

Sistemi in Tempo Reale

Compito del 30 giugno 2010

Esercizio 1

In un sistema real-time, un task produttore rileva un dato con periodo T , e lo inserisce in una struttura dati `FilterData` con la funzione `void insert(double d)`. Contestualmente, viene svegliato un task consumatore che invoca la funzione `double get()` che restituisce il seguente calcolo:

$$f(k) = a(0)x(k) + a(1)x(k-1) + \dots + a(n)x(k-n)$$

dove il vettore $a(\cdot)$ e l'intero n sono parametri della struttura dati, e $x(k)$ rappresenta il k -esimo campione.

Scrivere il codice della struttura dati e le funzioni `insert()` e `get()`, in modo da garantire consistenza dei dati. Inoltre, descrivere (e possibilmente implementare) un meccanismo per rilevare l'eventualità che la `get()` non venga eseguita tra due `insert()` (deadline miss).

Esercizio 2

Consideriamo il seguente sistema di task schedulato con EDF + SRP.

	C_i	T_i	S_1	S_2	S_3
τ_1	2	10	0	1	0
τ_2	3	12	1	0	0
τ_3	2	16	1	0	2
τ_4	8	24	0	1	2

1. Supponendo che le deadline relative siano uguali al periodo, calcolare il tempo di bloccaggio dei task, e verificare la schedulabilità del sistema.
2. Trascurando il tempo di bloccaggio e l'uso delle risorse, assegnare le deadline relative ai task τ_2 e τ_3 in modo che il loro jitter sia minimizzato, mantenendo la schedulabilità del sistema.
3. Discutere (senza calcolare) una possibile metodologia per fare la stessa cosa del punto 2, tenendo anche conto dei tempi di bloccaggio.

Soluzione esercizio 1

Riporto di seguito il codice di una possibile implementazione della struttura dati e delle funzioni.

```
#define NUM_SAMPLES 4
struct FilterData {
    double sample[NUM_SAMPLES];
    double weight[NUM_SAMPLES];
    int head;
    int last_head;
    int dmiss;    // count number of deadline misses
    pthread_mutex_t m;
};

void fd_init(struct FilterData *s, double a[])
{
    int i;
    for (i=0; i<NUM_SAMPLES; i++) {
        s->weight[i] = a[i];
        s->sample[i] = 0;
    }
    s->head = 0;
    s->last_head = 0;
    s->dmiss = 0;
    pthread_mutex_init(&s->m, 0);
}

void fd_insert(struct FilterData *s, double d)
{
    pthread_mutex_lock(&s->m);
    if (s->head != s->last_head) s->dmiss++;

    s->sample[s->head] = d;
    s->head = (s->head + 1) % NUM_SAMPLES;

    pthread_mutex_unlock(&s->m);
}

double fd_get(struct FilterData *s)
{
    int i;
    double sum = 0;
    int h;
    pthread_mutex_lock(&s->m);
    h = s->head;
    s->last_head = s->head;
    for (i=0; i<NUM_SAMPLES; i++) {
        h--;
        if (h<0) h+=NUM_SAMPLES;
        sum += s->weight[i] * s->sample[h];
    }
    pthread_mutex_unlock(&s->m);
    return sum;
}
```

La rilevazione della deadline miss viene fatta attraverso la variabile `dmiss` nella struttura dati. Se il consumatore non ha in tempo a leggere il dato, `head` sarà diverso da `last_head` e quindi la variabile `dmiss` viene incrementata.

Il codice dei task:

```
struct FilterData fd;
void *producer(void *arg)
{
    int i;
    double c = 0;
    long period = *((long *)arg);
    struct timespec next;

    clock_gettime(CLOCK_REALTIME, &next);
    while (1) {
        for (i=0; i<1000000; i++);
        fd_insert(&fd, c);
        printf("Inserting_%lg\n", c);
        c++;
        task_waitperiod(&next, period);
    }
}

void *consumer(void *arg)
{
    int i;
    double c = 0;
    long period = *((long *)arg);
    struct timespec next;

    clock_gettime(CLOCK_REALTIME, &next);
    while (1) {
        for (i=0; i<1000000; i++);
        c = fd_get(&fd);
        printf("Got_%lg\n", c);
        task_waitperiod(&next, period);
    }
}
```

La funzione `task_waitperiod()` aspetta fino al prossimo arrivo periodico. Ecco un esempio di implementazione:

```
int task_waitperiod(struct timespec *next, long period)
{
    struct timespec now;
    clock_gettime(CLOCK_REALTIME, &now);
    timespec_addus(next, period);
    if (timespec_cmp(&now, next) > 0) return 1;
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, next, NULL);
    return 0;
}
```

E infine il codice di inizializzazione nel main:

```
int main()
{
    pthread_t tp, tc;
    double a[] = {.25, .25, .25, .25};
    fd_init(&fd, a);
    long periods[2] = {100000, 100000};
```

```

char c;

pthread_create(&tp, 0, producer, &periods[0]);
pthread_create(&tc, 0, consumer, &periods[1]);

while (1) {
    c = getchar();
    if (c == 'q') break;
}
printf("MAIN:_exiting\n");
return 0;
}

```

Soluzione esercizio 2

I tempi di bloccaggio sono rispettivamente:

$$B_1 = 1 \quad B_2 = 1 \quad B_3 = 2 \quad B_4 = 0$$

Per verificare la schedulabilità, applichiamo prima il test basato sull'utilizzazione:

$$\begin{aligned}
 U_1 + B_1 &= 0.2 + 0.1 < 1 \\
 U_1 + U_2 + B_2 &= 0.2 + 0.25 + 0.1 < 1 \\
 U_1 + U_2 + U_3 + B_3 &= 0.2 + 0.25 + 0.125 + 0.1 < 1 \\
 U_1 + U_2 + U_3 + U_4 &= 0.2 + 0.25 + 0.125 + 0.333 < 1
 \end{aligned}$$

Il sistema risulta dunque schedulabile.

Per minimizzare il jitter dei task τ_2 e τ_3 , mantenendo la schedulabilità, proviamo a mettere la deadline relativa pari al tempo di calcolo e ad applicare il test della demand bound function. La tabella diventa: Chiaramente

	C	D	T
τ_1	2	10	10
τ_2	3	3	12
τ_3	2	2	16
τ_4	8	24	24

il sistema non è schedulabile, perché in un intervallo di 3 bisogna eseguire ben 5 unità di calcolo. E' chiaro che la deadline relativa massima deve dunque essere 5. Due possibilità sono quelle di assegnare $D_2 = 3$ e $D_3 = 5$, oppure $D_2 = 5$ e $D_3 = 2$. La scelta dipende dai requisiti del progettista (non specificati nel compito). Proviamo con la seconda possibilità:

	C	D	T
τ_1	2	10	10
τ_2	3	5	12
τ_3	2	2	16
τ_4	8	24	24

Dobbiamo testare fino al primo idle time:

$$W^0 = \sum C_i = 15 \quad \dots W^5 = 24$$

Le deadline da controllare con la relativa dbf

d	2	5	10	17	18	20	24
dbf	2	5	7	10	12	14	22

Il sistema è dunque schedulabile.

Infine, per quanto riguarda i tempi di bloccaggio, si possono riutilizzare quelli già calcolati finché l'ordinamento tra le deadline rimane uguale. In questo caso, basta aggiungere il tempo di bloccaggio al calcolo della `dbf()` per controllare la schedulabilità.