ESERCIZI DI SINCRONIZZAZIONE TRA THREAD A LIVELLO UTENTE

ESERCIZIO SincrThread-1

In un processo P sono definiti i thread MAIN, A e B , realizzati a livello utente con scheduling *round robin* e quanto di tempo di 5 *msec*.
I thread A e B cooperano scambiando messaggi attraverso un buffer condiviso, capace di contenere un solo messaggio. Le parti rilevanti del codice dei thread A e B sono le seguenti:

```
Thread A:
   a.1)      while (not fine) {
   a.2)            <produce  mess>
   a.3)            lock(&BufferVuoto);
   a.4)            <deposita mess nel buffer>
   a.5)            void fun();
   a.6)            unlock(&BufferPieno);
      }

Thread B:
   b.1)      while (not fine) {
   b.2)            lock(&BufferPieno);
   b.3)            <preleva mess dal buffer>
   b.4)            unlock(&BufferVuoto);
   b.5)            <consuma mess>
    }
```

dove `BufferPieno`, `BufferVuoto` e `fine` sono variabili binarie condivise. I protocolli `lock(K)` e `unlock(K)` sono realizzati nel modo seguente:

```
lock(K)                                  unlock(K)
         .........                            .............
loop     TSL K, R1                            MOV K, #1
         JNZ R1, SezCrit                           ...........
         JMP loop
SezCrit  ..........,
```

dove R1 è un registro generale del processore e K è l'indirizzo simbolico della chiave. Si conviene che le sezioni critiche siano accessibili quando la chiave ha il valore 1.

Al tempo T si ha `BufferPieno= 1, BufferVuoto= 0` e `fine= 0` ed è in esecuzione il thread A che esegue la riga a.1). I rimanenti thread sono pronti, con l'ordinamento B → MAIN e il thread B è pronto a eseguire la riga b.1). Il tempo di esecuzione di tutte le righe di codice di A e B è trascurabile, ad eccezione della riga a.5) che richiede 8 msec.
Si chiede il tempo necessario ad A per depositare 2 messaggi, a partire dal tempo T. Si suppone che la variabile *fine* conservi il valore 0 e che il thread MAIN consumi sempre per intero il suo quanto di tempo.

SOLUZIONE

1) Primo messaggio depositato al tempo T+ ..... Infatti

- Thread A esegue a.1), a.2) e poi a.3) per 5 msec;
- Thread ..... esegue .... per ..... msec
- Thread ..... esegue .... per ..... msec;
- Thread ....
- Thread ....

2) Secondo messaggio depositato al tempo T+ ..... Infatti

- Thread ..... esegue .... per ..... msec
- Thread ..... esegue .... per ..... msec
- Thread ..... esegue .... per ..... msec
- Thread ..... esegue .... per ..... msec
- Thread ..... esegue .... per ..... msec
- Thread ....
- Thread ....
- Thread ....

ESERCIZIO SincrThread-2

In un processo P sono definiti il thread MAIN, A e B , realizzati a livello utente con scheduling *round robin* e quanto di tempo di 5 *msec*.
I thread A e B cooperano scambiando messaggi attraverso un buffer condiviso, capace di contenere un solo messaggio. Le parti rilevanti del codice dei thread A e B sono le seguenti:

```
Thread A:
   a.1)      while (not fine) {
   a.2)            <produce  mess>
   a.3)            lock(&BufferVuoto);
   a.4)            <deposita mess nel buffer>
   a.5)            void fun();
   a.6)            unlock(&BufferPieno);
      }

Thread B:
   b.1)      while (not fine) {
   b.2)            lock(&BufferPieno);
   b.3)            <preleva mess dal buffer>
   b.4)            unlock(&BufferVuoto);
   b.5)            <consuma mess>
    }
```

dove `BufferPieno`, `BufferVuoto` e `fine` sono variabili binarie condivise.
Per limitare la durata dell'attesa attiva dei thread nell'esecuzione della `thread_lock` i protocolli `thread_lock` e `thread_unlock` utilizzano la funzione `thread_yield` per rilasciare il processore. La loro realizzazione è la seguente:

```
thread_lock(K)
         .........
loop     TSL K, R1
         JNZ R1, SezCrit
         CALL thread_yield
         JMP loop
SezCrit .......
```

```
thread_unlock(K)
     ............
     MOV K, #1
CALL thread_yield
     ...........
```

dove R1 è un registro generale del processore, K è l'indirizzo simbolico della chiave e la funzione `thread_yield` provoca il rilascio del processore da parte del thread che la esegue. Si conviene che le sezioni critiche siano accessibili quando la chiave ha il valore 1.
Al tempo T si ha `BufferPieno= 1, BufferVuoto= 0` e `fine= 0` ed è in esecuzione il thread A che esegue la riga a.1). I rimanenti thread sono pronti, con l'ordinamento B → MAIN e il thread B è pronto a eseguire la riga b.1). Il tempo di esecuzione di tutte le righe di codice di A e B è trascurabile, ad eccezione della riga a.5) che richiede 8 msec.
Si chiede il tempo necessario ad A per depositare 2 messaggi, a partire dal tempo T. Si suppone che la variabile *fine* conservi il valore 0 e che il thread MAIN consumi sempre per intero il suo quanto di tempo.

