamente** risultanti.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	*IF*	*ID 2 1*	*EX*	*M*	*WB 3*												
2		*IF*	*ID stall*	*ID stall*	*ID 1 3*	*EX*	*M*	*WB*									
3			*IF stall*	*IF stall*	*IF*	*ID 3 3*	*EX*	*M*	*WB 3*								
4						*IF*	*ID stall*	*ID stall*	*ID 2 3*	*EX*	*M*	*WB 2*					
5							*IF stall*	*IF stall*	*IF*	*ID stall*	*ID stall*	*ID 2 1*	*EX*	*M*	*WB*		
6																	

punto 3 – Si faccia l'ipotesi che la pipeline sia dotata dei cammini di propagazione **EX / EX**, **MEM / EX** e **MEM / MEM**. Per facilitare la risposta è stato riprodotto il codice di partenza.

Si riportino in **Tabella 3** le dipendenze di dato di Tabella 1 che sono risultate essere conflitti, e:

a) **si disegni** in **Tabella 4** il diagramma temporale della pipeline, mettendo in evidenza gli eventuali cammini di propagazione che **possono** essere attivati per risolvere i conflitti, e gli stalli eventualmente da inserire affinché la propagazione sia efficace

b) **si indichino** in **Tabella 3** i cammini di propagazione **effettivamente** attivati e gli stalli associati

```
1  add  $3, $2, $1
2  sw   $3, ($1)
3  lw   $3, 40($3)
4  sub  $2, $2, $3
5  beq  $2, $1, ADDR
```

Tabella 3 (domanda 3.b)
(numero di righe non significativo)

istruzione	istruzione da cui dipende	registro coinvolto	cammino di propagazione e stalli
2	*1*	*$3*	*EX / EX + nessuno stallo*
3	*1*	*$3*	*MEM / EX + nessuno stallo*
4	*3*	*$3*	*MEM / EX + 1 stallo*
5	*4*	*$2*	*EX / EX + nessuno stallo*

Tabella 4 (domanda 3.a)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	*IF*	*ID 2 1*	*EX (3)*	*M (3)*	*WB 3*									
2		*IF*	*ID 1 3*	*EX*	*M*	*WB*								
3			*IF*	*ID 3 3*	*EX*	*M (3)*	*WB 3*							
4				*IF*	*ID stallo*	*ID 2 3*	*EX (2)*	*M (2)*	*WB 2*					
5						*IF*	*ID 2 1*	*EX*	*M*	*WB*				
6														

seconda parte – struttura del processore

Si consideri lo schema di PROCESSORE PIPELINE CON STALLO E PROPAGAZIONE, e i **cammini di propagazione effettivamente attivati** e riportati in **Tabella 3** della domanda precedente, esclusi i cammini MEM / MEM (per i quali non è definita l'unità di propagazione negli schemi circuitali dell'architettura). Per ciascun cammino **si completi** una colonna della **Tabella 5** con le informazioni richieste. Per i registri di interstadio che sono stati scritti in conseguenza di uno stallo, **si indichi V.S.** (valore di stallo).

Tabella 5

		cammino di prop. 1	**cammino di prop. 2**	**cammino di prop. 3**	**cammino di prop. 4**	**cammino di prop. 5**
	istruzione che ne usufruisce	*2*	*3*	*4*	*5*	
	tipo di propagazione	*EX-EX*	*MEM-EX*	*MEM-EX*	*EX-EX*	
	ciclo di clock in cui è attiva la propagazione	*4*	*5*	*7*	*8*	
segnali di ingresso all'unità di propagazione	Rs	*1*	*3*	*2*	*2*	
	Rt	*3*	*3*	*3*	*1*	
	EX / MEM.R	*3*	*3*	*V.S.*	*2*	
	EX / MEM.RegWrite	*1*	*0*	*V.S.*	*1*	
	MEM / WB.R	*XXXXX*	*3*	*3*	*V.S.*	
	MEM / WB.RegWrite	*XXXXX*	*1*	*1*	*V.S.*	
	MUX di propagazione interessato	*PB*	*PA*	*PB*	*PA*	
segnali di uscita dell'unità di propagazione	ingresso di selezione MUX PA	*00*	*01*	*00*	*10*	
	ingresso di selezione MUX PB	*10*	*00*	*01*	*00*	

esercizio n. 3 – logica digitale e memoria cache

prima parte – logica digitale

Sia dato il circuito sequenziale con un **ingresso I** e un'**uscita Z**, descritto dalle equazioni logiche seguenti:

$$D1 = I$$

$$D2 = Q1$$

$$Z = !((!Q1 + !Q2) (!Q1\ !Q2 + !I))$$

Il circuito sequenziale è composto da **due bistabili** master-slave di *tipo D*, cioè (D1, Q1) e (D2, Q2), dove *Di* è l'ingresso del bistabile *i* e *Qi* è lo stato / uscita del bistabile *i*.

Prima di tutto si chiede di semplificare la funzione Z portandola in forma di somma di prodotti (SOP), per facilitare l'analisi:

