Master en Computación

Plataformas de Tiempo Real

POSIX Avanzado y Extensiones

Tema 3. Monitorización y control avanzado del tiempo de ejecución

3.1 Relojes y temporizadores de tiempo de ejecución

Ya vistos en "Programación Concurrente"

Relojes de tiempo de ejecución

- **Permiten conocer el tiempo consumido por un thread**

Temporizadores de tiempo de ejecución

- **Permiten detectar un consumo excesivo de tiempo de ejecución**

Relojes de tiempo de ejecución: Resumen

Constante que identifica el reloj del thread actual:

```
const clockid_t CLOCK_THREAD_CPUTIME_ID = ...;
```

Obtener el identificador de reloj de un thread cualquiera:

```
#include <pthread.h>
#include <time.h>
int pthread_getcpuclockid(pthread_t thread_id,
                          clockid_t *clock_id);
```

Leer el tiempo consumido por el thread actual:

```
struct timespec time;
...
clock_gettime(CLOCK_THREAD_CPUTIME_ID, &time);
```


Temporizadores de tiempo de ejec.: Resumen

Se utiliza la interfaz de temporizadores POSIX

- **basados en un reloj de tiempo de ejecución**
- **lanzan una señal cuando expiran**

Crear un temporizador:

```
#include <signal.h>
#include <time.h>
int timer_create (clockid_t clock_id,
                  struct sigevent *evp,
                  timer_t *timerid);
```

Armar (programar) un temporizador:

```
int timer_settime (timer_t timerid, int flags,
                   const struct itimerspec *value,
                   struct itimerspec *ovalue);
```


3.2 Patrón de diseño para threads de TR

Define un marco que permite implementar de forma sencilla:

- **threads periódicos y esporádicos**

Con detección automática de:

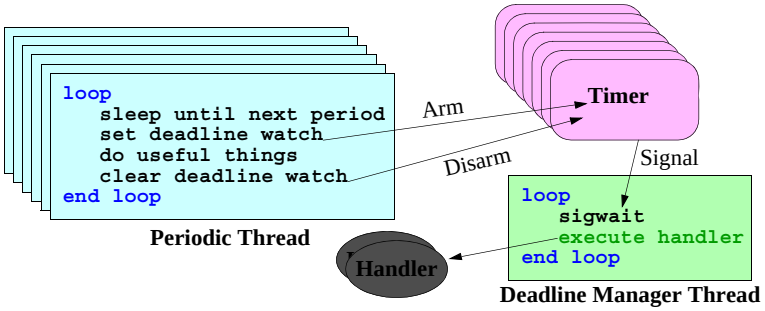
- **sobrepaso de plazo**
- **sobrepaso de tiempo de ejecución de peor caso**

Basado en la utilización de temporizadores y relojes POSIX

Checking Deadlines

Periodic and sporadic hard real-time threads have deadlines

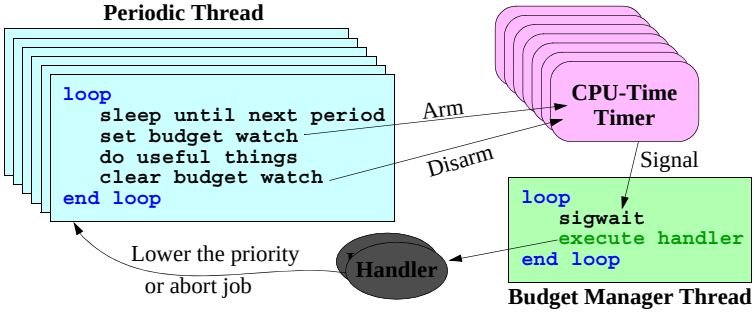
It is useful to be able to perform some error-handling action if the deadline is not met



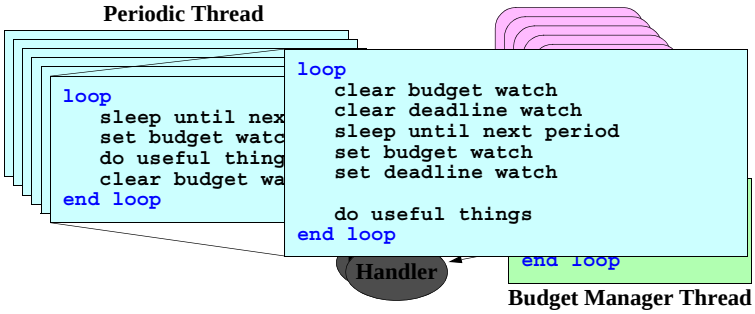
Enforcing CPU-time budgets

Hard real-time threads may overrun their estimated WCETs, perhaps causing somebody else to miss its deadline

