Politecnico di Milano

Dipartimento di Elettronica, Informazione e Bioingegneria

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 mercoledì 7 settembre 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	6			
esercizio 2	6			
esercizio 3	4			
esercizio 4	5			
esercizio 5	6			
esercizio 6	5			
voto finale				

ATTENZIONE: alcuni esercizi sono suddivisi in parti.

esercizio n. 1 – linguaggio macchina

prima parte – traduzione da C a 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), preferibilmente in ordine di elenco:

1. **Si descriva** il segmento dei dati statici: si traducano in linguaggio macchina le dichiarazioni delle variabili globali e si diano gli spiazzamenti delle variabili globali rispetto al registro global pointer *gp*.
2. **Si descrivano** l'area di attivazione della funzione *scan* e l'allocazione delle variabili locali nei registri del processore.
3. **Si risponda** brevemente, nello spazio assegnato, alle domande elencate.
4. **Si traduca** in linguaggio macchina il codice della funzione *scan*, limitatamente alla porzione di codice nel riquadro, tenendo conto di quanto già detto rispondendo ai tre punti precedenti

```
/* costanti e variabili globali */
#define MAX 5
char string [MAX]

/* funzione scan */
int scan (char * p, int pos) {

   /* variabili locali */
   int temp

   /* parte esecutiva */
   temp = pos
   if (*p == "a") {
      return temp
   } else if (p == string + MAX) {
      return –1
   } else {
      temp++
      p++
      return scan (p, temp)
   } /* end if */

} /* scan */

scan (string, 0) /* chiamata a scan in main */
```

punto 1 – codice MIPS della sezione dichiarativa globale (num. righe non signif.)
`        .data                    // seg. dati statici - 0x 1000 0000`
`        .eqv   MAX,  5           // dichiarazione della costante MAX`
`STRING: .space 10                // spazio per varglob STRING (non iniz.)`

La costante simbolica MAX è dichiarata per comodità espressiva, ma è sempre sostituibile con il relativo valore numerico.

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

contenuto simbolico	indirizzo	spiazzamento rispetto a *gp*	
			indirizzi alti
STRING [4]	*0x 1000 0004*	*0x 8004*	
STRING [3]	*0x 1000 0003*	*0x 8003*	
STRING [2]	*0x 1000 0002*	*0x 8002*	
STRING [1]	*0x 1000 0001*	*0x 8001*	
STRING [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 ***SCAN*** (numero di righe non significativo)

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

Il registro ra DEVE essere salvato nell'area di attivazione in pila, poiché la funzione SCAN è ricorsiva (e quindi NON è di tipo foglia). La variabile locale TEMP (intero) viene allocata nel registro s0, poiché non occorre che abbia un indirizzo esplicito.

allocazione delle variabili locali di *scan* nei registri	
variabile locale	**registro**
TEMP (intero)	*s0*

La funzione SCAN utilizza il registro "callee-saved" s0 per la sua variabile locale, e dunque tale registro va salvato (da parte della funzione) all'inizio e ripristinato (da parte della funzione) alla fine.

punto 3 – rispondere sinteticamente alle domande seguenti (nello spazio assegnato)

Qual è il valore dei due argomenti della funzione *SCAN* al momento della chiamata in *MAIN* ?

Argomento p: base del vettore STRING, ossia l'indirizzo 0x 1000 0000. Argomento pos: ovviamente l'intero 0, ossia 0x 0000 0000.

La funzione *SCAN* ha due argomenti: dove vengono allocati ?

I due argomenti della funzione SCAN sono allocati nei registri $a0 (arg p) e $a1 (arg pos). Poiché la funzione SCAN ha meno di quattro argomenti, non occorre allocare argomenti in pila.

La funzione *SCAN* è ricorsiva: deve salvare in pila i propri argomenti ?