Z = *!((!Q1 + !Q2) (!(Q1 + Q2) + !I))* *applicando De Morgan ripetutamente*

Z = *!(!(Q1 Q2) (!(Q1 + Q2) + !I))*

Z = *!(!(Q1 Q2) !((Q1 + Q2) I))*

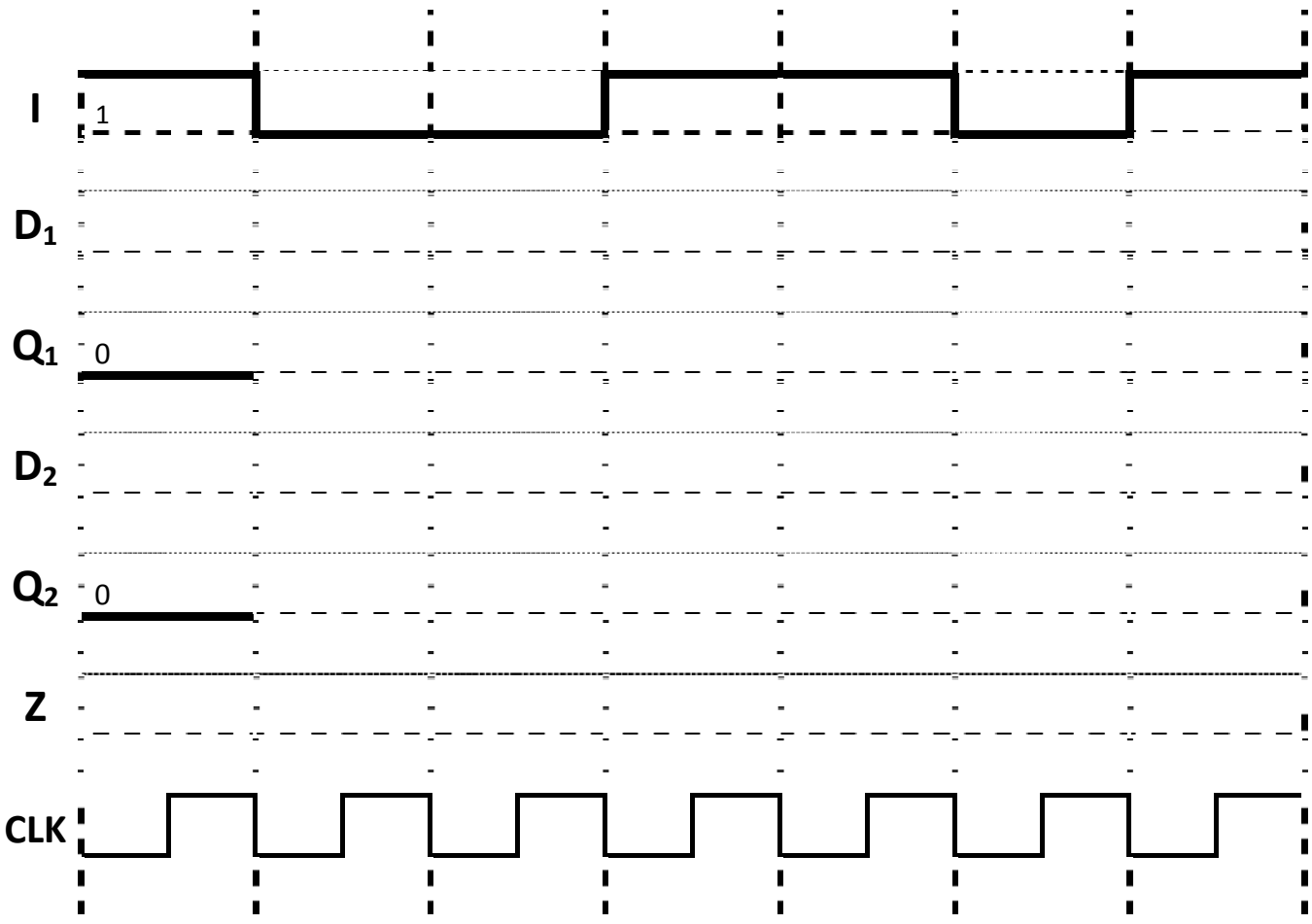
Z = *((Q1 Q2) + ((Q1 + Q2) I))* *e poi svolgendo i prodotti del secondo addendo*

Z = *Q1 Q2 + Q1 I + Q2 I* *questa successione di trasformazioni è solo una delle soluzioni possibili*

Poi si chiede di completare il diagramma temporale riportato qui sotto. Si noti che:

- si devono trascurare completamente i ritardi di propagazione delle porte logiche AND e OR, e i ritardi di commutazione dei bistabili
- il bistabile è il tipo master-slave la cui struttura è stata trattata a lezione, con uscita che commuta sul fronte di discesa del clock

