SISTEMAS EMPOTRADOS II

Trabajo de laboratorio

Simón Ortego Parra

---

1. Descripción del problema

Se trata de modelar un programa concurrente con restricciones tiempo real en el cual varias tareas periódicas acceden a unos servidores compartidos con el fin de realizar una sincronización. Los servidores ofrecen una serie de funciones, cada una de ellas es accesible únicamente en exclusión mutua.

Las restricciones de tiempo real implican que las tareas tengan asignadas una serie de prioridades, asignadas por urgencia en el cumplimiento de plazos. En principio, una tarea de mayor prioridad no tiene porque verse afectada por tareas de prioridad inferior al realizar la compartición de recursos con estas otras. En caso contrario, se provocan las denominadas inversiones de prioridad. El análisis de este tipo de situaciones y la garantía del cumplimiento de plazos es el objetivo de este trabajo.

El desarrollo del trabajo se ha realizado sobre un kernel de linux de tiempo real (en concreto el patch PREEMPT_RT) y utilizando la interfaz de programación POSIX.

2. Desarrollo del algoritmo básico

El algoritmo básico del cual partir para más adelante realizar el análisis temporal y de cumplimiento de plazos consta de tres partes claramente diferenciadas. Un programa principal, en el cual se pone en funcionamiento todo, las tareas periódicas y los servidores de acceso concurrente. Se ha realizado en cada parte una programación lo más genérica posible, con el fin de poder reutilizar el código, y que el algoritmo sea lo más conciso y claro posible. A continuación se examinan cada una de las partes.

2.1. Tareas periódicas

Lo primero a realizar para cada tareas es la inicialización. Ya que se implementan sobre la interfaz POSIX, esto además implica establecer los atributos apropiados para los hilos (pthreads) que ejecutan el código de cada una de las tareas. En la función `create_tasks()` se inicializan todas las tareas con los parámetros (`task_params`) y los atributos de los `pthreads` apropiados.

Las tareas periódicas ejecutan una función específica para cada tipo de tarea, pero el comportamiento temporal y la estructura es para todas la misma. A continuación se observa un extracto de código dónde se puede apreciar la función genérica base de todas tareas (`periodic_task`). El parámetro que se le pasa a la función genérica es la dirección de la función que ejecuta cada una de las diferentes tareas.

```
void periodic_task(void *task)
{
    int task_id;
```

```
    void (*task_body) (struct timespec, events_history *);
    events_history *history;
    struct timespec next, comp_time, period;

    task_id = ((task_params *) task) -> task_id;
    task_body = ((task_params *) task) -> task_body;
    history = ((task_params *) task) -> history;

    /* Fills the TIMESPEC struct from the computation time given in milliseconds */
    comp_time.tv_sec = ((task_params *) task) -> computation_time /
        MILLIS_IN_ONE_SEC;
    comp_time.tv_nsec = (((task_params *) task) -> computation_time %
        MILLIS_IN_ONE_SEC) * NANOS_IN_MILLIS;

    /* Fills the TIMESPEC struct from the period value given in milliseconds */
    period.tv_sec = ((task_params *) task) -> period / MILLIS_IN_ONE_SEC;
    period.tv_nsec = (((task_params *) task) -> period % MILLIS_IN_ONE_SEC) *
        NANOS_IN_MILLIS;

    if (clock_gettime (CLOCK_MONOTONIC, &next) != 0) {
        fprintf(stderr, "T%d: periodic_task(): failed to get the current time: ",
            task_id);
        perror(NULL);
        return;
    }

    int i;
    for (i = 0; i < NUM_TASK_ITERATIONS; i++) {
        add_task_event(TASK_ACTIVATION, history);
        task_body(comp_time,history);
        next = tsAdd(next, period);
        clock_nanosleep (CLOCK_MONOTONIC, TIMER_ABSTIME, &next, 0);
        add_task_event(TASK_COMPLETION, history);
    }

    pthread_exit(task);
}
```

Una vez realizada la programación del código genérico, tan sólo es necesario realizar la función que se le pasa como parámetro a esta función genérica comentada con anterioridad, de tal forma que, en el cuerpo de la función genérica cada tarea específica llame a la función que le corresponda. Un ejemplo de las funciones específicas se puede observar a continuación para el caso de la tarea 1.

```
void t1_task_body(struct timespec comp_time, events_history *history)
{
    calc(comp_time); /* doing stuff */
    server1_func_1(1,history);
}
```

Lo único que falta por aclarar es que ejecuta cada tarea periódicamente (en el algoritmo anterior aparece una llamada a la función `calc` que no es más que una simulación de la ejecución de un código arbitrario que mantiene a la tarea ocupada durante el tiempo que se le pase como parámetro. Esta función se detalla en la siguiente sección.

2.1.1. Tiempos de cómputo

El objetivo de este proyecto es puramente educacional. Esto quiere decir que, con el fin de garantizar unos tiempos de cómputo constantes para cada una de las tareas, y así poder realizar el análisis temporal correctamente, se ha programado una función que realiza una espera activa hasta que haya pasado el tiempo que se le pase como parámetro (es importante recalcar lo de espera activa: la tarea no se puede dormir ya que supone que se

está modelando que la tarea está ejecutando una sección de código que dura exactamente el tiempo de cómputo que se le pasa a la función como parámetro).

```
void calc (struct timespec ms)
{
    struct timespec now, end;

    clock_gettime(CLOCK_REALTIME, &now);
    end = tsAdd(now, ms);

    while (tsCompare(now, end) != 1) {
        clock_gettime(CLOCK_REALTIME, &now);
    }
}
```

2.2. Servidores

2.3. Programa principal

3. Análisis de tiempo real

3.1. Recolección de eventos. Temporización

4. Entorno de desarrollo

4.1. Kernel build: Digital Ocean VM3

A. Ficheros de configuración

A.1. Servidor Kerberos maestro: `krb1`

A.1.1. `/etc/rc.conf.local`