La funzione SCAN può modificare l'argomento p. Tuttavia essa non fa uso di tale argomento dopo la chiamata ricorsiva, e dunque non si aspetta di ritrovarlo immodificato dal chiamato., pertanto non deve salvare l'argomento p (reg $a0). La funzione SCAN non modifica l'argomento pos (reg $a1), non lo usa dopo la chiamata ricorsiva e a maggior ragione non lo deve salvare.

punto 4 – codice MIPS di ***SCAN*** **- solo blocco nel riquadro** (num. righe non signif.)

```
addi    $s0, $s0, 1        // calcola temp++
addi    $a0, $a0, 1        // calcola p++
move    $a1, $s0           // carica arg temp di scan
jal     SCAN               // chiama funz scan
lw      $s0, 0($sp)        // ripristina reg s0
lw      $ra, 4($sp)        // ripristina reg ra
addiu   $sp, $sp, -4       // elimina area
jr      $ra                // rientra a chiamante
```

seconda parte – processo di assemblaggio

Dati i due moduli assemblatore seguenti, **si compilino** le quattro tabelle seguenti relative a:

1. i due moduli oggetto prodotti dall'assemblatore
2. le basi di rilocazione del codice e dei dati dei moduli (per ragioni di impaginazione la tabella 2 compare prima della tabella 1)
3. la tabella globale dei simboli
4. il contenuto del file eseguibile prodotto dal collegatore (linker)

modulo MAIN	modulo PROC

<pre>   .data
NUM: .word   50
VALUE: .space   4
ADDR: .space   4
        .text
        .globl   MAIN
MAIN: jal   PROC
        lw $t0, NUM
        sw $t0, VALUE
        la $t1, NUM
        sw $t1, ADDR
MEND: syscall</pre>
	<pre> .data
	A: .word 3
	B: .word 7
	        .text
	        .globl PROC
	PROC: lw $t3, A
	        addi $t4, $0, 50
	        beq $t3, $t4, PEND
	        sw $t3, B
	PEND: jr $ra</pre>

Si noti che "**la**" è una **pseudo-istruzione**, da espandere in istruzioni macchina native, come noto.

Regola generale per la compilazione di **tutte** le tabelle contenenti codice:

- i codici operativi e i nomi dei registri vanno indicati in formato simbolico
- tutte le costanti numeriche all'interno del codice vanno indicate in esadecimale, senza prefisso 0x, e di lunghezza giusta per il codice che rappresentano

  esempio: un'istruzione come ***addi $t0, $t0, 15*** è rappresentata come ***addi $t0, $t0, 000F***

- nei moduli oggetto i valori numerici che non possono essere indicati poiché dipendono dalla rilocazione successiva, vanno posti a zero e avranno un valore definitivo nel codice eseguibile

  esempio: ***lw $t0, RIL*** diventa ***lw $t0, 0000($gp)*** nell'oggetto e diventa ***lw $t0, nnnn($gp)*** nell'eseguibile, dove *RIL* è un simbolo rilocabile che si riferisce a un dato globale e *nnnn* è il valore opportunamente calcolato durante la generazione dell'eseguibile

(2) – posizione in memoria dei moduli	
MAIN	***PROC***
base del testo: *0040 0000*	base del testo: *0040 001C*
base dei dati: *1000 0000*	base dei dati: *1000 000C*

È noto che la pseudo-istruzione "la $t1, NUM" si può espandere in due istruzioni native:

```
lui  $t1, NUM
addi $t1, $t1, NUM        (oppure ori, per l'indirizzamento non cambia niente)
```

Pertanto la pseudo-istruzione si espande in due istruzioni native. Dunque il modulo MAIN a sette istruzioni (mentre il modulo PROC ne ha cinque). Ora è facile calcolare gli indirizzi di impianto dei moduli concatenati: prima MAIN poi PROC.

(1) – moduli oggetto

modulo *MAIN*

dimensione testo: 1C hex (28 dec)

dimensione dati: C hex (12 dec)

testo

indirizzo	istruzione
0	**jal** 0000000
4	**lw** $t0, 0000($gp)
8	**sw** $t0, 0000($gp)
C	**lui** $t1, 0000
10	**addi** $t1, $t1, 0000
14	**sw** $t1, 0000($gp)
18	**syscall**

dati

indirizzo	contenuto (in esadecimale)
0	0032
4	non inizializzato
8	non inizializzato

tabella dei simboli
tipo può essere *T* (testo) oppure *D* (dato)

simbolo	tipo	valore
MAIN	T	0000 0000
MEND	T	0000 0018
NUM	D	0000 0000
VALUE	D	0000 0004
ADDR	D	0000 0008