diagramma temporale da completare

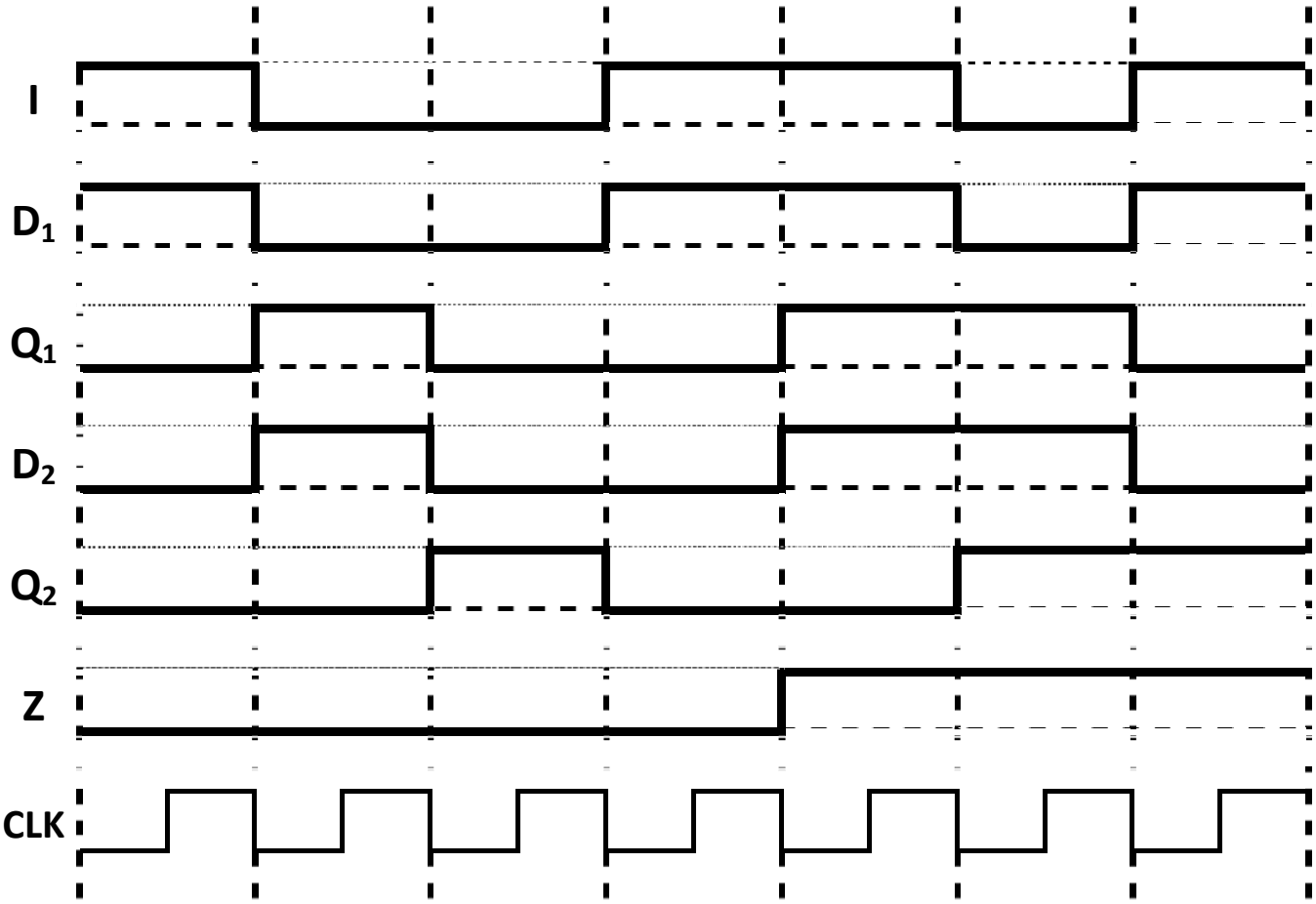


SOLUZIONE

Dalle equazioni si vede facilmente che i due bistabili formano un registro a scorrimento, in cui entra e scorre il segnale di ingresso. Dunque per le forme d'onda si può subito dire:

- *l'ingresso D1 è uguale a I*
- *lo stato Q1 ripete D1 con un intervallo di ritardo (il primo intervallo è dato)*
- *l'ingresso D2 ripete Q1*
- *lo stato Q2 ripete D2 con un intervallo di ritardo (il primo intervallo è dato)*
- *l'uscita Z viene calcolato con la formula, intervallo per intervallo; il calcolo è molto semplificato se si considera che Z = Q1 Q2 + Q1 I + Q2 I* ***vale 1 se e solo se*** *almeno due tra Q1, Q2 e I valgono 1; in ogni intervallo basta quindi contare quanti 1 ci sono tra Q1, Q2 e I*

Pertanto ricavare il diagramma temporale è semplice e rapido. Eccolo:



seconda parte – memoria cache

Si consideri un sistema di memoria (memoria centrale + cache) con le caratteristiche seguenti:

- memoria di lavoro di **4 K byte**, indirizzata a livello di singolo byte
- cache associativa a gruppi (set-associative) a **2 vie** di **512 byte** in totale
- ogni blocco della cache contiene **128 byte**, dunque la cache contiene **4 blocchi**, denotati dalle lettere **A**, **B**, **C** e **D**
- i blocchi **A** e **B** appartengono all'**insieme 0**
- i blocchi **C** e **D** appartengono all'**insieme 1**
- la politica di sostituzione adottata nella cache è **LRU** (Least Recently Used)

Si chiede di completare la Tabella seguente, considerando la sequenza di richieste alla memoria (lettura o scrittura non fa differenza) riportata nella colonna **indirizzo richiesto**.

Il significato delle diverse colonne della tabella è il seguente:

Nella colonna **esito** riportare **H** (hit) se il blocco richiesto si trova nella cache, oppure **M** (miss) se invece il blocco va caricato dalla memoria.

Nelle colonne **dati** va riportato il **numero del blocco della memoria** che si trova nel corrispondente blocco di cache. Si noti che questi valori vanno espressi come numeri decimali (base dieci), mentre le etichette sono scritte in binario.

Nella colonna **azione** va indicato il **blocco a cui si accede** (in caso di successo **H**) o il blocco in cui vengono caricati i dati della memoria (in caso di fallimento **M**).

Indirizzi verso la memoria: 4K byte richiedono 12 bit di indirizzo, ma i 7 bit meno significativi sono usati per identificare la posizione del byte nel blocco di 128 byte; dunque rimangono i 5 bit più significativi per identificare la posizione del blocco in memoria: li usiamo – tradotti in decimale – per specificare il numero del blocco richiesto.

Indirizzi verso la cache: trascurando i 7 bit meno significativi (byte nel blocco), l'ottavo bit identifica l'insieme (cioè indirizza e attiva la prima o la seconda posizione in ogni via: blocchi A o B se 0, C o D se 1) e i rimanenti quattro bit – i più significativi – vanno usati come etichetta (si immagini che una via comprenda i blocchi A e C, e l'altra via i blocchi B e D, con A e B ai primi posti e C e D ai secondi posti).

			insieme 0						**insieme 1**						
			blocco A			**blocco B**			**blocco C**			**blocco D**			
passo	**indirizzo richiesto**	**esito**	**valido**	**etichetta**	**dati**	**valido**	**etichetta**	**dati**	**valido**	**etichetta**	**dati**	**valido**	**etichetta**	**dati**	**azione**
0			**1**	**1000**	**17**	**0**	**–**	**–**	**1**	**0101**	**3**	**0**	**–**	**–**	**situazione iniziale**
1	0011 1101 0110	*M*	*1*	*1000*	*17*	*0*	*–*	*–*	*1*	*0101*	*3*	*1*	*0011*	*7*	*carica blocco 7 in D**
2	1001 1101 0100	*M*	*1*	*1000*	*17*	*0*	*–*	*–*	*1*	*1001*	*19*	*1*	*0011*	*7*	*carica blocco 19 in C***
3	1001 1011 1111	*H*	*1*	*1000*	*17*	*0*	*–*	*–*	*1*	*1001*	*19*	*1*	*0011*	*7*	*accesso a C*
4	1100 0000 1010	*M*	*1*	*1000*	*17*	*1*	*1100*	*24*	*1*	*1001*	*19*	*1*	*0011*	*7*	*carica blocco 24 in B°*
5	1101 1101 0010	*M*	*1*	*1000*	*17*	*1*	*1100*	*24*	*1*	*1001*	*19*	*1*	*1101*	*27*	*carica blocco 27 in D °°*
6	1011 0100 0110	*M*	*1*	*1011*	*22*	*1*	*1100*	*24*	*1*	*1001*	*19*	*1*	*1101*	*27*	*carico blocco 22 in A ^*