SOLUZIONE

1) Primo messaggio depositato al tempo T+ ..... Infatti

- Thread A esegue a.1), a.2) e poi a.3) per 0 msec;
- Thread ..... esegue .... per ..... msec
- Thread ..... esegue .... per ..... msec;
- Thread ....

2) Secondo messaggio depositato al tempo T+ ..... Infatti

- Thread ..... esegue .... per ..... msec
- Thread ..... esegue .... per ..... msec
- Thread ..... esegue .... per ..... msec
- Thread ..... esegue .... per ..... msec
- Thread ..... esegue .... per ..... msec
- Thread ....
- Thread ....
- Thread ....

ESERCIZIO SincrThread-3

In un sistema con thread *realizzati a livello utente*, dove l'unità schedulabile dal kernel è il processo, sono presenti i processi P1 con thread T11, T12, T13, e P2 con thread T21 e T22. Ogni processo gestisce i suoi thread con politica Round Robin.
Immediatamente prima del tempo *t* è in esecuzione il processo P1 e il processo P2 è pronto. Nel processo P1 è in esecuzione il thread T11 e i rimanenti thread dei due processi sono pronti, con il seguente ordinamento nelle rispettive code:
processo P1: T12-> T13 (ultimo).
Processo P2: T21->T22 (ultimo).

Quale thread va in esecuzione se al tempo t si verificano (in alternativa) i seguenti eventi:

	Evento che si verifica al tempo t	Thread in esecuzione dopo il tempo t
(a)	T11 esegue una wait su un semaforo con valore 0	
(b)	T11 esegue una signal su un semaforo con valore 2	
(c)	T11 esaurisce il proprio quanto di tempo	
(d)	Il sistema operativo scarica il processo P1 in memoria secondaria.	

ESERCIZIO SincrThread-4

In un sistema con thread realizzati a livello utente, sono presenti i processi P1 con thread T11, T12, T13, il processo P2 con thread T21 e T22 e il processo P3 con il solo thread T31. Per ogni processo, lo scheduler dei thread applica la politica FIFO e interviene quando un thread esegue l'operazione *thread_yield*. La politica di scheduling dei processi non prevede il prerilascio.
Al tempo *T* è in esecuzione il processo P1, il processo P2 è pronto e il processo P3 è sospeso sul semaforo *sem1*. Nel processo P1 è in esecuzione il thread T11 e i rimanenti thread dei tre processi sono pronti, con il seguente ordinamento nelle rispettive code:
Processo P1: T12-> T13 Processo P2: T21->T22 Processo P3: T31
Si chiede qual è il contenuto della coda pronti dei processi e quale thread è in esecuzione dopo che si sono verificate (in alternativa) le due seguenti sequenze di eventi:

a. Il thread in esecuzione esegue *thread_yield,* quindi il thread in esecuzione esegue una *signal* sul semaforo *sem1*; quindi il thread in esecuzione esegue una *wait* sul semaforo *sem2,* che ha valore 0;
b. Il thread in esecuzione esegue una *wait* sul semaforo *sem1*; il thread in esecuzione esegue una *thread_yield*; quindi il thread in esecuzione esegue una *signal* sul semaforo *sem1*.

SOLUZIONE

	Sequenze di eventi	Thread in esecuzione	Coda pronti
a-1	Il thread in esecuzione esegue *thread_yield*		
a-2	il thread in esecuzione esegue *signal*(*sem1)*		
a-3	il thread in esecuzione esegue *wait(sem2)*		
b-1	il thread in esecuzione esegue *wait*(*sem1)*		
b-2	il thread in esecuzione esegue *thread_yield*		
b-3	il thread in esecuzione esegue *signal*(*sem1)*		