tabella di rilocazione

indirizzo	cod. operativo	simbolo
0	**jal**	PROC
4	**lw**	NUM
8	**sw**	VALUE
C	**lui**	NUM
10	**addi**	NUM
14	**sw**	ADDR

modulo *PROC*

dimensione testo: 14 hex (20 dec)

dimensione dati: 8 hex (8 dec)

testo

indirizzo	istruzione
0	**lw** $t3, 0000($gp)
4	**addi** $t4, $0, 0032
8	**beq** $t3, &t4, 0001
C	**sw** $t3, 0000($gp)
10	**jr** $ra

dati

indirizzo	contenuto (in esadecimale)
0	0003
4	0007

tabella dei simboli
tipo può essere *T* (testo) oppure *D* (dato)

simbolo	tipo	valore
PROC	T	0000 0000
PEND	T	0000 0010
A	D	0000 0000
B	D	0000 0004

tabella di rilocazione

indirizzo	cod. operativo	simbolo
0	**lw**	A
C	**sw**	B
8	**beq**	PEND

(3) – tabella globale dei simboli			
simbolo	**valore iniziale**	**base**	**valore finale**
MAIN	*0000 0000*	*0040 0000*	*0040 0000*
MEND	*0000 0018*	*0040 0000*	*0040 0018*
NUM	*0000 0000*	*1000 0000*	*1000 0000*
VALUE	*0000 0004*	*1000 0000*	*1000 0004*
ADDR	*0000 0008*	*1000 0000*	*1000 0008*
PROC	*0000 0000*	*0040 001C*	*0040 001C*
PEND	*0000 0010*	*0040 001C*	*0040 002C*
A	*0000 0000*	*1000 000C*	*1000 000C*
B	*0000 0004*	*1000 000C*	*1000 0010*

(4) – codice eseguibile	
testo	
indirizzo	**codice** (con codici operativi e registri in forma simbolica)
0040 0000	***jal*** *040 0007 // bin 26 bit: 00010000000000000000000111*
0040 0004	***lw*** *$t0, 8000($gp) // spi. negativo di NUM con base gp*
0040 0008	***sw*** *$t0, 8004($gp) // spi. negativo di NUM con base gp*
0040 000C	***lui*** *$t1, 1000 // parte hi di NUM*
0040 0010	***addi*** *$t1, $t1, 0000 // parte lo di NUM*
0040 0014	***sw*** *$t1, 8008($gp) // spi. Neg. di ADDR con base gp*
0040 0018	***syscall*** *// non contiene simboli*
0040 001C	***lw*** *$t3, 800C($gp) // spi. negativo di A con base gp*
0040 0020	***addi*** *$t4, $0, 0032 // parte lo di cost. 50*
0040 0024	***beq*** *$t3, &t4, 0001 // distanza di salto in # di istr.*
0040 0028	***sw*** *$t3, 8010($gp) // spi. negativo di B con base gp*
0040 002C	***jr*** *$ra // non contiene simboli*

Non si richiede la tabella globale dei dati.

esercizio n. 2 – struttura e controllo del processore

prima parte – struttura del processore

Riferimento: processore pipeline e campi registri interstadio.

Sono dati il seguente frammento di codice macchina *MIPS* (simbolico), che comincia l'esecuzione all'indirizzo indicato, i valori iniziali specificati per alcuni registri, e il contenuto (e relativo indirizzo) di alcune parole della memoria dati.

indirizzo	codice assemblatore
0x 0000 0880	sw $t2, 0x0080($t1)
	add $t2, $t2, $t1
	lw $t2, 0x000A($t2)
	beq $t2, $t1, 64
	sw $t1, 0x000C($t2)

registro	contenuto iniziale
$t0	0x000F C422
$t1	0x0C04 08C0
$t2	0x0040 0800

memoria dati	
indirizzo	**parola**
0x 001F C4CC	0x AAAA 2001
0x 0040 080A	0x 0C04 08C0
0x 0040 08A0	0x FFFF AAAA
0x 0C04 08CC *(ind seconda sw)*	0x 000F C400
0x 0C04 0940 *(ind prima sw)*	0x FEEE ACAC
0x 0C44 10CA	0x 000A 0FF0

