1 Deadlock

Indicare se il seguente programma può generare situazioni di deadlock e, in caso affermativo, indicare una sequenza di istruzioni che porta al deadlock con il relativo grafo di attesa.

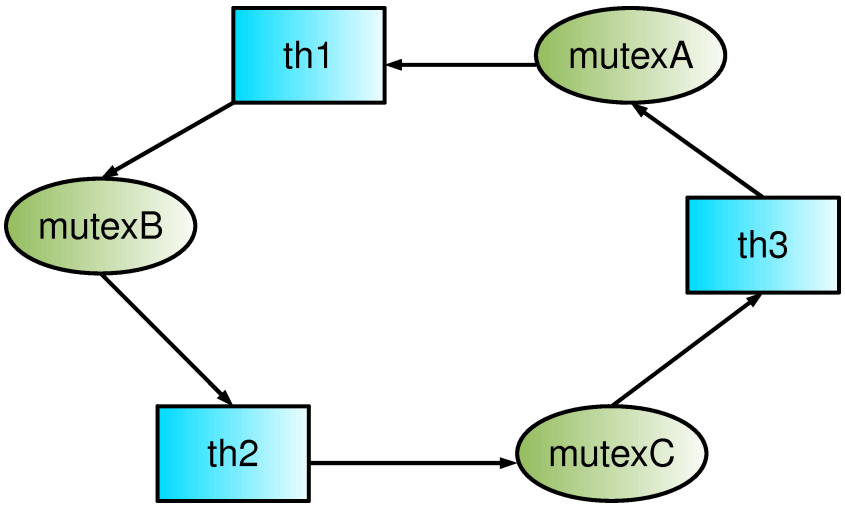
```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexB = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexC = PTHREAD_MUTEX_INITIALIZER;

void * thread1( void *arg ){
  pthread_mutex_lock( &mutexA );
  pthread_mutex_lock( &mutexB );
  printf( "thread1\n" );
  pthread_mutex_unlock( &mutexB );
  pthread_mutex_unlock( &mutexA );
  return NULL;
}

void * thread2( void *arg ){
  pthread_mutex_lock( &mutexB );
  pthread_mutex_lock( &mutexC );
  printf( "thread2\n" );
  pthread_mutex_unlock( &mutexC );
  pthread_mutex_unlock( &mutexB );
  return NULL;
}

void * thread3( void *arg ){
  pthread_mutex_lock( &mutexC );
  pthread_mutex_lock( &mutexA );
  printf( "thread3\n" );
  pthread_mutex_unlock( &mutexA );
  pthread_mutex_unlock( &mutexC );
  return NULL;
}

int main(int argc , char * argv []){
  pthread_t th1 , th2, th3;
  pthread_create( &th1, NULL, thread1, NULL);
  pthread_create( &th2, NULL, thread2, NULL);
  pthread_create( &th3, NULL, thread3, NULL);
  pthread_join( th1, NULL);
  pthread_join( th2, NULL);
  pthread_join( th3, NULL);
  return 0;
}
```

Soluzione

Se i tre thread arrivano a bloccare un mutex ciascuno allora sicuramente si svilupperà un situazione di deadlock. Una delle possibili sequenze di istruzioni che porta al deadlock è la seguente:

1. `th1` ottiene la risorsa mutexA con l'istruzione: pthread_mutex_lock(&mutexA);
2. `th2` ottiene la risorsa mutexB con l'istruzione: pthread_mutex_lock(&mutexB);
3. `th1` attende la risorsa mutexB con l'istruzione: pthread_mutex_lock(&mutexB);
4. `th3` ottiene la risorsa mutexC con l'istruzione: pthread_mutex_lock(&mutexC);
5. `th2` attende la risorsa mutexC con l'istruzione: pthread_mutex_lock(&mutexC);
6. `th3` attende la risorsa mutexA con l'istruzione: pthread_mutex_lock(&mutexA);

Il relativo grafo di attesa è riportato nella seguente figura. Il grafo sarà lo stesso per tutte le sequenze che portano ad un deadlock.



esercizio n. 2 – memoria virtuale

Un sistema dotato di memoria virtuale con paginazione e segmentazione di tipo UNIX è caratterizzato dai parametri seguenti: la memoria centrale fisica ha capacità di 32 K byte, quella logica di 32 K byte e la pagina ha taglia di 4 K byte. Si chiede di svolgere i punti seguenti:

(a) **Si definisca** la struttura degli indirizzi fisico e logico indicando la lunghezza dei campi:

NPF: *3 bit* spiazzamento fisico: *12 bit* NPL: *3 bit* spiazzamento logico: *12* bit

(b) **Si inserisca** nella tabella a fianco la struttura in pagine della memoria virtuale di due programmi X e Y (mediante la notazione CX0 CX1 DX0 PX0 ... CY0 ...), sapendo che la dimensione iniziale dei segmenti di tali programmi è la seguente:

CX: 4 K DX:12 K PX:4 K
CY:16 K DY: 4 K PY:4 K

indir. virtuale	prog. X	prog. Y
0	*CX0*	*CY0*
1	*DX0*	*CY1*
2	*DX1*	*CY2*
3	*DX2*	*CY3*
4		*DY0*
5		
6		
7	*PX0*	*PY0*

(c) Nel sistema vengono creati alcuni processi, indicati nel seguito con P, Q, R e S.

A pagina seguente sono indicate due serie di eventi; la prima serie termina all'istante t_0, la seconda all'istante t_1.