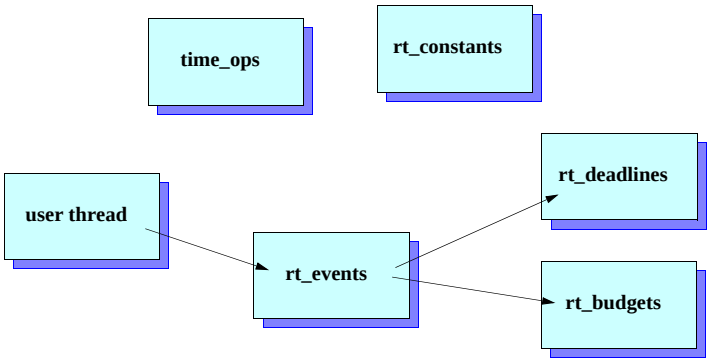
It is necessary to detect and limit budget overruns



Common actions for a periodic thread



Example: packages for periodic threads



Example sources: rt_budgets.h

```
#include <time.h>
#include "rt_constants.h"

#ifndef RT_BUDGET_TIMERS
#define RT_BUDGET_TIMERS

#define RT_BUDGET_SIGNAL SIGRTMIN

// Opaque datatype for the budget timer
typedef struct { ... } rt_budget_timer;
```


rt_budgets.h (cont'd)

```
// Creates the budget manager thread if it is not already created
int rt_init_budget_manager();

// Creates a budget timer for the calling thread, with a
// normal priority level equal to the calling thread's priority
int rt_init_budget_timer(rt_budget_timer *budgettimer,
                         void (*action)(void *arg),
                         void * arg);

// Starts counting the budget
int rt_set_budget_timer(rt_budget_timer *budgettimer,
                        struct timespec budget);

// Stops counting the budget
int rt_clear_budget_timer(rt_budget_timer *budgettimer);

#endif //RT_BUDGET_TIMERS
```

Example sources: rt_deadlines.h

```
#include <time.h>

#ifndef RT_DEADLINES
#define RT_DEADLINES

#define RT_DEADLINE_SIGNAL SIGRTMIN+1

// Opaque datatype for the deadline timer
typedef struct { ... } rt_dln_timer;
```


rt_deadlines.h (cont'd)

```
// Creates the deadline manager thread if it is not already created
int rt_init_deadline_manager();

// Creates a budget timer for the calling thread, with the
// specified action and arg. associated with the expiration
int rt_init_deadline_timer(rt_dln_timer *dlntimer,
                           void (void *arg) action,
                           void * arg);

// Sets the deadline timer to expire at the deadline
int rt_set_deadline_timer(rt_dln_timer *dlntimer,
                          struct timespec budget);

// Stops the deadline timer
int rt_clear_deadline_timer(rt_budget_timer *budgettimer);

#endif //RT_DEADLINES
```


Example sources: rt_events.h

```
#include <time.h>
#include "rt_constants.h"
#include "rt_budgets.h"
#include "rt_deadlines.h"

#ifndef RT_EVENT_MANAGERS
#define RT_EVENT_MANAGERS

// Event manager types
typedef enum {PERIODIC, SPORADIC, ...} event_manager_t;

// Opaque datatype for the event manager
typedef struct {
  rt_budget_timer budgettimer;
  rt_dln_timer dlntimer;
  ... } rt_event_manager;
```

Los campos `budgettimer` y `dlntimer` permiten cambiar el manejador usando las funciones `rt_init_budget_timer` y `rt_init_deadline_timer` respectivamente.

Los manejadores por defecto se limitan a escribir un mensaje por consola.

rt_events.h (cont'd)

```
// Initializes all the resources needed by the event managers
int rt_init_event_managers();

// Creates a periodic event manager
int rt_init_periodic_event_manager(rt_event_manager *evntmanager,
                                   struct timespec period,
                                   struct timespec budget,
                                   struct timespec reldeadline,
                                   int task_num);

// Wait for next period
int rt_schedule_periodic_job(rt_event_manager *evntmanager,
                             int *deadline_was_missed,
                             int *budget_was_overrun);

#endif //RT_EVENT_MANAGERS
```


Example sources: usage

```
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <sched.h>
#include <signal.h>
#include <unistd.h>
#include <misc/error_checks.h>
#include "load.h"
#include "rt_events.h"

static int error_status=-1;

struct periodic_data {
  struct timespec period;
  struct timespec wcet;
  struct timespec deadline;
  int id;
};
```