Si consideri il ciclo di clock in cui l'esecuzione delle istruzioni nei vari stadi è la seguente:

IF	*istruzione non specificata*
ID	sw $t1, 0x000C($t2)
EX	beq $t2, $t1, 64
MEM	lw $t2, 0x000A($t2)
WB	add $t2, $t2, $t1

1) **Indicare** qual è il ciclo di clock considerato: ________ *6*

2) **Calcolare** il valore di (t1) + (t2):________ *0x 0C44 10C0 (0C04 08C0 + 0040 0800)*

3) **Calcolare** l'indirizzo della parola di memoria referenziata da *lw*:*0x 0040 080A (0040 0800 + 0000 000A)*

4) **Calcolare** l'indirizzo della destinazione di salto: ________*0x 0000 0990(0000 0890 + 0100)*

5) **Completare** le tabelle della pagina seguente:

I campi Istruzione e di tipo NumeroRegistro possono essere indicati in forma simbolica, tutti gli altri in esadecimale (omettendo il prefisso 0x implicito).

segnali all'ingresso dei registri di interstadio (subito prima del fronte di SALITA del clock)			
IF	**ID** *sw_2*	**EX** *beq*	**MEM** *lw*
registro IF/ID	***registro ID/EX***	***registro EX/MEM***	***registro MEM/WB***
	.WB.MemtoReg *X*	.WB.MemtoReg *X*	.WB.MemtoReg *1*
	.WB.RegWrite *0*	.WB.RegWrite *0*	.WB.RegWrite *1*
	.M.MemWrite *1*	.M.MemWrite *0*	
	.M.MemRead *0*	.M.MemRead *0*	
	.M.Branch *0*	.M.Branch *1*	
.PC *0000 0898*	.PC *0000 0894*	.PC *0000 0990 (ind di sw_2 + 64x4)*	
.istruzione *non specificata*	.(Rs) *0040 0800 (val iniz t2)*		
	.(Rt) *0C04 08C0 (val iniz t1)*	.(Rt) ***************	
	.Rt *09 ossia t1*	.R ***************	.R *0A ossia t2*
	.Rd ***************		
	.imm/offset esteso *0000 000C*	.ALU_out *t2 iniz – t1 iniz (= rs–rt)*	.ALU_out *0040 080A (ind mem lw)*
	.EX.ALUSrc *1*	.Zero *0*	.DatoLetto *0C04 08C0*
	.EX.RegDest *X*		

segnali relativi al RF (subito prima del fronte di DISCESA interno al ciclo di clock)		
RF.RegLettura1 *0A (t2)*	RF.RegScrittura *0A (t2)*	RF.DatoLetto1 *0040 0800 (t2 iniziale)*
RF.RegLettura2 *09 (t1)*	RF.DatodaScrivere *0C44 10C0 (calcolato da add)*	RF.DatoLetto2 *0C04 08C0 (t1 iniziale)*

segnali di altre unità funzionali (subito prima del fronte di SALITA del clock)	
Mem.indirizzo ********************************	RegWrite ********************************
MemWrite ********************************	RegDest ********************************
	MemtoReg ********************************

seconda parte – gestione di conflitti e stalli

Si consideri la seguente sequenza di istruzioni appartenente al codice precedente (di cui viene fornito per comodità di analisi anche il diagramma multiciclo teorico).

```
1.  add  $2, $2, $1
2.  lw   $2, 0x000A($2)
3.  beq  $2, $1, 64
```

ciclo di clock

istruzione	1	2	3	4	5	6	7	8	9	10	11	12	13
1	IF	ID 2,1	EX	MEM	WB 2								
2		IF	ID 2,2	EX	MEM	WB 2							
3			IF	ID 2,1	EX	MEM	WB						

Si risponda alle domande seguenti, facendo l'ipotesi che la pipeline abbia **esclusivamente** dei cammini di propagazione **EX/EX** e **MEM/EX**:

a) **si riportino** in **tabella 1** – prime tre colonne – le dipendenze di dato che risultano essere **conflitti**
b) **si disegni** in **Tabella 2** 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
c) **si indichino** in **Tabella 1** – ultima colonna – i cammini di propagazione **effettivamente** attivati e gli stalli associati