Si compilino le tabelle che descrivono i contenuti della memoria fisica e della MMU agli istanti t_0 e t_1, utilizzando la notazione CP0 CP1 DP0 PP0 ... CQ0 ... per indicare le pagine virtuali dei processi; nelle tabelle della MMU si aggiunga anche il NPV effettivo con la notazione CP0/0 ecc ...; in tutte le tabelle si usi la notazione (CP0) ecc ... per indicare un dato non più valido e la notazione CP0 = CQ0 ecc ... per indicare che una pagina fisica contiene più di una pagina logica.

Si considerino valide le seguenti ipotesi relative al sistema:

- il lancio di una programma avviene caricando solamente la pagina di codice con l'istruzione di partenza e una sola pagina di pila
- il caricamento di pagine ulteriori è in Demand Paging (cioè le pagine si caricano su richiesta senza scaricare le precedenti fino al raggiungimento del numero massimo di pagine residenti)
- l'indirizzo (esadecimale) dell'istruzione di partenza di X è 0AAA
- l'indirizzo (esadecimale) dell'istruzione di partenza di Y è 39F2
- il numero di pagine residenti **R** vale **4**
- viene utilizzato l'algoritmo LRU (ove richiesto prima si dealloca una pagina di processo e poi si procede alla nuova assegnazione)
- in assenza di indicazioni esplicite relative all'accesso alle pagine, le pagine meno utilizzate in ogni processo sono quelle caricate da più tempo; inoltre la pagina di codice caricata più di recente è acceduta continuamente
- al momento di una "fork" viene duplicata solamente la pagina di pila caricata più recentemente, ma tutte le pagine virtuali del padre sono considerate condivise con il figlio
- dopo una "fork", se uno dei due processi padre o figlio scrive in una pagina condivisa, la nuova pagina fisica che viene allocata appartiene al processo che ha eseguito la scrittura
- l'allocazione delle pagine virtuali nelle pagine fisiche avviene **sempre** in sequenza, senza buchi, a partire dalla pagina fisica 0
- le righe della tabella della MMU vengono allocate ordinatamente man mano che vengono allocate le pagine di memoria virtuale
- gli eventi influenzanti la MMU partono da una situazione di tabella vergine
- se è richiesta una nuova riga di MMU, si utilizza **sempre** la prima riga libera

(d) A un certo istante di tempo t_0 sono terminati, nell'ordine, gli eventi seguenti:

1. creazione del processo P e lancio del programma X ("fork" di P ed "exec" di X)
2. P accede a pagine nel seguente ordine: dati 1, dati 0, pila
3. P crea due pagine dati dinamiche tramite BRK
4. creazione del processo Q e lancio del programma Y ("fork" di Q ed "exec" di Y)
5. Q salta all'istruzione di indirizzo 2B35, poi accede alla pila
6. Q crea 2 pagine di pila

Si compilino le tabelle sotto, al tempo t_0.

situazione al tempo t_0

memoria fisica	
indir. fisico	**pagine allocate**
0	*CP0*
1	*PP0*
2	*(DP1) DP4*
3	*(DP0) DP5*
4	*(CQ3) PQ2*
5	*PQ0*
6	*CQ2*
7	*PQ1*

MMU			
proc.	**NPV**	**NPF**	**valid bit**
P	*CP0/0*	*0*	*1*
P	*PP0/7*	*1*	*1*
P	*(DP1/2) DP4/5*	*2*	*1*
P	*(DP0/0 DP5/6*	*3*	*1*
Q	*(CQ3/3) PQ2/5*	*4*	*1*
Q	*PQ0/7*	*5*	*1*
Q	*CQ2/2*	*6*	*1*
Q	*PQ1/6*	*7*	*1*
			0
			0
			0
			0

(e) A un certo istante di tempo $t_1 > t_0$ sono terminati, nell'ordine, gli eventi seguenti:

7. P termina ("exit" di P)
8. Q esegue una fork e crea il processo R
9. R esegue una fork e crea il processo S

Si compilino le tabelle sotto, al tempo t_1.

situazione al tempo t_1

memoria fisica	
indir. fisico	**pagine allocate**
0	*(CP0) PR2*
1	*(PP0) PS2*
2	*(DP1) (DP4)*
3	*(DP0) (DP5)*
4	*(CQ3) PQ2*
5	*PQ0 = PR0 = PS0*
6	*CQ2 = CR2 = CS2*
7	*PQ1 = PR1 = PS1*

MMU			
proc.	**NPV**	**NPF**	**valid bit**
(P) R	*(CP0/0) PR2/5*	*(0) 0*	*(1)(0)1*
(P) R	*(PP0/7) PR0/7*	*(1) 5*	*(1)(0)1*
(P) R	*(DP1/2) (DP4/5) CR2/2*	*(2) 6*	*(1)(0)1*
(P) R	*(DP0/0) (DP5/6) PR1/6*	*(3) 7*	*(1)(0)1*
Q	*(CQ3/3) PQ2/5*	*4*	*1*
Q	*PQ0/7*	*5*	*1*
Q	*CQ2/2*	*6*	*1*
Q	*PQ1/6*	*7*	*1*
S	*PS2/5*	*1*	*(0) 1*
S	*PS0/7*	*5*	*(0) 1*
S	*CS2/2*	*6*	*(0) 1*
S	*PS1/6*	*7*	*(0) 1*