ESERCIZIO SincrThread-5

In un sistema dove i processi operano in ambiente locale e dispongono di una libreria per la realizzazione dei thread a livello utente, sono presenti il processo P1 con priorità 10, il processo P2 con priorità 5 e il processo P3 con priorità 10. Lo scheduling dei processi avviene con una politica a priorità (va in esecuzione il processo pronto con il massimo valore di priorità) e prevede il prerilascio. Per ogni processo, i thread alternano tra lo stato di esecuzione e quello di pronto. Il thread che passa dallo stato di esecuzione a quello di pronto viene inserito nell'ultima posizione della coda dei thread pronti. Lo scheduling dei thread avviene con politica FIFO, senza prerilascio.
Al tempo *T* è in esecuzione il processo P1, il processo P2 è pronto e il processo P3 è sospeso in seguito all'esecuzione della primitiva *receive(P1, &mess)*. Inoltre sono stati creati seguenti thread:
Processo P1: thread T11, in stato di esecuzione;
Processo P2 : thread T21, T22, tutti in stato di pronto con il seguente ordinamento della coda: → T21 → T22 ;
Processo P3 : thread T31, T32, T33, tutti in stato di pronto con il seguente ordinamento della coda: → T31 → T32 → T33.
Si chiede quale è lo stato dei tre processi e quale thread è in esecuzione **dopo** ciascuno degli eventi della sequenza riportata in tabella.

SOLUZIONE

SEQUENZA DI EVENTI		DOPO L'EVENTO			
		Stato di P1	Stato di P2	Stato di P3	Thread in esecuzione
a)	il thread in esecuzione esegue la primitiva asincrona *send(P3, &messaggio)*				
b)	Il thread in esecuzione esegue l'operazione *thread_yield*				
b)	il thread in esecuzione esegue la primitiva bloccante *receive(P1, &risposta)*				
d)	il thread in esecuzione commette un errore di indirizzamento non recuperabile				
e)	il thread in esecuzione esegue la primitiva asincrona *send(&mess, P1)*				

ESERCIZI DI SINCRONIZZAZIONE TRA THREAD A LIVELLO KERNEL

ESERCIZIO SincrThread-6

In un sistema con thread *realizzati a livello kernel,* dove l'unità schedulabile è il thread, sono presenti i processi P1 con thread T11, T12, T13, e P2 con thread T21 e T22. Il processore viene gestito con politica Round Robin. Immediatamente prima del tempo *t* è in esecuzione il thread T11 e i rimanenti thread sono pronti, con il seguente ordinamento nella coda pronti:
(primo) T12->T21->T13->T22 (ultimo).
Quale thread è in esecuzione se al tempo t si verificano (in alternativa) i seguenti eventi:

	Evento che si verifica al tempo t	Thread in esecuzione dopo il tempo t
(a)	T11 esegue una wait su un semaforo con valore 0	
(b)	T11 esegue una signal su un semaforo con valore 2	
(c)	T11 esaurisce il proprio quanto di tempo	
(d)	Il sistema operativo scarica il processo P1 in memoria secondaria.	

ESERCIZIO SincrThread-7

In un sistema operativo che realizza i thread a livello kernel, il thread T_{1i} del processo P_1 e il thread T_{2k} del processo P_2 si scambiano messaggi attraverso una pila di capacità illimitata. Si consideri la seguente soluzione realizzata con il semaforo Mutex, inizializzato ad 1.

Thread T_{1i}

```
While (True) {
      ……….
      messaggio = produce_messaggio();
      wait(&Mutex);
      puntatore ++;
      Pila[puntatore]= messaggio;
      signal(&Mutex);
                  {
```

Thread T_{2k}

```
While (True) {
      wait(&Mutex);
      mess= Pila[puntatore];
      puntatore --;
      signal(&Mutex);
      consuma_messaggio(mess)
      }
```

La soluzione è corretta? Motivare la risposta.

SOLUZIONE

La soluzione è ……..

Perché ……

ESERCIZIO SincrThread-8

In un sistema operativo che realizza i thread a livello kernel, i thread T1 e T2 del processo P si scambiano messaggi attraverso una pila a tre posizioni (la pila è indicizzata dalla variabile `puntatore` inizializzata a `0`). Si consideri la seguente soluzione realizzata con il semaforo `Mutex`, inizializzato ad `1`, il semaforo `ElementoLibero` inizializzato a `3` e il semaforo `MessaggioPresente` inizializzato a `0`.

Thread T1:	Thread T2:

`While (True) {`
`messaggio= produce_messaggio()`
`wait(&Mutex);`
`wait(&ElementoLibero);`
`Pila[puntatore]= messaggio;`
`puntatore ++;`
`signal(&MessaggioPresente);`
`signal(&Mutex);`
`}`
	`While (True) {`
	`wait(&Mutex);`
	`wait(&MessaggioPresente);`
	`puntatore --;`
	`mess= Pila[puntatore];`
	`signal(&ElementoLibero);`
	`signal(&Mutex);`
	`consuma_messaggio(mess);`
	`}`

La soluzione è corretta? Motivare la risposta.