Tabella 1 (numero di righe non significativo)

istruzione	istruzione da cui dipende	registro coinvolto	cammino di propagazione e stalli
2	*1*	*$2*	*EX-EX*
3	*1*	*$2*	*nessuno*
3	*2*	*$2*	*MEM-EX + 1 Stallo*

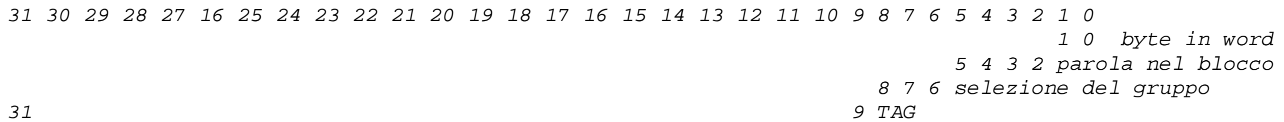
Tabella 2

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

esercizio n. 3 –memoria cache

Un sistema basato su MIPS adotta una cache **set-associative** a **quattro vie**. Ogni blocco è da **64 byte** (16 parole). La cache contiene **32 blocchi** in totale e di conseguenza ha **8 gruppi**. I quattro blocchi nel gruppo sono identificati dalle lettere A, B, C e D. Si ricordi che il processore MIPS indirizza il byte.

Considerando i 32 bit dell'indirizzo generato dal processore MIPS, **si identifichi** in essi la posizione dei bit usati per selezionare il byte nella parola, la parola nel blocco e il gruppo nella cache, e poi **si indichino** quelli che costituiscono l'etichetta (TAG).



```
31 30 29 28 27 16 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
                                                                                1 0  byte in word
                                                                        5 4 3 2 parola nel blocco
                                                                  8 7 6 selezione del gruppo
31                                                              9 TAG
```

Si immagini che MIPS **esegua in sequenza** l'accesso a tutti gli indirizzi da 0x ABCD 0000 a 0x ABCD 08BF estremi inclusi (lettura oppure scrittura di UN SINGOLO BYTE a ogni accesso).

Quanti sono questi accessi ai byte ?

da 0x 000 incluso a 0x 8BF incluso, ossia 8BF + 1, ossia 2239 + 1 = 2240

E a quante parole si accede ?

2240 : 4 = 560

Quanti blocchi corrispondono a questa sequenza di accessi ai byte ?

560 : 16 = 35 (= 0x 23)

Identificare i blocchi con 0, 1, 2, ... in esadecimale: da ________ *0, 1, 2, ...*fino a ________ *1F, 20, 21, 22*

Nota bene: per come li abbiamo costruiti, e poiché la sequenza di accessi parte dall'indirizzo 0x 0000, questi stessi numeri esadecimali costituiscono i bit 11 10 9 8 7 6 degli indirizzi generati dalla CPU durante gli accessi alla memoria; quindi i tre bit meno significativi di questi numeri identificanti il blocco selezionano anche il gruppo in cui il blocco deve finire in cache. Esempio: il blocco che chiamiamo 0x 1A, cioè 0001 1010, ha i tre bit meno significativi 010 e quindi deve finire nel gruppo 010.

Si supponga che all'inizio la cache sia vuota, e che la sequenza di accessi venga **eseguita due volte di seguito.** Mostrare come i blocchi si inseriscano nella cache, riempiendo con i simboli dei blocchi 0, 1, 2 ... le caselle della tabella sottostante.

Se in una casella un blocco ne sostituisce un altro, **lasciarli** indicati entrambi, **cancellando** con un tratto di penna quello sostituito (politica: LRU; se un gruppo ha più blocchi vuoti, li si riempia nell'ordine A, B, C e D).