** D è libero e C no* *** C è usato meno di recente (LRU) rispetto a D* *° B è libero e A no*

°° D è LRU rispetto a C *^ A è LRU rispetto a B*

esercizio n. 4 – processi e parallelismo

Si consideri il programma C seguente (gli "#include" e le inizializzazioni dei mutex sono omessi):

```
pthread_mutex_t SEA, WIND
sem_t waves
int global = 0
```

```
void * one (void * arg) {
   pthread_mutex_lock (&SEA)
   global = 1                                          /* statement A */
   sem_wait (&waves)
   pthread_mutex_unlock (&SEA)                         /* statement B */
   pthread_mutex_lock (&WIND)
   sem_post (&waves)
   pthread_mutex_unlock (&WIND)
   return NULL
} /* end one */
```

```
void * two (void * arg) {
   pthread_mutex_lock (&WIND)
   pthread_mutex_lock (&SEA)
   sem_post (&waves)                                   /* statement C */
   pthread_mutex_unlock (&SEA)
   pthread_mutex_unlock (&WIND)
   global = 2                                          /* statement D */
   pthread_mutex_lock (&WIND)
   sem_wait (&waves)
   pthread_mutex_unlock (&WIND)
   return (void *) 3
} /* end two */
```

```
void main ( ) {
   pthread_t th_1, th_2
   sem_init (&waves, 0, 0)
   pthread_create (&th_1, NULL, one, NULL)
   pthread_create (&th_2, NULL, two, NULL)
   pthread_join (th_2, &global)
   printf (global)                                     /* statement E */
   pthread_join (th_1, NULL)
   return
} /* end main */
```

Si completi la tabella qui sotto **indicando lo stato di esistenza del thread** nell'istante di tempo specificato da ciascuna condizione, così: se il thread **esiste**, si scriva ESISTE; se **non esiste**, si scriva NON ESISTE; e se può essere **esistente o inesistente**, si scriva PUÒ ESISTERE. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna "condizione": con "subito dopo statement X" si chiede lo stato che il thread assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**thread**	
	th_1	**th_2**
subito dopo stat. **A**	*ESISTE*	*PUÒ ESISTERE **
subito dopo stat. **C**	*ESISTE*	*ESISTE*
subito dopo stat. **D**	*PUÒ ESISTERE +*	*ESISTE*
subito dopo stat. **E**	*PUÒ ESISTERE +*	*NON ESISTE*

** Non creato, esistente o terminato.* *+ Esistente o terminato.*

Si completi la tabella qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)
- il mutex ha valore 0 se è libero oppure ha valore 1 se è occupato

Si badi bene alla colonna "condizione": con "subito dopo statement X" si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabili globali**			
	SEA	*WIND*	*waves*	*global*
subito dopo stat. **A**	*1*	*0 / 1*	*0 / 1*	*1 / 2*
subito dopo stat. **B**	*0*	*0 / 1*	*0*	*1 / 2*
subito dopo stat. **C**	*1*	*1*	*1*	*0*

Il sistema può andare in stallo (deadlock), con uno o più thread che si bloccano, in **TRE casi diversi** (si osservi che si chiede anche il thread principale). Si indichino gli statement dove avvengono i blocchi, con il valore (o i valori) della variabile `global`:

caso	**th_1**	**th_2**	**main**	*global*
1	***wait waves***	***lock SEA***	***join th_2***	*1*
2	***wait waves***		***join th_1***	*2 / 3*
3	***lock WIND***	***wait waves***	***join th_2***	*1 / 2*

esercizio n. 5 – gestione della memoria

Si consideri una memoria fisica di **32 K byte** disponibile per i processi di modo *U*, con ***MIN_FREE*** = **2** e ***MAX_FREE*** = **3**. In memoria c'è un processo **P** in esecuzione.

domanda 1 – Il processo **P** esegue un servizio di *exec* ( ), poi accede alla pagina **ps0** (in lettura), e alle pagine **pd0** e **pd1** (in scrittura).

Si mostri lo stato della memoria, lo stato delle liste **LRU** e lo stato del **TLB** dopo tali eventi.

memoria fisica (solo modo *U*)							
NPF	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***
0	*pc0*	**1**	*pp0*	**2**	*ps0*	**3**	*pd0*
4	*pd1*	**5**		**6**		**7**	
swap file							

liste LRU		
attivazioni	***active***	***inactive***
iniziale	irrilevante (pagine di **P** prima di *exec*)	irrilevante (pagine di **P** prima di *exec*)
finale	*PD1 PD0 PS0 PP0 PC0*	

TLB (solo righe di modo *U*) (completare inserendo le pagine in ordine virtuale di pagina)							
NPV	***NPF***	***D***	***A***	***NPV***	***NPF***	***D***	***A***
pc0	*0*		*1*	*ps0*	*2*		*1*
pd0	*3*	*1*	*1*	*pd1*	*4*	*1*	*1*
pp0	*1*	*1*	*1*				

domanda 2 – Il processo **P** accede continuamente alle pagine **pc0** (in lettura) e **pd1** (in scrittura); durante questo periodo il daemon ***kswapd*** interviene tre volte.

