Sistemi in Tempo Reale
Compito del 9 giugno 2010

Esercizio 1

Considerate 3 task periodici τ_1, τ_2 e τ_3 con periodo 10 ms. Il task τ_2 ha un offset di 5 ms rispetto al task τ_1, mentre il task τ_3 ha un offset di 3 ms rispetto a τ_2. La priorità dei tre task sono $1, 2, 3$, rispettivamente.

1. Scrivere il codice dei 3 task e il codice del main che li crea, facendo attenzione a che gli offset siano corretti.
2. Inoltre, supponiamo che il task τ_1 produca dei dati da mandare al task τ_2, e il task τ_2 abbia dei dati da mandare al task τ_3. Discutere i possibili sistemi di comunicazione dei dati tra i task che garantiscano la consistenza dei dati, indipendentemente dal tempo di calcolo e dal tempo di risposta dei task.
3. Infine, considerare i dati presenti in tabella. Discutere come si possa verificare la schedulabilità del sistema. Disegnare la schedulazione fino a un tempo "sufficientemente" grande.

	C_i	T_i	off_i	p_i
τ_1	2	10	0	1
τ_2	1	10	5	2
τ_3	3	10	8	3
τ_4	2	5	0	4

Esercizio 2

Consideriamo il seguente sistema di task con due modi di di funzionamento, schedulato con Fixed Priority.

	C_i	T_i	p_i	Modo
τ_1	2	5	3	1,2
τ_2	2	8	2	1
τ_3	4	18	1	1
τ_4	2	9	1	2
τ_5	3	12	2	2

Assumere che le deadline relative siano pari al periodo.

1. Calcolare la schedulabilità del sistema separatamente nei due modi di funzionamento.
2. Calcolare il massimo tempo necessario per fare un cambio di modo dal modo 1 al modo 2 usando l'Idle Time Protocol.

NOTA BENE: nella tabella non viene elencato il thread manager. Trascurare il suo tempo di calcolo.

Soluzione esercizio 1

Per fare in modo che l'offset sia corretto, bisogna definire un meccanismo di sincronizzazione tra il thread main e i tre thread periodici. Ecco un possible esempio di struttura dati:

```
struct task_param {
        int index;
        int prio;
        long period_us; // period in usec
        long offset;    // initial offset
};
struct sync {
        int num;       // number of tasks to synchronize
        int arr;       // number of arrived tasks
        pthread_mutex_t m; // mutex
        pthread_cond_t c;  // initial blocking condition
        struct timespec start;
};
void sync_init(struct sync *s, int nthreads)
{
        s->num = nthreads;
        s->arr = 0;
        pthread_mutex_init(&s->m, 0);
        pthread_cond_init(&s->c, 0);
}
struct timespec sync_wait(struct sync *s, long off)
{
        struct timespec mystart;
        pthread_mutex_lock(&s->m);
        s->arr++;
        if (s->arr < s->num)
                pthread_cond_wait(&s->c, &s->m);
        else {
                // take the time
                clock_gettime(CLOCK_REALTIME, &s->start);
                // wake up all other threads
                printf("Starting time is %ld\n", s->start.tv_nsec);
                pthread_cond_broadcast(&s->c);
        }
        mystart = s->start;
        pthread_mutex_unlock(&s->m);
        timespec_addus(&mystart, off);
        printf("Sleeping until %ld\n", mystart.tv_nsec);
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &mystart, 0);
        return mystart;
}
```

Nel codice, la struttura **`struct`** `sync` è utilizzata dai thread per sincronizzarsi sulla partenza. Il codice dei thread è il seguente:

```
struct sync sync_point;

void *thread_body(void *arg)
{
        struct task_param *tp = (struct task_param *)arg;
```

```
        struct timespec next; // next arrival time
        int i;

        printf("task␣created,␣now␣synchronizing\n");
        next = sync_wait(&sync_point, tp->offset);
        printf("task␣%d␣started␣at␣%ld\n", tp->index, next.tv_nsec);
        while (1) {
                // execute
                for (i=0; i<100000; i++); // simulate execution
                printf("task␣%d␣executed,␣waking␣up␣at␣%ld\n", tp->index, next.tv_nsec);
                timespec_addus(&next, tp->period_us);
                clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &next, 0);
        }
}
```

