Informatica e Sistemi in Tempo Reale
Compito del 16 febbraio 2010

Esercizio 1

Un sistema real-time consiste di 3 task periodici τ_1, τ_2, τ_3. In ogni istanza, ogni task esegue una parte obbligatoria (che va eseguita in ogni caso) e una parte opzionale. I task sono schedulati da Fixed Priority, con τ_1 che ha la priorità più alta, e τ_3 quella più bassa.

Normalmente, ogni task esegue sia la sua parte obbligatoria che la sua parte opzionale. Inoltre, ogni task monitora la propria deadline. Nel caso il task τ_i rilevi una deadline miss, esso sospende la propria parte opzionale per le prossime N_i istanze (con N_i configurabile dall'utente); se l'ultima istanza ha ancora una deadline miss, allora tutto il sistema abortisce con errore, altrimenti il task riprende ad eseguire la propria parte opzionale.

1. Scrivere il codice dell'i-esimo task in maniera generica.
2. Discutere sotto quali condizioni un tale metodo non risolva affatto il problema del sovraccarico. Provare a fare un semplice esempio numerico (assegnando periodi e tempi di calcolo) in cui dopo N_2 istanze il task τ_2 ha ancora un sovraccarico e spiegare il perché questo succede.

Esercizio 2

Considerare il seguente sistema schedulato con Fixed Priority e il protocollo Stack Resource Policy:

	C	T	S_1	S_2	S_3	B
τ_1	3	10	2	0	0	
τ_2	2	12	0	1	1	
τ_3	4	18	3	2	0	
τ_4	4	20	1	0	2	

1. Calcolare il tempo di bloccaggio dei vari task
2. Verificare la schedulabilità
3. Calcolare il jitter del task τ_3
4. Verificare se è possibile ridurre il jitter del task τ_3 e come.

Soluzione esercizio 1

Riportiamo per prima cosa il file task.h che contiene la dichiarazione della struttura dati per creare i task:

```
#ifndef __TASK_H__
#define __TASK_H__

#include <pthread.h>

typedef struct taskdata {
    int index;
    int period;   //in musec
    int n_i;      //numero di istanze massimo dmiss consecutive
    int priority; // priorita' del task
} task_data_t;

void *taskbody(void *);

#endif
```

Ecco il codice dei task:

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include "task.h"
#include "time_utils.h"

enum mode_t {NORMAL, EMERGENCY};

int check_deadline(struct timespec *d)
{
    struct timespec now;
    clock_gettime(CLOCK_REALTIME, &now);
    int flag = timespec_cmp(d, &now);
    if (flag < 0) return 0;
    else return 1;
}

void set_priority(int prio)
{
    struct sched_param param;
    param.sched_priority = prio;
    sched_setscheduler(pthread_self(), SCHED_FIFO, &param);
}

void *taskbody(void *arg)
{
    task_data_t data = *((task_data_t *)arg);
    int nmiss;
    int mode = NORMAL;
    int check;
    struct timespec next;

    set_priority(data.priority);
    clock_gettime(CLOCK_REALTIME, &next);
    while (1) {
        printf("Task␣%d:␣Parte␣obbligatoria\n", data.index);

        if (mode == NORMAL) {
            printf("Task␣%d:␣Parte␣opzionale\n", data.index);
```

```
        }

        timespec_add_us(&next, data.period);
        check = check_deadline(&next);

        if (!check && mode == NORMAL) {
            nmiss = 1;
            mode = EMERGENCY;
        }
        else if (!check && (mode == EMERGENCY)) {
            nmiss++;
            if (nmiss > data.n_i) {
                printf("Task␣%d:␣troppe␣deadline␣miss␣consecutive,␣ABORT\n",
                       data.index);
                exit(-1);
            }
        }
        else if (check && (mode == EMERGENCY)) {
            nmiss = 0;
            mode = NORMAL;
        }

        printf("Task␣%d:␣sleeping\n", data.index);
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &next, 0);
    }
    return 0;
}
```

Infine, ecco il codice del main (non richiesto dall'esercizio).

```
#include "task.h"
#include <stdio.h>
#include <unistd.h>

int main()
{
    task_data_t td[3] = {
        {1, 100000, 2,  10},
        {2, 400000, 2,  9},
        {3, 1000000, 4, 8}
    };

    pthread_t id[3];
    int i;

    printf("Main:␣starting\n");

    for (i = 0; i<3; i++)
        pthread_create(&id[i], 0, taskbody, &td[i]);

    printf("Main:␣creati␣tutti␣i␣task␣\n");

    pause();
    return 0;
}
```

Ricordo che in sede di esame non è necessario produrre un programma compilabile o eseguibile. Ad esempio, è possibile evitare di specificare i file include da utilizzare, oppure addirittura tralasciare pezzi di codice non essenziali allo svolgimento dell'esercizio. E' però necessario che il codice utilizzato per la sincronizzazione dei task sia corretto.

Soluzione esercizio 2

Riportiamo qui la tabella dei task, con il loro tempo di bloccaggio. Inoltre, metto nelle ultime due colonne il calcolo dell'utilizzazione, compreso il tempo di bloccaggio, e il tempo di risposta.

task	C	T	B	somma U + B	R
1	3	10	3	0.6	6
2	2	12	3	0.716	8
3	4	18	2	0.8	16
4	4	20	0	0.89	18

Purtroppo, il criterio dell'utilization bound (che per 4 task è pari a 0.7568) fallisce. Quindi, possiamo calcolare i tempi di risposta. Per i primi due task è banale. Per il task τ_3:

$$\begin{aligned} R_3 &= 11 \\ R_3 &= 6+2+4+2=14 \\ R_3 &= 6+4+4+2=16 \\ R_3 &= 6+4+4+2=16 \end{aligned}$$

E per il task τ_4:

$$\begin{aligned} R_4 &= 13 \\ R_4 &= 6+4+4+4=18 \\ R_4 &= 6+4+4+4=18 \end{aligned}$$

Infine, un upper bound per il jitter del task τ_3 è:

$$J_3 \leq R_3 - C_3 = 16 - 4 = 12 \tag{1}$$

Vediamo ora come possiamo diminuire tale jitter. In Fixed Priority, l'unico modo è di innalzare la priorità di τ_3. Proviamo a metterla al massimo. Innalzando la priorità però bisogna anche ricalcolare il tempo di bloccaggio. Ecco la tabella risultante:

task	C	T	S_1	S_2	S_3	B	R
3	4	18	3	2	0	2	6
1	3	10	2	0	0	1	8
2	2	12	0	1	1	2	14
4	4	20	1	0	2	0	–

Come si vede, il tempo di risposta del task τ_2 è ora superiore alla sua deadline, quindi il task missa la sua deadline. Quindi, non è possibile ridurre il tempo di risposta di τ_3 senza causare una deadline miss a τ_2.