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.

El tener restricciones de tiempo real implica que las tareas tengan asignadas prioridades estáticas, definidas en función de la urgencia del cumplimiento de sus plazos de entrega. En principio, una tarea de mayor prioridad no tiene por qué verse afectada por tareas de prioridad inferior al compartir recursos con estas últimas. En caso contrario, se provocan las denominadas inversiones de prioridad. El análisis de este tipo de situaciones y la garantía del cumplimiento de los plazos de las tareas 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 se ha utilizado 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 tarea 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, pero el comportamiento temporal y la estructura es para todas la misma. A continuación se muestra un extracto de código en dónde se puede apreciar la función genérica utilizada como base para todas tareas (`periodic_task()`). El parámetro que se le pasa a la función genérica contiene, entre otras cosas, la dirección de la función específica 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)) {
        fprintf(stderr, "T%d: periodic_task(): failed to get the current time: ",
            task_id);
        perror(NULL);
        return;
    }
    add_event(TASK_BIRTH, 0, history);

    /* Main loop of the task */
    int i;
    add_event(TASK_ACTIVATION, 0, history);
    for (i = 0; i < NUM_TASK_ITERATIONS; i++) {
        task_body(comp_time,history);
        add_event(TASK_FINISHED, 0, history);
        next = tsAdd(next, period);
        clock_nanosleep (CLOCK_MONOTONIC, TIMER_ABSTIME, &next, 0);
        add_event(TASK_ACTIVATION, 0, history);
    }

    add_event(TASK_DEATH, 0, history);
    pthread_exit(task);
}
```

Ahora, 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, para cada tarea, se realice una llamada a la función correspondiente. Un ejemplo de las funciones específicas se puede observar a continuación (para el caso de la tarea 1).

```
void t1_body(struct timespec comp_time, events_history *history)
{
    add_event(TASK_BUSY, 0, history);
    calc(comp_time); /* doing stuff */
    s11(1, history);
}
```

Lo único que falta por aclarar es el qué ejecuta cada tarea periódicamente (en el algoritmo anterior aparece una llamada a la función `calc()`, que no es más que 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

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 fun-

ción que realiza una espera activa hasta que haya transcurrido el tiempo que se le pase como parámetro (es importante recalcar lo de espera activa: la tarea no se puede dormir ya que se supone que se está modelando un sistema en el cual la tarea estaría ejecutando una sección de código real, que dura aproximadamente 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 (`create_servers()`).

También, 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 (unsigned int func_id, struct timespec comp_time,
    pthread_mutex_t *mutex, events_history *history)
{
    add_event(SERVER_ENTRY, func_id, history);

   /* tries to aquire mutex */
    add_event(MUTEX_LOCK, func_id, history);
   pthread_mutex_lock(mutex);

   /* --- Critical Section ----- */
    add_event(MUTEX_AQUIRE, func_id, history);
   calc(comp_time);
   /* ------------------------- */

   /* releases mutex */
    add_event(MUTEX_RELEASE, func_id, history);
   pthread_mutex_unlock(mutex);

   add_event(SERVER_EXIT, func_id, history);
}
```

De nuevo, se ha utilizado la función genérica en cada una de las diferentes funciones que ofrece como interfaz el servidor. Un ejemplo se muestra en el siguiente extracto.

```
void s11 (int task_id, events_history *history)
{
   server_function (S11, s11_comp_time, &s1_mutex, history);
}
```

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();

    /* Initialize clock */
    start.tv_sec = 0;
    start.tv_nsec = 0;
    if (clock_settime(CLOCK_REALTIME, &start)) {
        fprintf(stderr, "main(): failed to reset the timer: ");
        perror(NULL);
        return -1;
    }

    int i;
    /* 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])) {
            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)) {
            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 para así 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 porque resulta complicado dimensionarla sin que haya desbordamientos 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_BIRTH 0
#define TASK_ACTIVATION 1
#define TASK_BUSY 2
#define TASK_FINISHED 3
#define SERVER_ENTRY 4
#define MUTEX_LOCK 5
#define MUTEX_AQUIRE 6
#define MUTEX_RELEASE 7
#define SERVER_EXIT 8
#define TASK_DEATH 9

typedef struct event {
    unsigned char type, server_id;
    struct timespec timestamp;
    struct event *next_event;
} event;

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

En la inicialización de la lista simplemente se reserva una cantidad de memoria 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. Almacenar 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. En esta función, se eleva la prioridad del hilo que la ejecuta a la máxima, para que no haya expulsión. Una vez se tiene registro del evento, se vuelve a bajar la prioridad.

```
void add_event (unsigned char type, unsigned char server_id, events_history *history
    )
{
    int thread_priority;
    pthread_attr_t thread_attr;
    struct sched_param thread_sched;
    struct timespec timestamp;
   event *new_event;

    // Retrieve the thread attributes and with those, the scheduling parameters:
        thread priority
    if (pthread_getattr_np(pthread_self(), &thread_attr)) {
        fprintf(stderr, "add_event(): failed to get the running thread attributes: "
            );
        perror(NULL);
        return;
    } else if (pthread_attr_getschedparam(&thread_attr, &thread_sched)) {
        fprintf(stderr, "add_event(): failed to get the running thread scheduling
            parameters (priority): ");
        perror(NULL);
        return;
    }

    // Save the current thread's priority
    thread_priority = thread_sched.sched_priority;

    // Set the thread's priority to the max level
    if (pthread_setschedprio(pthread_self(), max_sched_priority)) {
        fprintf(stderr, "add_event(): failed to elevate the thread's priority: ");
        perror(NULL);
        return;
    }

      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->server_id = server_id;
      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: ");
   }

    // Restore the original thread priority
    if (pthread_setschedprio(pthread_self(), thread_priority)) {
```

```
        fprintf(stderr, "add_event(): failed to restore the thread's priority: ");
        perror(NULL);
        return;
    }
}
```

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. Inversión de prioridad. Protocolo de Techo de Prioridad Inmediato

Después de realizar la versión sencilla, se ha pasado a utilizar el protocolo de techo de prioridad inmediato para así evitar las inversiones de prioridad provocadas por el acceso a recursos compartidos entre las diferentes tareas.