Si mostri lo stato delle liste **LRU** e del **TLB** (a pagina seguente).

liste LRU		
attivazioni	***active***	***inactive***
iniziale	*PD1 PD0 PS0 PP0 PC0*	
finale	*PD1 PC0*	*pd0 ps0 pp0*

TLB (solo righe di modo *U*)							
NPV	***NPF***	***D***	***A***	***NPV***	***NPF***	***D***	***A***
pc0	*0*		*1*	*ps0*	*2*		
pd0	*3*	*1*		*pd1*	*4*	*1*	*1*
pp0	*1*	*1*					

domanda 3 – Il processo **P** alloca una nuova pagina di pila e una nuova pagina di dati dinamici in una sola operazione.

Si compili lo stato della memoria.

PFRA interviene per liberare due pagine: pp0 e ps0; ma solo pp0 va nello swap file, poiché ps0 non è dirty.

memoria fisica (solo modo *U*)							
NPF	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***
0	*pc0*	**1**	*pp1*	**2**	*pd2*	**3**	*pd0*
4	*pd1*	**5**		**6**		**7**	
swap file	*pp0*						

domanda 4 – Il processo **P** esegue una *fork* creando il processo **Q**, poi **P** viene sospeso e viene messo in esecuzione **Q**, che accede le pagine **qc0** e **qd2** (in lettura), e la pagina **qp1** (in scrittura).

Si compilino lo stato della memoria e del **TLB.**

Con la fork le pagine vengono condivise, tranne pp1 che viene riallocata nella prima cella libera (5), mentre qp1 prende il suo posto (1). I successivi accessi non causano modifiche alla memoria, ma solo al TLB.

memoria fisica (solo modo *U*)							
NPF	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***
0	*pc0 / qc0*	**1**	*qp1*	**2**	*pd2 / qd2*	**3**	*pd0 / qd0*
4	*pd1 / qd1*	**5**	*pp1*	**6**		**7**	
swap file	*pp0 / qp0*						

TLB (solo righe di modo *U*)							
NPV	***NPF***	***D***	***A***	***NPV***	***NPF***	***D***	***A***
qc0	*0*		*1*	*qd2*	*2*		*1*
qp1	*1*	*1*	*1*				

esercizio n. 6 – nucleo e componenti del SO

prima parte – scheduling dei processi

È data la tabella di *scheduling* mostrata alla pagina successiva, con un elenco di attività e tempi. Valgono le convenzioni seguenti:

- i tempi di esecuzione sono misurati in *millisecondi* (*ms*)
- i parametri di *CFS* hanno i valori di default: *LT* = 6 *ms*, *GR* = 0,75 *ms* e *WGR* = 1 *ms*

Le attività già specificate o ancora *da specificare* vanno annotate così:

exe U x ms	*ti.nome_funzione_di_SO …*	*csw ti → tj*
un task (da determinare) ha eseguito codice utente (modo *U*) dal tempo assoluto precedente fino al tempo assoluto *x ms*	il task *ti* ha eseguita la *funzione_di_SO*; se serve si specifica un oggetto … (argom. o indicazione); se il tempo non è dato, è trascurabile	avviene la commutazione di contesto dal task *ti* al task *tj*; il tempo di commutazione di contesto è sempre trascurabile

Si effettui lo scheduling *CFS* dei task (classe *NORMAL*) indicati sulla base di azioni e tempi indicati nelle colonne **ATTIVITÀ** e **TEMPI**, compilando i campi vuoti non oscurati. Per ogni riga, vanno riportati i valori che si avranno alla fine dell'attività.

La colonna **TASK e CONDIZIONI** è per eventuali annotazioni ausiliarie: a ciascuna attività specificata va aggiunto il nome del *task* che la svolge, va calcolato il *VRT* di risveglio, va valutata una condizione di *preemption*, o va detto se scade il quanto. Secondo le azioni specificate, *task* esistenti possono terminare e *task* nuovi possono essere creati (in quest'ultimo caso si usino le eventuali colonne lasciate vuote in partenza).

PER RISOLVERE L'ESERCIZIO SI USI LA TABELLA PREPARATA A PAGINA SEGUENTE.

TABELLA DI *SCHEDULING* DA COMPLETARE

ATTIVITÀ	TEMPO	TASK e CONDIZIONI	RUNQUEUE		TASK			
					***L0* = 1**	***t1* *LOAD* = 1**	***t2* *LOAD* = 2**	***t3* *LOAD* = 1**
inizio	0 *ms*		NRT		LC			
			PER		Q			
			RQL		VRTC			
			CURR	**t1**	DELTA	**0**	**0**	
			RB	**t2**	SUM	**1**	**0**	
			VMIN	**0**	VRT	**1**	**0**	
exe U	4,5 *ms*		NRT		LC			
			PER		Q			
			RQL		VRTC			
			CURR		DELTA			
			RB		SUM			
			VMIN		VRT			
sys_nanosleep (5,5 *ms*)			NRT		LC			
			PER		Q			
			RQL		VRTC			
			CURR		DELTA			
exe U	5,5 *ms*		RB		SUM			
			VMIN		VRT			
sys_clone			NRT		LC			
			PER		Q			
			RQL		VRTC			
			CURR		DELTA			
exe U	10 *ms*		RB		SUM			
			VMIN		VRT			
sys_exit			NRT		LC			
			PER		Q			
contr_timer			RQL		VRTC			
			CURR		DELTA			
exe U	11 *ms*		RB		SUM			
			VMIN		VRT			