SOLUZIONE

La soluzione è [corretta/errata] ……..

Motivo:

……

……

……

ESERCIZIO SincrThread-9

In un sistema operativo nel quale le unità schedulabili sono i threads (realizzati a livello kernel),il thread T_{11} del processo P_1 e il thread T_{12} del processo P_1 cooperano scambiandosi messaggi attraverso i buffer Buffer1 e Buffer 2, ciascuno di una sola cella. T_{11} agisce da produttore nei confronti di Buffer1 e da consumatore nei confronti di Buffer2, mentre T_{12} agisce da produttore nei confronti di Buffer2 e da consumatore nei confronti di Buffer1. I due buffer sono inizialmente vuoti.
Si consideri il seguente codice nella quale a Buffer1 sono associati i semafori Buffer1Vuoto (inizializzato ad 1) e Buffer1Pieno (inizializzato a 0), e a Buffer2 sono associati i semafori Buffer2Vuoto (inizializzato ad 1) e Buffer2Pieno (inizializzato a 0). Le operazioni *InsertBuffer1*, *InsertBuffer2*, *RemoveBuffer1* e *RemoveBuffer2*, sono indivisibili e pertanto eseguibili correttamente senza instaurare la mutua esclusione.

```
Thread T11
while (true) {
      NuovoMessaggio= produce_item();
      wait(&Buffer1Vuoto);
      InsertBuffer1(NuovoMessaggio);
      wait (Buffer2Pieno);
      Ricevuto= RemoveBuffer2()
      signal(&Buffer2Vuoto)
      signal(Buffer1Pieno)
      consume(Ricevuto)
      }
```

```
Thread T12
      while (true) {
      NuovoMessaggio= produce_item();
      wait(&Buffer2Vuoto);
      InsertBuffer2(NuovoMessaggio);
      wait (Buffer1Pieno);
      Ricevuto= RemoveBuffer1()
      signal(&Buffer1Vuoto)
      signal(Buffer2Pieno)
      consume(Ricevuto)
      }
```

La soluzione è corretta? In caso negativo fornire una soluzione corretta

SOLUZIONE

a) Soluzione corretta? SI / NO
Motivazione della risposta:

b) Eventualmente: soluzione corretta:

ESERCIZIO SincrThread-10

In un parcheggio per auto della capacità di 10 posti si accede da 2 ingressi, I1 e I2, e si esce dall'unica uscita U. Gli ingressi e le uscite sono controllati da sbarre. Le auto (indicate come A1, A2, ... An) sono thread di uno stesso processo, realizzati a livello kernel.
Per entrare nel parcheggio dall'ingresso I1 (oppure da I2), i thread si accodano a quelli già in attesa di entrare e quando ragiungono il primo posto della coda ottengono l'accesso (cioè provocano l'apertura della sbarra) se nel parcheggio esiste almeno un posto libero. Per uscire, i thread rilasciano il posto che occupavano e si sincronizzano con quelli in attesa di entrare.
Per la gestione del parcheggio si utilizzano i semafori:

- *PostiDisponibili*, inizializzato al valore 10, *(il valore esprime il numero di posti disponibili nel parcheggio)*
- *CodaI1* e *CodaI2,* inizializzati al valore 1. *(CodaI1 contiene i thread auto in attesa di raggiungere la sbarra, analogamente CodaI2)*)
- *MutexPosto,* inizializzato a 1 *(per la selezione, l'occupazione e il rilascio di un posto nel parcheggio)*

Scrivere le procedure eseguite dai thread per entrare dall'ingresso I1 e I2 e quella per uscire dal parcheggio.

SOLUZIONE

Per entrare dall'ingresso 1

```
void function Ingresso1 ()
……;
……;
<si alza la sbarra che lascia passare
un'auto>
<si chiude la sbarra>
……;
……;
<seleziona e occupa un posto libero>
……
```

Per entrare dall'ingresso 2

```
void function Ingresso2 ()
……;
……;
<si alza la sbarra che lascia passare
un'auto>
<si chiude la sbarra>
……;
……;
<seleziona e occupa un posto libero>
……
```

Per uscire dal parcheggio:

```
void function uscita ()
……;
<libera il posto che occupava >
……;
<raggiunge l'uscita e provoca l'apertura della sbarra>
<esce dal parcheggio>
……;
```