Infine ecco il codice del main:

```
int main()
{
        pthread_t tid[3];
        struct task_param param[3];
        pthread_attr_t attr;
        struct sched_param sp;
        int i;
        char c;

        pthread_attr_init(&attr);
        sync_init(&sync_point, 3);

        param[0].index = 1;
        param[0].period_us = 10000;
        param[0].offset = 0;
        param[0].prio = 1;

        param[1].index = 2;
        param[1].period_us = 10000;
        param[1].offset = 5000;
        param[1].prio = 2;

        param[2].index = 3;
        param[2].period_us = 10000;
        param[2].offset = 8000;
        param[2].prio = 3;

        for (i=0; i<3; i++) {
                pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
                sp.sched_priority = param[i].prio;
                pthread_attr_setschedparam(&attr, &sp);
                pthread_create(&tid[i], &attr, thread_body, &param[i]);
        }
        pthread_attr_destroy(&attr);

        while (1) {
                c = getchar();
                if (c == 'q') break;
```

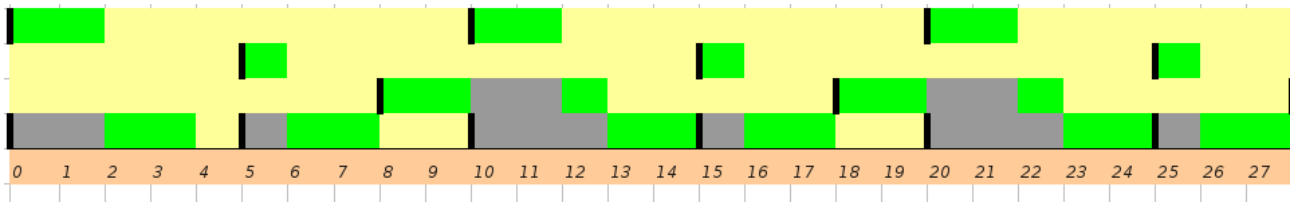
```
        }
        printf("MAIN: closing\n");
        return 0;
}
```

Per quanto riguarda i possibili sistemi di comunicazione:

- Se vogliamo evitare bloccaggi fra task, basta utilizzare una zona di memoria unica in cui un thread (produttore) memorizza il dato, e il thread successivo lo preleva. Se la locazione è inizializzata con un dato significativo, il consumatore legge sempre il dato senza bisogno di bloccarsi. Potrebbe però verificare che un thread possa leggere per due volte lo stesso elemento: ad esempio, se fra due esecuzioni del task τ_2 il task τ_1 non ha modo di eseguire, τ_2 leggerà lo stesso dato due volte. Viceversa, potrebbe accadere che il dato venga sovrascritto senza essere letto: ad esempio, se il task τ_1 esegue due istanze senza che il task τ_2 abbia l'occasione di leggere il dato, quest'ultimo viene sovrascritto. Non è detto che questo sia un errore, ma è una cosa di cui tenere conto.

- Se invece vogliamo essere sicuri di processare tutti i dati, dobbiamo usare una coda circolare. Tutti i thread hanno lo stesso periodo, quindi non si ha rallentamento dei produttori o dei consumatori. Potrebbe però succedere che inizialmente un task resti bloccato in attesa del dato per un certo tempo. Ad esempio, il task τ_3 ha un offset di 8 ms: se il task τ_2 ha un tempo di calcolo superiore ai 3 ms, esso viene interrotto dal task τ_3 (a più alta priorità) prima di produrre il dato, e quindi τ_3 rimane bloccato in attesa di ricevere un dato.

La schedulazione per i parametri in tabella è mostrata nella figura sottostante:



Per la schedulabilità, è necessario fare il disegno fino al tempo 28, poiché bisogna considerare il transitorio (pari all'offset di 8 più l'iperperiodo di 10) e la parte di steady state pari a un iperperiodo.

Soluzione esercizio 2

Consideriamo inizialmente il modo 1. I tempi di risposta dei 3 task sono:

$$R_1 = 2 \quad R_2 = 4 \quad R_3 = 14$$

Quindi il modo 1 è schedulabile. Per il modo 2:

$$R_1 = 2 \quad R_5 = 5 \quad R_4 = 9$$

Quindi, anche il modo 2 è schedulabile (sebbene entrambi τ_4 e τ_5 terminino sulla propria deadline.

Il massimo ritardo per passare dal modo 1 al modo 2 con l'idle time protocol è pari al tempo di risposta del task τ_3. Infatti, potrebbe arrivare una richiesta di cambio modo subito dopo l'attivazione del task τ_3 che nel caso peggiore ci mette 14 istanti di tempo per terminare.