SOLUZIONE CON TASK CORRENTE INIZIALE t1

ATTIVITÀ	TEMPO	TASK e CONDIZIONI	RUNQUEUE		TASK			
					***L0* = 1**	***t1* LOAD = 1**	***t2* LOAD = 2**	***t3* LOAD = 1**
inizio	0 *ms*		NRT	*2*	LC	*1 / 3 ≈ 0,33*	*2 / 3 ≈ 0,66*	
			PER	*6*	Q	*6 × 1/3 = 2*	*6 × 2/3 = 4*	
			RQL	*3*	VRTC	*1 / 1 = 1*	*1 / 2 = 0,5*	
			CURR	**t1**	DELTA	**0**	**0**	
			RB	**t2**	SUM	**1**	**0**	
			VMIN	**0**	VRT	**1**	**0**	
exe U	4,5 *ms*	*t1.exe U 2* *t1.Q scade* *csw t1 → t2* *t2.exe U 4,5*	NRT	*2*	LC	*1 / 3 ≈ 0,33*	*2 / 3 ≈ 0,66*	
			PER	*6*	Q	*6 × 1/3 = 2*	*6 × 2/3 = 4*	
			RQL	*3*	VRTC	*1 / 1 = 1*	*1 / 2 = 0,5*	
			CURR	*t2*	DELTA	*2*	*2,5*	
			RB	*t1*	SUM	*3*	*2,5*	
			VMIN	*1,25*	VRT	*3*	*1,25*	
sys_nanosleep (5,5 *ms*) exe U	5,5 *ms*	*t2.sys_nanosleep* *(risveglio tra 5,5 ms)* *csw t2 → t1* *t1.exe U 5,5*	NRT	*1*	LC	*1 / 1 = 1*	*IN ATTESA* *(risveglio via controlla_timer al tempo assoluto 7,5 ms)*	
			PER	*6*	Q	*6 × 1/1 = 6*		
			RQL	*1*	VRTC	*1 / 1 = 1*		
			CURR	*t1*	DELTA	*1*		
			RB	*vuota*	SUM	*4*		
			VMIN	*4*	VRT	*4*		
sys_clone exe U	10 *ms*	*t1.sys_clone t3* *t3.vrt = VMIN+t3.Q×t3.VRTC= 4 + 3×1 = 7* *t1.cond_pr 7 + 1×0,5 = 7,5 < 4 è falsa* *task t1 continua* *t1.exe U 7,5* *t1.Q scade* *csw t1 → t3* *t3.exe U 10*	NRT	*2*	LC	*1 / 2 = 0,5*	*IN ATTESA* *(risveglio via controlla_timer al tempo assoluto 7,5 ms)*	*1 / 2 = 0,5*
			PER	*6*	Q	*6 × 1/2 = 3*		*6 × 1/2 = 3*
			RQL	*2*	VRTC	*1 / 1 = 1*		*1 / 1 = 1*
			CURR	*t3*	DELTA	*3*		*2,5*
			RB	*t1*	SUM	*6*		*2,5*
			VMIN	*6*	VRT	*6*		*9,5*
sys_exit controlla_timer exe U	11 *ms*	*t3.sys_exit* *csw t3 → t1* *t1.contr_timer* *t1.wkup_proc t2* *t2.vrt = max(t2.vrt,VM−LT/2) = max(1,25, 6−6/2)= 3* *t1.cond_pr 3+1×0,66 = 3,66 < 6 è vera* *csw t1 → t2* *t2.exe U 11*	NRT	*2*	LC	*1 / 3 ≈ 0,33*	*2 / 3 ≈ 0,66*	*TERMINATO*
			PER	*6*	Q	*6 × 1/3 = 2*	*6 × 2/3 = 4*	
			RQL	*3*	VRTC	*1 / 1 = 1*	*1 / 2 = 0,5*	
			CURR	*t2*	DELTA	*0*	*1*	
			RB	*t1*	SUM	*6*	*3,5*	
			VMIN	*6*	VRT	*6*	*3,5*	

NOTA

Si osservi che il quanto di tempo del task t1 cambia mentre il task t1 è corrente, poiché al tempo assoluto di 5,5 ms il task t1 chiama il servizio sys_clone, e nell'eseguirlo modifica il parametro NRT e così induce il ricalcolo dei quanti di tempo: in particolare il quanto t1.Q cala da 6 ms a 3 ms. Poiché il task padre t1 resta corrente dopo sys_clone giacché la condizione di preemption risulta falsa, mentre il task figlio t3 resta accodato in RB per un po', dalla fine di sys_clone in poi il quanto che vale per t1 è quello ricalcolato. Tuttavia va tenuto conto che il task t1 sta già eseguendo da un po' di tempo prima di chiamare sys_clone: qui da 1 ms, come si vede dal suo campo DELTA; infatti il task t1 diventa corrente al tempo assoluto di 4,5 ms e chiama sys_clone al tempo assoluto di 5,5 ms, dunque dopo 1 ms. Però, concluso sys_clone, il quanto di t1, pur essendo calato (da 6 ms a 3 ms), non è ancora scaduto, pertanto il task t1 resta ancora corrente, MA SOLO PER ALTRI 2 ms (e non per altri 5 ms come in linea di principio il quanto avrebbe permesso prima del ricalcolo), poiché così consuma tutto il quanto RICALCOLATO. Si veda come è definita la condizione di scadenza del quanto di tempo che viene valutata nella funzione "task_tick_fair" a ogni tick di orologio (real time clock): il quanto corrente viene confrontato con l'intervallo di tempo trascorso da quando il task corrente è ANDATO IN ESECUZIONE per l'ultima volta, tramite il campo PREV del descrittore di task dove viene appunto salvato l'istante di tempo dell'ultima commutazione di contesto che ha (ri)messo il task in esecuzione. Di tutto ciò va tenuto conto nell'aggiornare i valori dei campi SUM e VRT (e a seguito di VMIN).

Più in generale giova ricordare che i campi DELTA, SUM e VRT del task corrente vengono (ri)calcolati nella funzione "task_tick_fair" a ogni tick, ma per comodità nella tabella di simulazione li si vede solo alla transizione tra righe consecutive, ciascuna delle quali contiene uno o più eventi e dura un numero più o meno grande di tick. Inoltre si ricordi che in simulazione il campo DELTA di un task esprime il tempo speso in esecuzione dal task a partire dall'ultima commutazione di contesto che ha (ri)messo il task in esecuzione (in sostanza il campo DELTA viene (re)inizializzato a zero ogniqualvolta il task viene (ri)messo in esecuzione per commutazione di contesto), e che il campo SUM di un task esprime (sia in simulazione sia in esecuzione reale) il tempo TOTALE speso in esecuzione da parte del task a partire dalla sua creazione (ossia dalla sua clonazione). In tale modo in simulazione risulta assai facile aggiornare il campo SUM e soprattutto il campo VRT, che è essenziale per ordinare i task in runqueue e decidere quale task andrà in esecuzione.