	A	B	C	D
000				
001				
010				
011				
100				
101				
110				
111				

	A	B	C	D
000	~~0~~ ~~20~~ 18	~~8~~ ~~0~~ 20	~~10~~ 8	~~18~~ 10
001	~~1~~ ~~21~~ 19	~~9~~ ~~1~~ 21	~~11~~ 9	~~19~~ 11
010	~~2~~ ~~22~~ 1A	~~A~~ ~~2~~ 22	~~12~~ A	~~1A~~ 12
011	3	B	13	1B
100	4	C	14	1C
101	5	D	15	1D
110	6	E	16	1E
111	7	F	17	1F

Si noti bene: gli otto gruppi sono selezionati con i tre bit dell'indirizzo la cui posizione è stata trovata rispondendo alla prima domanda. Per come abbiamo deciso di identificare con un numero esadecimale ogni blocco, questi tre bit sono anche i tre bit meno significativi del numero esadecimale che identifica ogni blocco.

Prima sequenza:

blocchi da 0 a 1F: la cache si riempie ordinatamente dall'alto in basso e da sinistra a destra

blocchi da 20 a 22: la cache è piena, i tre bit di selezione dei gruppi selezionano ordinatamente le prime tre righe, si sceglie la prima colonna (prima via) che è la LRU attuale, quindi i blocchi 0 1 2 sono sostituiti

Seconda sequenza:

blocchi 0 1 2: non ci sono più nella cache, ossia sono MISS, devono essere piazzati ordinatamente nei primi tre gruppi, gli LRU sono adesso nella via B, i blocchi 8 9 A sono sostituiti

blocchi 3- 7: HIT

blocchi 8 9 A: non ci sono più: MISS, vanno messi nelle prime tre righe della via C, che ora sono le LRU

... e così via ... vedi come è riempita la tabella

Riassumendo:

Quanti sono i MISS ? ______________________________

I MISS avvengono al primo accesso di un byte del blocco; gli accessi ai restanti byte del blocco non causano MISS. Quindi, un solo eventuale MISS per blocco. Ecco i blocchi che causano MISS:

prima sequenza: blocchi da 0 a 1F per riempimento, + blocchi 20, 21, 22 per sostituzione, ossia 35 MISS

seconda sequenza: blocchi 0, 1, 2 e 8, 9, A e 10, 11, 12 e 18, 19, 1A e 20, 21, 22 tutti per sostituzione, ossia 15 MISS

Totale: 35+15=50 MISS

Quanti sono gli accessi alla memoria ?

____

abbiamo già visto che sono 2240

Quanto vale il MISS RATE ?

50 : 2240 = 0,02232... ossia circa 2,23 %

Quanto vale lo HIT RATE?

1 - 0,02232... = 0,97767... ossia circa 97,77 %

esercizio n. 4 – processi e parallelismo

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

```
pthread_mutex_t door = PTHREAD_MUTEX_INITIALIZER
sem_t new
int global = 1

void * one (void * arg) {
   pthread_mutex_lock (&door)
   sem_post (&new)
   global++                                /* statement A */
   pthread_mutex_unlock (&door)
   return NULL
} /* end one */

void * two (void * id) {
   int done = 0
   pthread_mutex_lock (&door)              /* statement B */
   sem_wait (&new)
   pthread_mutex_unlock (&door)
   done = 3                                /* statement C */
   sem_post (&new)
   global = global + (int) id
   return NULL
} /* end two */

void main ( ) {
   pthread_t th_0, th_1, th_2
   sem_init (&new, 0, 0)
   pthread_create (&th_0, NULL, one, NULL)
   pthread_create (&th_1, NULL, two, (void *) 1)
   pthread_create (&th_2, NULL, two, (void *) 2)
   pthread_join (th_0, NULL)
   pthread_join (th_2, NULL)               /* statement D */
   pthread_join (th_1, NULL)
   return
} /* end main */
```

Si completi la tabella qui sotto **indicando lo stato di esistenza della variabile locale** nell'istante di tempo specificato da ciascuna condizione, così: se la variabile **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).

condizione	**variabile locale**	
	***done* in th_1**	***done* in th_2**
subito dopo stat. **A**	*PUÒ ESISTERE*	*PUÒ ESISTERE*
subito dopo stat. **C** in **th_1**	*ESISTE*	*PUÒ ESISTERE*
subito dopo stat. **D**	*PUÒ ESISTERE*	*NON ESISTE*

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; 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)