Esto implica únicamente modificar el comportamiento de los mutex que protegen la sección crítica de las funciones de los servidores que son llamadas por las tareas. A continuación se muestra dicha modificación.

```
void create_servers (void)
{
   pthread_mutexattr_t s1_mutex_attr, s2_mutex_attr;

    /* server 1 - function 1 */
    s11_comp_time.tv_sec = S11_COMP_TIME / MILLIS_IN_ONE_SEC;
    s11_comp_time.tv_nsec = (S11_COMP_TIME % MILLIS_IN_ONE_SEC) * NANOS_IN_MILLIS;

    /* server 1 - function 2 */
    s12_comp_time.tv_sec = S12_COMP_TIME / MILLIS_IN_ONE_SEC;
    s12_comp_time.tv_nsec = (S12_COMP_TIME % MILLIS_IN_ONE_SEC) * NANOS_IN_MILLIS;

    /* Init mutex for ICPP (ceiling = highest priority task, T1) */
    pthread_mutexattr_setprotocol(&s1_mutex_attr, PTHREAD_PRIO_PROTECT);
    pthread_mutexattr_setprioceiling (&s1_mutex_attr, T1_PRIORITY);
    pthread_mutex_init(&s1_mutex, &s1_mutex_attr);

    /* server 2 - function 1 */
    s21_comp_time.tv_sec = S21_COMP_TIME / MILLIS_IN_ONE_SEC;
    s21_comp_time.tv_nsec = (S21_COMP_TIME % MILLIS_IN_ONE_SEC) * NANOS_IN_MILLIS;

    /* server 2 - function 2 */
    s22_comp_time.tv_sec = S22_COMP_TIME / MILLIS_IN_ONE_SEC;
    s22_comp_time.tv_nsec = (S22_COMP_TIME % MILLIS_IN_ONE_SEC) * NANOS_IN_MILLIS;

    /* Init mutex for ICPP (ceiling = highest priority task, T2) */
    pthread_mutexattr_setprotocol(&s2_mutex_attr, PTHREAD_PRIO_PROTECT);
    pthread_mutexattr_setprioceiling (&s2_mutex_attr, T2_PRIORITY);
    pthread_mutex_init(&s2_mutex, &s2_mutex_attr);
}
```

4. Bolas extra

Además de la recolección de eventos básica, se quería realizar una recolección de eventos más fina, para determinar de manera exacta y con total precisión los instantes de expulsión de las tareas. El trabajo realizado (sin acabar) está en un repositorio de Github (`https://github.com/saimusdev/bone-rt`).

Se pretendía modificar la función `__schedule()` de Linux, para que en el instante de expulsión de una tarea, se almacene junto con una estampilla temporal, el identificador de proceso de la tarea expulsada. El almacenamiento de esta clase de eventos se podría realizar basándose en el código ya realizado en este trabajo, pero utilizando un módulo del Kernel (compilado junto con el Kernel) para que al ser cargado active un flag (variable global) que le indique al planificador de Linux que haga una llamada (`write()`, por ejemplo) para almacenar la información pertinente en un buffer.

Se ha conseguido imprimir por pantalla los pids de todos los procesos en el instante de expulsión (inútil), y también se ha conseguido compilar un driver (utilizando la convención de Linux para añadirlo dentro de la jerarquía de directorios del kernel y no fuera: Kconfig, Makefile,...) No se han conseguido utilizar variables globales al desconocer dónde realizar la declaración para ser visible en todo el kernel (y que no falle la compilación) y tampoco se ha conseguido realizar una llamada `ioctl()` o `write()` sencilla al driver, por falta de tiempo.

5. Metodología de trabajo. Entorno de desarrollo

Además de la configuración de una máquina virtual Linux 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 quebraderos de cabeza:

- `make`. Lo que primero se hizo cuando se comenzó el desarrollo del trabajo, es la configuración de un `Makefile`. La utilización de órdenes para realizar la depuración del código resultó gran utilidad.
- Gestión de versiones (`git`). Para la parte extra 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 la opcional:
    `https://github.com/saimusdev/bone-rt`
- `screen`. No solamente para poderse conectar a la beaglebone, sino como entorno de trabajo: para poder cambiar instantáneamente entre el host/target.
- En la parte extra desarrollada ha sido muy valioso el contar con máquinas virtuales en la nube (digitalocean) para: primero, descargarse repositorios de manera casi instantánea, y segundo, reducir el tiempo de construcción del núcleo considerablemente (las máquinas utilizadas poseían 12-20 cpus). Estas máquinas, aunque en circunstancias normales son muy costosas, con la oferta que se obtuvo de Github para estudiantes, resultó completamente gratis.