Sistemi in Tempo Reale
Compito del 3 luglio 2009

Esercizio 1

Un sistema consiste di un insieme di `N_TASK` task, ognuno dei quali ha il seguente codice:

```
void *taskbody(void *arg)
{
    int id = (int)arg;
    ...
    sync.wait_first_job(id);
    while (1) {
        // task body
        sync.wait_next_job(id);
    }
}
```

L'oggetto `sync` appartiene alla seguente classe:

```
class Synch {
    ...
public:
    Synch(int n);
    void wait_first_job(int task_id);
    void wait_next_job(int task_id);
    void register_task(int task_id,
                       int period,
                       int offset);
};

Synch synch(N_TASK);
```

Il programma funziona nella maniera seguente:

- Prima della creazione dei thread, per ogni thread viene invocata la funzione `register_task()` passando il task id (cioè un numero progressivo che va da 0 a `N_TASK - 1`), il periodo del task e il suo offset iniziale (espressi in microsecondi); tali dati vengono registrati nelle strutture interne della classe;
- Quindi, i task vengono creati, passando il numero progressivo come parametro (arg);
- Quando il task parte, dopo l'inizializzazione, invoca la funzione `wait_first_job()` che

1. Blocca il task fin quando tutti gli altri `N_TASK` sono arrivati al punto in questione, utilizzando il classico meccanismo della *barriera* visto a lezione;
2. A partire dall'istante in cui tutti i task sono arrivati, ogni task aspetta fino all'offset specificato.

- Dopo aver eseguito il suo codice, il task invoca la funzione `wait_next_job()` che aspetta fino al prossimo periodo del task (utilizzando la `clock_nanosleep()` come visto a lezione).

Scrivere le strutture dati necessarie dentro `Synch` e il codice delle funzioni della classe, per realizzare il funzionamento descritto qui sopra.

Esercizio 2

Calcolare se il seguente insieme di task è schedulabile con EDF.

Task	C_i	T_i	D_i
τ_1	1	10	6
τ_2	3	12	12
τ_3	3	15	12
τ_4	6	20	18

Supponendo di poter modificare la deadline del task τ_3 per diminuire il jitter, calcolare di quanto è possibile ridurre tale deadline.

Soluzione esercizio 1

Per risolvere l'esercizio abbiamo bisogno di:

- Le variabili tipiche per costruire una barriera, e cioè una variabile per memorizzare il numero di task (`ntasks`), un contatore (`arrived`) e una variabile condition (`b`).
- Una variabile che ci memorizza il momento esatto in cui tutti i task sono arrivati. Questo corrisponderà all'istante *zero* della nostra schedulazione (`first_offset`).
- Un array di variabili che ci memorizzi i periodi, e un array che ci memorizzi gli offset (`per` e `off`).
- Un array che ci memorizzi l'istante di risveglio per ogni task (`next`).
- Infine, un mutex (`m`).

La classe ha quindi i seguenti membri:

```
class Synch {
    int ntasks;
    int arrived;

    struct timespec  first_offset;
    int  *per;
    int  *off;
    struct timespec *next;

    pthread_mutex_t  m;
    pthread_cond_t   b;
public:
    Synch(int n);
    void wait_first_job(int task_id);
    void wait_next_job(int task_id);
    void register_task(int task_id,
                       int period,
                       int offset);
};
```

Il costruttore:

```
Synch::Synch(int n)
{
    ntasks = n;
    pthread_mutex_init(&m, 0);
    per = new int[n];
    off = new int[n];
    next = new (struct timespec)[n];

    pthread_cond_init(&b, 0);
    arrived = 0;
}
```

La funzione `register_task()` è semplicissima:

```
void Synch::register_task(int task_id,
                          int period,
                          int offset)
{
    per[task_id] = period;
    off[task_id] = offset;
}
```

La funzione `wait_first_job()`:

```
void Synch::wait_first_job(int tid)
{
    struct timespec my_offset;
    pthread_mutex_lock(&m);
    arrived++;
    if (arrived < ntasks)
        pthread_cond_wait(&b, &m);
    else {
        pthread_cond_broadcast(&b);
        clock_gettime(CLOCK_REALTIME, &first_offset);
    }
    pthread_mutex_unlock(&m);
    next[tid] = first_offset;
    timespec_add_us(next[tid], off[tid]);
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, next[tid], NULL);
}
```

Notare che la prima parte della funzione implementa una barriera. L'ultimo task arrivato, oltre a svegliare tutti gli altri (con una `pthread_cond_broadcast()`, prende il tempo e lo memorizza in `first_offset`.