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**	
	new	*global*
subito dopo stat. **A**	*1*	*2*
subito dopo stat. **B** in **th_1**	*0 / 1*	*1 / 2 / 4*
subito dopo stat. **C** in **th_2**	*0*	*2 / 3*

Il sistema può andare in stallo (deadlock), con uno o più thread che si bloccano. Si indichino gli statement dove avvengono i blocchi.

caso	**th_0**	**th_1**	**th_2**
1	*lock door*	*wait new*	*prima di lock o a lock*
2	*lock door*	*prima di lock o a lock*	*wait new*

esercizio n. 5 – gestione della memoria

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

domanda 1 – si sono verificati i seguenti eventi:

- il processo **P** esegue un servizio *exec* di un programma che inizia nella seconda pagina di codice (NPV = **pc1**),
- il processo P alloca e scrive due pagine di memoria dinamica
- il processo P scrive nella prima pagina di dati statici
- successivamente il processo P entra in un ciclo nel quale accede continuamente alle pagine **pc1** e **pp0**; durante l'esecuzione di tale ciclo il daemon *kswapd* interviene quattro volte

Si mostrino lo stato della memoria e dello swap file, 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	*pc1*	**1**	*pp0*	**2**	*pd0*	**3**	*pd1*
4	*ps0*	**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	*PC1 PP0*	*ps0 pd1 pd0*

TLB (solo righe di modo *U*)

(completare inserendo le pagine in ordine virtuale di pagina)

NPV	***NPF***	***D***	***A***	***NPV***	***NPF***	***D***	***A***
pc1	*0*		*1*	*pp0*	*1*	*1*	*1*
pd0	*2*	*1*	*0*	*pd1*	*3*	*1*	*0*
ps0	*4*	*1*	*0*				

domanda 2 – si sono verificati i seguenti eventi:

- il processo P ha eseguita una *fork* creando il processo Q

Si mostrino lo stato della memoria e dello swap file, e lo stato delle liste LRU.

memoria fisica (solo modo *U*)							
NPF	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***
0	*pc1 / qc1*	**1**	*qp0*	**2**	*pd0 / qd0*	**3**	*pd1 / qd1*
4	*ps0 / qs0*	**5**	*pp0*	**6**		**7**	
swap file							

liste LRU		
attivazioni	***active***	***inactive***
iniziale	*PC1 PP0*	*ps0 pd1 pd0*
finale	*QC1 QP0 PC1 PP0*	*qs0 qd0 qd1 ps0 pd1 pd0*

domanda 3 – si sono verificati i seguenti eventi:

- il processo P ha letta la seconda pagina (**ps1**) dei dati statici

Si mostrino lo stato della memoria e dello swap file, e lo stato delle liste LRU (attenzione – la nuova pagina allocata viene posta in testa alla active list con ref =1).

è richiesta una nuova pagina, sono libere solo MINFREE (due) pagine, quindi si invoca PFRA per liberare due pagine riportando il numero di pagine libere a MAXFREE

memoria fisica (solo modo *U*)							
NPF	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***
0	*pc1 / qc1*	**1**	*qp0*	**2**	*ps1*	**3**	
4	*ps0 / qs0*	**5**	*pp0*	**6**		**7**	
swap file	*pd0 / qd0 pd1 / qd1*						

liste LRU		
attivazioni	***active***	***inactive***
iniziale	*QC1 QP0 PC1 PP0*	*qs0 qd0 qd1 ps0 pd1 pd0*
finale	*PS1 QC1 QP0 PC1 PP0*	*qs0 ps0*

domanda 4 – si sono verificati i seguenti eventi:

- il processo P ha eseguita *exit* e il processo Q è stato messo in esecuzione

Si mostrino lo stato della memoria e dello swap file, e lo stato delle liste **LRU**.