Se però si esamina in dettaglio il codice della funzione "task_tick_fair", si constata che il campo DELTA esprime l'intervallo di tempo tra due tick consecutivi, e che sono questi (brevi) intervalli (durata ≈ 0,01 ms) ad essere progressivamente accumulati nel campo SUM. Ovviamente si tratta di due modi equivalenti di gestire il medesimo computo del tempo reale e di quello virtuale del task: in simulazione non si segue l'effetto di ciascun tick sui campi SUM e VRT del task corrente (infatti il tick dura ≈ 0,01 ms e il quanto tipicamente qualche ms, dunque in media ci sono centinaia di tick per quanto, che sarebbero troppi per visualizzarne gli effetti uno a uno); invece l'esecuzione reale ricalcola i campi SUM e VRT a ciascun tick, con una granularità temporale piuttosto fine allo scopo di dare allo scheduler CFS tutta la precisione richiesta.

AVVERTENZA

Per un refuso di stampa, durante l'esame nella tabella di simulazione non figurava specificato quale fosse il task corrente all'inizio dell'osservazione, ossia se fosse t1 oppure t2 (lo stato di partenza previsto alla stesura dell'esercizio avrebbe dovuto essere quello con task corrente t1). A tale riguardo si osservi quanto segue:

- *All'inizio dell'osservazione il task corrente può essere uno qualunque di quelli in runqueue, non necessariamente quello con VRT minimo. Infatti il VRT del task corrente cresce progressivamente mentre il task è in esecuzione, e per un certo intervallo di tempo può superare i VRT dei task pronti in coda RB poiché la commutazione di contesto può avvenire solo in certi momenti, ossia alla scadenza di un quanto di tempo oppure al risveglio di un task sospeso. Dunque quale sia il task corrente all'inizio dell'osservazione non è deducibile dal solo esame dei VRT dei task in runqueue.*
- *In generale un task figlio clonato dal task padre via servizio sys_clone ha lo stesso peso (LOAD) del task padre. Tuttavia in ogni istante il sistema può cambiare il peso di qualunque task, con i comandi di shell o i servizi di sistema a ciò preposti. Dunque per un task figlio clonato si può senz'altro specificare un peso diverso da quello del task padre, poiché una modifica di peso può intervenire in qualunque istante.*

Pertanto nell'esercizio dato, se manca la specifica del task corrente all'inizio dell'osservazione, supporre che esso sia il task t2 è un'ipotesi coerente tanto quanto supporre che lo sia il task t1. A pagina seguente è riportata la soluzione nell'ipotesi di task corrente iniziale t2. La nuova soluzione differisce dalla precedente quasi solo per lo scambio di ruolo dei task t1 e t2, nonché per lievi cambiamenti dei tempi: infatti le condizioni di preemption hanno lo stesso esito di prima, ma il task t2 esegue per un po' più di tempo rispetto a prima poiché ha peso (LOAD) maggiore del task t1, e ciò comporta un quanto di tempo più lungo. Però si badi bene che, in linea generale, la nuova soluzione potrebbe avere un andamento del tutto differente.

Inoltre vale la stessa nota fatta alla soluzione precedente riguardo al fatto che in determinate circostanze, p. es. chiamando il servizio sys_clone, il quanto di tempo del task corrente può cambiare: qui succede di nuovo ma scambiando i ruoli dei task t1 e t2, e con tempi un po' diversi.

SOLUZIONE CON IPOTESI DI TASK CORRENTE INIZIALE t2

ATTIVITÀ	TEMPO	TASK e CONDIZIONI	RUNQUEUE		TASK			
					***L0* = 1**	***t1* LOAD = 1**	***t2* LOAD = 2**	***t3* LOAD = 1**
inizio	0 *ms*		NRT	*2*	LC	*1 / 3 ≈ 0,33*	*2 / 3 ≈ 0,66*	
			PER	*6*	Q	*6 × 1/3 = 2*	*6 × 2/3 = 4*	
			RQL	*3*	VRTC	*1 / 1 = 1*	*1 / 2 = 0,5*	
			CURR	**t2**	DELTA	**0**	**0**	
			RB	**t1**	SUM	**1**	**0**	
			VMIN	**0**	VRT	**1**	**0**	
exe U	4,5 *ms*	*t2.exe U 4* *t2.Q scade* *csw t2 → t1* *t1.exe U 4,5*	NRT	*2*	LC	*1 / 3 ≈ 0,33*	*2 / 3 ≈ 0,66*	
			PER	*6*	Q	*6 × 1/3 = 2*	*6 × 2/3 = 4*	
			RQL	*3*	VRTC	*1 / 1 = 1*	*1 / 2 = 0,5*	
			CURR	*t1*	DELTA	*0,5*	*4*	
			RB	*t2*	SUM	*1,5*	*4*	
			VMIN	*1,5*	VRT	*1,5*	*2*	
sys_nanosleep (5,5 *ms*) exe U	5,5 *ms*	*t1.sys_nanosleep* *(risveglio tra 5,5 ms)* *csw t1 → t2* *t2.exe U 5,5*	NRT	*1*	LC	*IN ATTESA* *(risveglio via controlla_timer al tempo assoluto 8,5 ms)*	*2 / 2 = 1*	
			PER	*6*	Q		*6 × 2/2 = 6*	
			RQL	*2*	VRTC		*1 / 2 = 0,5*	
			CURR	*t2*	DELTA		*1*	
			RB	*vuota*	SUM		*5*	
			VMIN	*2,5*	VRT		*2,5*	
sys_clone exe U	10 *ms*	*t2.sys_clone t3* *t3.vrt = VMIN+t3.Q×t3.VRTC= 2,5+2×1 = 4,5* *t2.cond_pr 4,5+1×0,33 = 4,83 < 2,5 è falsa* *task t2 continua* *t2.exe U 8,5* *t2.Q scade* *csw t2 → t3* *t3.exe U 10*	NRT	*2*	LC	*IN ATTESA* *(risveglio via controlla_timer al tempo assoluto 8,5 ms)*	*2 / 3 ≈ 0,66*	*1 / 3 ≈ 0,33*
			PER	*6*	Q		*6 × 2/3 = 4*	*6 × 1/3 = 2*
			RQL	*3*	VRTC		*1 / 2 = 0,5*	*1 / 1 = 1*
			CURR	*t3*	DELTA		*4*	*1,5*
			RB	*t2*	SUM		*8*	*1,5*
			VMIN	*4*	VRT		*4*	*6*
sys_exit controlla_timer exe U	11 *ms*	*t3.sys_exit* *csw t3 → t2* *t2.contr_timer* *t2.wkup_proc t1* *t1.vrt = max(t1.vrt,VM−LT/2) = max(1,5, 4−6/2)= 1,5* *t2.cond_pr 1,5+1×0,33 = 1,83 < 4 è vera* *csw t2 → t1* *t1.exe U 11*	NRT	*2*	LC	*1 / 3 ≈ 0,33*	*2 / 3 ≈ 0,66*	*TERMINATO*
			PER	*6*	Q	*6 × 1/3 = 2*	*6 × 2/3 = 4*	
			RQL	*3*	VRTC	*1 / 1 = 1*	*1 / 2 = 0,5*	
			CURR	*t1*	DELTA	*1*	*0*	
			RB	*t2*	SUM	*2,5*	*8*	
			VMIN	*4*	VRT	*2,5*	*4*	