La seconda parte della funzione, invece, calcola l'istante di risveglio e si blocca fino a quell'istante. Notare che quest'ultima parte non ha bisogno di essere eseguita in mutua esclusione, perché ogni thread agisce sulle proprie variabili `next[tid]` e `off[tid]`.

Infine, la funzione `wait_next_period()` è semplicissima:

```
void Synch::wait_next_period(int tid)
{
    timespec_add_us(next[tid], per[tid]);
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, next[tid], NULL);
}
```

Come prima, anche qui non c'è bisogno del mutex.

Soluzione esercizio 2

Per risolvere l'esercizio, dobbiamo calcolare la dbf() dell'insieme dei task. Per prima cosa, calcoliamo il primo istante di idle.

$$W^{(0)} = \sum_{i=1}^{4} C_i = 13$$

$$
\begin{aligned}
W^{(1)} &= \sum_{i=1}^{4} \left\lceil \frac{W^{(0)}}{T_i} \right\rceil C_i = 2+6+3+6 = 17 \\
W^{(2)} &= \sum_{i=1}^{4} \left\lceil \frac{W^{(1)}}{T_i} \right\rceil C_i = 2+6+6+6 = 20 \\
W^{(3)} &= \sum_{i=1}^{4} \left\lceil \frac{W^{(2)}}{T_i} \right\rceil C_i = 20
\end{aligned}
$$

Quindi, $W^* = 20$. Le deadline da considerare sono tutte quelle fino a W^*, quindi:

$$\mathsf{dline} = \{6, 12, 16, 18\}$$

La formula è:

$$\forall L \in \mathsf{dline}, \quad \mathsf{dbf}(L) \leq L$$

Questo si traduce nelle seguenti disequazioni:

$$
\begin{aligned}
C_1 &\leq 6 \\
C_1 + C_2 + C_3 &\leq 12 \\
2C_1 + C_2 + C_3 &\leq 16 \\
2C_1 + C_2 + C_3 + C_4 &\leq 18
\end{aligned}
$$

Per il punto nominale, l'insieme risulta schedulabile.

Per ridurre la deadline del task τ_3, proviamo ad andare per tentativi. Come primo tentativo, dimezziamo l'attuale deadline relativa a $D_3 = 6$. Il punto 12, corrispondente alla prima deadline assoluta del task τ_3 viene quindi spostato a 6. La seconda deadline assoluta va a 21, oltre W^*, quindi non si considera. Le equazioni diventano:

$$
\begin{aligned}
C_1 + C_3 &\leq 6 \\
C_1 + C_2 + C_3 &\leq 12 \\
2C_1 + C_2 + C_3 &\leq 16 \\
2C_1 + C_2 + C_3 + C_4 &\leq 18
\end{aligned}
$$

Ancora una volta, il sistema risulta schedulabile. Proviamo adesso a ridurre la deadline al minimo possibile $D_3 = C_3 = 3$. Il punto 6 si sdoppia e si crea un nuovo punto a 3, corrispondente alla deadline del primo job di τ_3. Inoltre, il secondo job avrà deadline a 18. Le equazioni si modificano in:

$$
\begin{aligned}
C_3 &\leq 3 \\
C_1 + C_3 &\leq 6 \\
C_1 + C_2 + C_3 &\leq 12 \\
2C_1 + C_2 + C_3 &\leq 16 \\
2C_1 + C_2 + 2C_3 + C_4 &\leq 18
\end{aligned}
$$

Ancora una volta l'insieme risulta schedulabile. Siamo quindi riusciti a ridurre a zero il jitter di τ_3.