memoria fisica (solo modo *U*)							
NPF	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***	***NPF***	***NPV***
0	*qc1*	**1**	*qp0*	**2**		**3**	
4	*qs0*	**5**		**6**		**7**	
swap file	*qd0 qd1*						

liste LRU		
attivazioni	***active***	***inactive***
iniziale	*PS1 QC1 QP0 PC1 PP0*	*qs0 ps0*
finale	*QC1 QP0*	*qs0*

esercizio n. 6 – nucleo e componenti del SO

prima parte – scheduling dei processi

```
// programma prova.c


pthread_mutex_t GATE = PTHREAD_MUTEX_INITIALIZER
sem_t GO

void * A (void * arg) {
(1) pthread_mutex_lock (&GATE)
(2) sem_wait (&GO)
(3) pthread_mutex_unlock (&GATE)
    return NULL
}  // thread A

void * B (void * arg) {
 (4) pthread_mutex_lock (&GATE)
 (5) pthread_mutex_unlock (&GATE)
 (6) sem_wait (&GO)
 (7) sem_post (&GO)
     return NULL
}  // thread B

main ( ) {  // codice eseguito da P

    pthread_t TH_A, TH_B
    sem_init (&GO, 0, 1)
    pthread_create (&TH_B, NULL, B, NULL)
    pthread_create (&TH_A, NULL, A, NULL)
    read (stdin, vett, 1)
    pthread_join (&TH_A, NULL)
(8) sem_post (&GO)
    pthread_join (&TH_B, NULL)
    exit (1)
}  // main
```

Un processo ***P*** esegue il programma ***prova*** creando i thread ***TH_A*** e ***TH_B***.

Si simuli l'esecuzione dei processi (fino a ***udt*** = **150**) così come risulta dal codice dato, dagli eventi indicati e facendo bene attenzione allo stato iniziale considerato per la simulazione.

Domanda 1)

Si completi la tabella riportando quanto segue:

- ⟨ *PID*, *TGID* ⟩ di ogni processo che viene creato
- ⟨ *identificativo del processo-chiamata di sistema / libreria* ⟩ nella prima colonna, dove è necessario e in funzione del codice proposto
- in ciascuna riga lo stato dei processi **al termine del tempo indicato**; si noti che alcune righe della tabella possono essere già parzialmente completate

NOTA BENE:

- si considerino **solo le funzioni marcate in grassetto** nel codice dato
- l'esecuzione della funzione *exec* **non sospende il processo**

TABELLA DA COMPILARE (numero di colonne non significativo)

identificativo simbolico del processo		**IDLE**	**P**	*B*	*A*			
	PID	**1**	*2*	*3*	*4*			
evento/processo-chiamata	*TGID*	**1**	*2*	*2*	*2*			
P - sem_init	**0**	**pronto**	**esec**	*NE*	*NE*			
P – create B	10	*pronto*	*esec*	*pronto*	*NE*			
interrupt da RT_clock, scadenza quanto di tempo	20	*pronto*	*pronto*	*esec*	*NE*			
B - lock	30	*pronto*	*pronto*	*esec*	*NE*			
interrupt da RT_clock, scadenza quanto di tempo	40	*pronto*	*esec*	*pronto*	*NE*			
P – create A	50	*pronto*	*esec*	*pronto*	*pronto*			
P – read	60	*pronto*	attesa *(read)*	*esec*	*pronto*			
interrupt da RT_clock, scadenza quanto di tempo	70	*pronto*	*attesa (read)*	*pronto*	*esec*			
A - lock	80	*pronto*	*attesa (read)*	*esec*	*attesa (lock)*			
B - unlock	90	*pronto*	*attesa (read)*	*pronto*	*esec*			
A – sem_wait	100	*pronto*	*attesa (read)*	pronto	esec			
A - unlock	110	*pronto*	*attesa (read)*	*pronto*	*esec*			
interrupt da stdin, tutti i caratteri richiesti trasferiti	120	*pronto*	*esec*	*pronto*	*pronto*			
P – join A	130	*pronto*	*attesa (join)*	*esec*	*pronto*			
B – sem_wait	140	*pronto*	*attesa (join)*	*attesa (sem_wait)*	*esec*			
A - return	150	*pronto*	*esec*	*attesa (sem_wait)*	*NE*			
	160							
	170							

Domanda 2)

Si considerino le chiamate in ***main***, ***A*** e ***B*** contrassegnate dai numeri d'ordine da **1** a **8**. Con riferimento alla loro implementazione tramite *futex* e alla simulazione riportata in tabella, **si indichino** quelle eseguite:

- senza invocare *system_call*: *4, 2, 3*

- con invocazione di *system_call*: *1, 5, 6*