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 ponen en funcionamiento todas las partes, 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;

    /* Main loop of the task */
    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 rea-

liza 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

La metodología de programación a la hora de realizar los servidores es exactamente la misma.

Al igual que las tareas, los servidores requieren de una inicialización.

Por otro lado, se ha programado una función genérica (`server_function`), a la que se le pasa como parámetros una serie de datos para distinguir cada una de las funciones de los servidores ya específicas. Dicha función genérica aparece a continuación.

```
void server_function (int server_id, int function_id,
      struct timespec computation_time, pthread_mutex_t mutex, int task_id,
          events_history *history)
{

   /* tries to aquire mutex */
   pthread_mutex_lock(&mutex);

   /* --- Critical Section ----- */
   add_task_event(CS_ENTRY, history);
   calc(computation_time);
   add_task_event(CS_EXIT, history);
   /* ------------------------- */

   /* releases mutex */
   pthread_mutex_unlock(&mutex);
}
```

2.3. Programa principal

En el programa principal se inicializan las tareas, los servidores y se ponen en funcionamiento las tareas, cada una de ellas en un `pthread` diferente. En algún momento, acaba el programa (se ha limitado la ejecución de las tareas a un número de iteraciones), y entonces se realiza la labor de liberación de recursos, etc.

Este programa principal, que representa el punto de entrada del programa concurrente, se muestra (con alguna edición) en el siguiente extracto de código.

```
int main (int argc, char **argv)
{
    /* Threads that execute tasks and its attributes */
    pthread_t threads[NUM_TASKS];
    pthread_attr_t thread_attributes[NUM_TASKS];
```

```
    /* Timespec structure to set the timer to zero at
     * the beginning of the program execution */
    struct timespec start;

    /* Data needed for the execution of each periodic task:
     * id, period, computation time & task code executed */
    task_params params[NUM_TASKS];

    /* Set the parameters for the tasks (including thread attributes) */
    create_tasks(threads, thread_attributes, params);

    /* Create the servers, which the tasks make use of */
    create_servers();

    /* Create one independent thread for each task */
    for (i = 0; i < NUM_TASKS; i++) {
        if (pthread_create(&threads[i], &thread_attributes[i], (void *)periodic_task
            , &params[i]) != 0) {
            fprintf(stderr, "pthread_create(): failed to create thread#%d: ", i);
            perror(NULL);
            return -1;
        }
    }

    /* Dead code (supposedly, if tasks run forever) */
    for (i = 0; i < NUM_TASKS; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            fprintf(stderr, "pthread_join(): Thread %d did not terminate normally: "
                , i);
            perror(NULL);
            return -1;
        }
        print_events(*params[i].history);
        clear_history(params[i].history);
    }

    return 0;
}
```

3. Análisis de tiempo real

3.1. Recolección de eventos. Temporización

Antes de poder realizar ninguna clase de análisis, es necesario tener (si no se realiza de manera teórica) un historial de eventos o diagrama de alguna clase que muestre una ejecución de las tareas, para poder sacar conclusiones.

Se han programado una serie de funciones para recolectar los eventos que se deseen en los instantes que se estime oportuno. Éstas se describen a continuación.

3.1.1. Inicialización

Ya que en primera instancia se desconoce el número de eventos que va a ser capturado, se ha pensado que éstos no pueden ser almacenados en una estructura estática ya que es complicado dimensionarla sin que haya desbordamiento y se pierdan eventos o se reserve una cantidad de memoria innecesaria.

Como se puede apreciar a continuación, se ha definido la estructura dónde almacenar los eventos como una lista encadenada simple.

```
/* events.h */
```

```
...
#define TASK_ACTIVATION 0
#define TASK_COMPLETION 1
#define CS_ENTRY 2
#define CS_EXIT 3

typedef struct event {
    int type;
    struct timespec timestamp;
    struct event *next_event;
} event;

typedef struct events_history {
   int task_id;
   event *first_event, *last_event;
} events_history;
```

En la inicialización de la lista simplemente se reserva una cantidad de memoria dinámica, para la estructura que almacenará los eventos.

```
events_history *create_events_history (int task_id)
{
   events_history *new_history;
   new_history = malloc(sizeof(events_history));
   if (new_history != NULL) {
      new_history->task_id = task_id;
      new_history->first_event = NULL;
      new_history->last_event = NULL;
   }
   return new_history;
}
```

3.1.2. Guardar un evento

En el momento de la ejecución de una tarea que se desee tener registro de un evento, se realizará una llamada a la función `add_event()`, que almacenará dicho evento en la estructura comentada con anterioridad.

```
void add_event (int type, events_history *history)
{
    struct timespec timestamp;
    event *new_event;

    clock_gettime(CLOCK_REALTIME, &timestamp);
    new_event = malloc(sizeof(event));

    if (new_event != NULL) {
        /* if could allocate memory for the new event */
        new_event->type = type;
        new_event->timestamp.tv_sec = timestamp.tv_sec;
        new_event->timestamp.tv_nsec = timestamp.tv_nsec;
        new_event->next_event = NULL;

        if (history->last_event != NULL) {
            /* if history is not empty */
            history->last_event->next_event = new_event;
      } else {
          /* if history is empty */
          history->first_event = new_event;
      }
      history->last_event = new_event;
   } else {
       fprintf(stderr, "add_event(): could not allocate memory for new event: ");
   }
}
```

3.1.3. Imprimir los eventos y liberar la memoria

Únicamente cuando ha terminado la ejecución del programa concurrente (todas las tareas han terminado), se muestra por pantalla la recolección de eventos (`print_event()`) y a continuación, se libera la memoria reservada (`clear_history()`).

3.2. Recolección de eventos. Temporización

4. Sin herencia de prioridad vs. Techo de prioridad inmediato

5. Parte opcional implementada

Además de la recolección de eventos básica, se quería realizar una recolección de eventos más fina, para determinar correctamente los instantes de expulsión de las tareas. La parte de trabajo realizado (sin implementar) se detalla a continuación.

6. Metodología de trabajo. Entorno de desarrollo

Además de la rápida configuración de una máquina virtual Ubuntu para poder trabajar desde casa y poder realizar la compilación cruzada, el formateado de la tarjeta sd, etc. (aspectos que obviamente no entran dentro del ámbito de esta asignatura), se quiere hacer hincapié en una serie de herramientas que han sido imprescindibles para agilizar el desarrollo y evitar dolores de cabeza futuros:

- En primer lugar, es más, esto es lo que primero se hizo cuando se comenzó el desarrollo del trabajo, es necesario la configuración de un `Makefile`.
- En segundo lugar un programa de gestión de versiones (`git`). Para la parte opcional del trabajo no solamente es imprescindible sino que no existe otro método de trabajo eficaz (no es concebible realizar modificaciones en el kernel de Linux, o descargárselo sin perder mucho tiempo, si no se utilizan esta clase de herramientas). Por otro lado, el código tanto de la parte obligatoria, como de la parte opcional se encuentra alojado un repositorio de `github`, lo cual ha permitido cambiar de entorno de trabajo de manera eficaz.
  - Repositorio con la parte obligatoria:
    `https://github.com/saimusdev/SE2Proyecto`
  - Repositorio con lo desarrollado como inicio de parte opcional:
    `https://github.com/saimusdev/bone-rt`
- En la parte opcional de la asignatura ha sido muy valioso el contar con máquinas virtuales en la nube para: primero, que

A. Ficheros de configuración

A.1. Servidor Kerberos maestro: `krb1`

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