Example sources: usage (cont'd)

```
// Body of periodic thread
void * periodic (void *arg)
{
  struct periodic_data my_data;
  int errorcode;
  int deadline_was_missed;
  int budget_was_overrun;
  rt_event_manager evntmanager;

  // initialization
  my_data = * (struct periodic_data*)arg;
  if ((errorcode = rt_init_periodic_event_manager
       (&evntmanager,my_data.period, my_data.wcet,
        my_data.deadline, my_data.id)) != OK)
    {
      error_status=errorcode; pthread_exit((void *) &error_status);
    }
```

Example sources: usage (cont'd)

```
  // periodic loop
  while (1) {
    if ((errorcode=rt_schedule_periodic_job
        (&evntmanager,&deadline_was_missed,
         &budget_was_overrun))!=OK)
      {
         error_status=errorcode;
         pthread_exit((void *) &error_status);
      }
    // do useful work
    ....
  }
}
```


Example sources: usage (cont'd)

```
// Main program, that creates a periodic thread
int main ()
{
  pthread_t t1;
  sigset_t set;
  struct periodic_data per_params1;
  struct sched_param sch_param;
  pthread_attr_t attr;
  int errorcode;

  sigemptyset(&set);
  sigaddset(&set,RT_BUDGET_SIGNAL);
  sigaddset(&set,RT_DEADLINE_SIGNAL);
  CHK( pthread_sigmask(SIG_BLOCK, &set, NULL) );
```


Example sources: usage (cont'd)

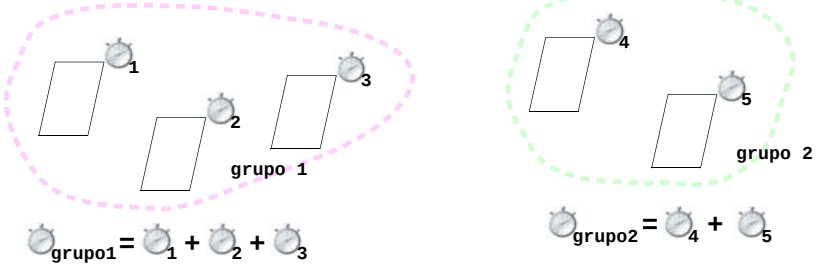
```
  if ((errorcode=rt_init_event_managers())!=OK) {
    printf ("Error code %d in init event managers\n",errorcode);
    exit (1);
  }
  per_params1.period.tv_sec=2;
  per_params1.period.tv_nsec=500000000;
  per_params1.wcet.tv_sec=1;
  per_params1.wcet.tv_nsec=500000000;
  per_params1.deadline=per_params1.period;
  per_params1.id=1;

  // Create the thread attributes object
  CHK( pthread_attr_init(&attr) );
```

Example sources: usage (cont'd)

```
  // Set each of the scheduling attributes
  CHK( pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED) );

  CHK( pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) );

  CHK( pthread_attr_setschedpolicy(&attr, SCHED_FIFO) );

  sch_param.sched_priority = (sched_get_priority_min(SCHED_FIFO)+3);
  CHK( pthread_attr_setschedparam (&attr,&sch_param) );

  CHK( pthread_create (&t1, &attr, periodic, &per_params1) );

  // do other things
}
```


3.3 Relojes de tiempo de ejecución para grupos de threads

Miden el tiempo de ejecución consumido por un grupo de threads



La interfaz `<thread_sets.h>` proporciona funciones para:

- **Gestionar los grupos de threads (*threads sets*)**
- **Obtener el reloj asociado a un grupo**

Utilidad de los relojes de grupo

Estos relojes pueden utilizarse como cualquier otro reloj POSIX:

- **`clock_gettime()`, `clock_settime()`**
- ***Temporizadores***
- **...**

Permiten lograr el aislamiento temporal entre distintas partes de la aplicación

- **evitando que cada parte (componente) consuma más tiempo del que le corresponde**
- **utilizando un algoritmo de reserva de ancho de banda (como el servidor esporádico)**
- **cuando un grupo sobrepasa su presupuesto de ejecución el temporizador expira y se realiza la acción correspondiente**
  - **p.e. bajar la prioridad de los threads del grupo**

Creación y destrucción de *thread sets*

- **Creación:**

```
#include <thread_sets.h>
int marte_threadset_create(
                marte_thread_set_t *set);
```

- **Destrucción**

```
int marte_threadset_destroy(
                marte_thread_set_t set);
```


Añadir y eliminar threads de un *thread set*

- **Deja el grupo vacío:**

```
int marte_threadset_empty(marte_thread_set_t set);
```