seconda parte – file system

Un processo **P** esegue il programma mostrato di seguito, creando un processo figlio **Q**, che a sua volta crea un processo figlio **R**.

```
int main ( ) { /* processo P */
   /* dichiarazioni varie */
   fd1 = open ("/cat1/cat3/filealfa", O_RDONLY)
   read (fd1, bufP1, 3)
   pid = fork ( )
   if (pid == 0) {  /* processo Q */
      fd2 = open ("/cat1/cat3/filebeta", O_RDONLY)
      lseek (fd2, 1020, 0)
      read (fd2, bufQ1, 10)
      read (fd2, bufQ1, 600)
      pid = fork ( )
      if (pid == 0) {  /* processo R */
          fd3 = open ("/cat1/cat2/filealfa", O_RDONLY)
          read (fd3, bufR3, 6)
          read (fd3, bufR3, 2)
          exit (0)
      } /* end if */
      pid = wait (NULL)
      close (fd2)
      close (fd1)
      exit (0)
   } /* end if */
exit (0)
} /* end main */
```

A un certo istante di tempo T_1 si sono verificati i tre eventi seguenti:

1) dopo l'esecuzione della prima *fork* è andato subito in esecuzione il processo **Q**
2) dopo l'esecuzione della seconda *fork* è andato subito in esecuzione il processo **R**
3) il processo **Q** ha eseguita la funzione *exit*
4) il processo **P** non è ancora terminato

Il contenuto del **volume** durante l'esecuzione è il seguente:

i-lista:	< 0, dir, 8 > < 3, dir, 9 > < 7, dir, 20 > < 10, dir, 12 > < 62, norm, 150 >
	< 86, norm, {172 , 173, 174, 175}> < 98, norm, 190 >
blocco 8:	... < 1, dev > < 3, cat1 > ...
blocco 9:	... < 7, cat3 > ... < 10, cat2 > ...
blocco 12:	... < 62, filealfa >
blocco 20:	... < 98, filealfa > ... < 86, filebeta > ...

Nota bene: lo *i-node* associato al catalogo radice "/" ha 0 come *i-number*, la *i-lista* contiene terne < *i-number*, *tipo_file*, *indice_blocco* >, e i cataloghi contengono coppie < *i-number*, *nome_file* >.

Si completi il contenuto delle tabelle seguenti, indicando la sequenza di valori assunti fino all'istante T_1 (si scriva **L** oppure **NE** per indicare che una cella si è liberata oppure non esiste più, rispettivamente).

tabella dei file del processo *P*		**tabella dei file del processo *Q***		**tabella dei file del processo *R***		**tabella globale dei file aperti**			
file des	**riferimento a riga**	**file des**	**riferimento a riga**	**file des**	**riferimento a riga**	**riga**	**posizione corrente**	**numero di riferimenti**	**i-number**
0	×	**0**	×	**0**	×	**0**	×	×	×
1	×	**1**	×	**1**	×	**1**	×	×	×
2	×	**2**	×	**2**	×	**2**	×	×	×
3	*3*	**3**	*3 L NE*	**3**	*3 NE*	**3**	*0 3*	*1 2 3 2 1*	*98*
4		**4**	*4 L NE*	**4**	*4 NE*	**4**	*0 1020 1030 1630 L*	*1 2 1 L*	*86 L*
5		**5**		**5**	*5 NE*	**5**	*0 6 8 L*	*1 L*	*62 L*
6		**6**		**6**		**6**			

Nota bene i nomi dei campi: *file des* indica il numero di descrittore di file; *riferimento a riga* indica il riferimento alla riga della tabella globale dei file aperti dove si trova il descrittore del file; *posizione corrente* è l'indicatore di posizione corrente all'interno del file; *numero di riferimenti* è il numero di riferimenti da parte dei processi che hanno aperto il file; *i-number* si riferisce allo *i-node* del file.

Il segno "**×**" indica che è presente un valore non significativo ai fini del problema. Le tabelle sono semplificate e contengono solamente le colonne che si chiede di riempire.