- **Añade un thread al grupo:**

```
int marte_threadset_add(marte_thread_set_t set,
                        pthread_t thread_id);
```

  - Retorna `ENOTSUP` si el thread ya es miembro de algún otro grupo

- **Elimina un thread del grupo:**

```
int marte_threadset_del(marte_thread_set_t set,
                        pthread_t thread_id);
```


Recorrer los threads de un *thread set*

- **Obtiene el primer thread del grupo**

```
int marte_threadset_first(marte_thread_set_t set,
                          pthread_t *thread_id);
```

  - Retorna `ESRCH` si el grupo está vacío

- **Obtiene el siguiente thread del grupo**

```
int marte_threadset_next(marte_thread_set_t set,
                         pthread_t *thread_id);
```

  - Retorna `EINVAL` si el thread actual ha sido eliminado del grupo desde la última llamada a `marte_threadset_first()` o `marte_threadset_next()`
  - Retorna `ESRCH` cuando no hay más threads en el grupo

Test de pertenencia a un *thread set*

- **Retorna el grupo al que pertenece un thread:**

```
int marte_threadset_getset(pthread_t thread_id,
                           marte_thread_set_t *set);
```

  - **Cuando el thread no pertenece a ningún grupo `set` toma el valor `NULL_THREAD_SET`**

- **Comprobar si un thread pertenece a un grupo:**

```
int marte_threadset_ismember(
        const marte_thread_set_t *set,
        pthread_t thread_id,
        int * is_member);
```


Obtener el reloj de tiempo de ejecución de un *thread set*

```
int marte_getgroupcpuclockid(
        const marte_thread_set_t set,
        clockid_t *clock_id);
```

- **`clock_id` puede utilizarse como cualquier otro reloj POSIX:**
  - **`clock_gettime()`, `clock_settime()`**
  - **Temporizadores**
  - **…**

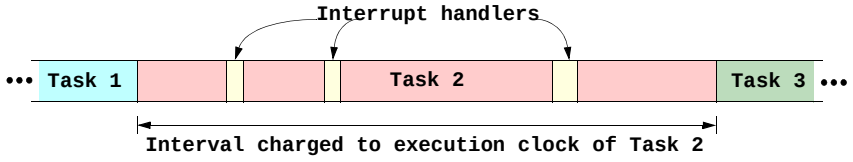
3.4 Tiempo de ejecución de las Interrupciones

Normalmente se asume que el efecto de los manejadores de interrupción en el tiempo de CPU de las tareas es despreciable

- **porque habitualmente los manejadores son muy cortos**

POSIX: deja como "definido por la implementación" la forma en la que el sistema contabiliza el tiempo de CPU de las interrupciones

Por simplicidad los SOs cargan el tiempo consumido por las interrupciones a la tarea que ejecuta en ese momento



Esta aproximación puede no ser realista en sistemas de tiempo real que hagan un uso intensivo de interrupciones

Por esta razón, el estándar Ada 2012

- **incluye un paquete opcional para medir separadamente el tiempo de ejecución de las interrupciones**

MaRTE OS proporciona un servicio parecido al del Ada 2012:

- **un reloj que mide el tiempo de ejecución consumido por los manejadores de interrupción**
- **ese tiempo no se contabiliza en los relojes de tiempo de ejecución de los threads**

Interfaz

Nuevo reloj en `time.h`: `CLOCK_INTERRUPTS_CPUTIME`

- **puede ser utilizado en la función `clock_gettime()` como cualquier otro reloj (no puede utilizarse con temporizadores)**
- **mide la suma de los tiempos de ejecución consumidos por _todos_ los manejadores de interrupción**

Configuración en `marte-configuration_parameters.ads`:

- **`Measure_Interrupt_Handlers_Time`:**
  - **se _mide el tiempo de ejecución de los manejadores_**
  - **ese tiempo se carga en el reloj `CLOCK_INTERRUPTS_CPUTIME`**
- **`Account_Interrupt_Handlers_Time_Separately`:**
  - **Implica `Measure_Interrupt_Handlers_Time`**
  - **además el tiempo de CPU de los manejadores _no se carga_ en los relojes de tiempo de ejecución de los threads**

Sobrecarga

Description	Envoltura de un manejador	
	Antes manejador	Después manejador
Functionality disabled (execution time of interrupt handlers charged to user tasks)	31ns	33ns
Only measure execution time of interrupt handlers	62ns	104ns
Measure execution time of interrupt handlers and account for it separately from the execution time of user tasks	62ns	130ns

Sobrecarga relativamente baja

- **Manejador de la interrupción del temporizador (3 μs)**
- **Manejador de la interrupción del teclado (75 μs)**