Master en Computación

Programación Concurrente

Bloque II: Programación concurrente en POSIX

Tema 1. Introducción al estándar POSIX
Tema 2. Sistema Operativo MaRTE OS
Tema 3. Gestión de Threads
Tema 4. Gestión del Tiempo
Tema 5. Planificación de Threads
Tema 6. Sincronización
Tema 7. Señales

Tema 8. Temporizadores y Relojes de Tiempo de Ejecución

Tema 8. Temporizadores y Relojes de Tiempo de Ejecución

8.1. Concepto de temporizador
8.2. Notificación de Eventos
8.3. Funciones para manejar temporizadores
8.4. Ejemplo: Threads Periódicos con temporizadores
8.5. Relojes y temporizadores de tiempo de ejecución
8.6. Ejemplo: limitación del tiempo de ejecución
8.7. Relojes de tiempo de ejecución para grupos de threads
8.8. Gestión de eventos temporales en MaRTE OS

8.1 Concepto de temporizador

Temporizador:

- un objeto que puede notificar a un proceso si ha transcurrido un cierto intervalo de tiempo o se ha alcanzado una hora determinada
- cada temporizador está asociado a un reloj

8.2 Notificación de Eventos

POSIX define un mecanismo para especificar el tipo de notificación deseada, que es común a varios servicios:

- **finalización de I/O asíncrona, *expiración de un temporizador*, llegada de un mensaje**

El tipo de notificación deseado se especifica mediante una estructura `sigevent`

- **el envío de una señal es uno de los tres posibles tipos de notificación que es posible indicar con esta estructura:**
  - **no hay notificación (`SIGEV_NONE`)**
  - **se genera una señal (`SIGEV_SIGNAL`)**
  - **se crea un thread y se ejecuta (`SIGEV_THREAD`) (no soportado en MaRTE OS)**

Campos de la estructura `sigevent`

```
struct sigevent {
  int sigev_notify;
  // Notificación: SIGEV_NONE, SIGEV_SIGNAL o
  // SIGEV_THREAD

  int sigev_signo;
  // Número de señal a enviar (SIGEV_SIGNAL)

  union sigval sigev_value;
  // Valor asociado a la señal (SIGEV_SIGNAL)
  // o argumento para el thread (SIGEV_THREAD)

  void (*sigev_notify_function)(union sigval);
  // Cuerpo del thread (SIGEV_THREAD)

  pthread_attr_t *sigev_notify_attributes;
  // Atributos del thread (SIGEV_THREAD)
};
```


8.3 Funciones para manejar temporizadores

- **Crear un temporizador:**

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

  - **crea un temporizador asociado al reloj `clock_id`**
  - **`*evp` indica la notificación deseada: ninguna, enviar una señal con información, o crear y ejecutar un thread**
  - **en `*timerid` se devuelve el identificador del temporizador**
  - **el temporizador se crea en estado "desarmado"**
  - **el temporizador es visible para el proceso que lo creó**

- **Borrar un temporizador**

```
int timer_delete (timer_t timerid);
```

Armar un temporizador

Se utiliza la función:

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

```
struct itimerspec {
  struct timespec it_value;    // primera expiración
  struct timespec it_interval; // periodo entre
}                              //      sucesivas expiraciones
```

- si `value.it_value = 0` el temporizador se desarma
- si `value.it_value > 0` el temporizador se arma, y su valor se hace igual a `value`
  - si el temporizador ya estaba armado, se rearma
- si `value.it_interval > 0` el temporizador es periódico después de la primera expiración

Armar un temporizador (cont.)

- `flag` indica si el temporizador es absoluto o relativo:
  - si `TIMER_ABSTIME` se especifica, la primera expiración será cuando el reloj valga `value.it_value`
  - si no se especifica, la primera expiración será cuando transcurra un intervalo igual a `value.it_value`
- cada vez que el temporizador expira, se envía la notificación solicitada en `*evp` al crearlo
- si `ovalue` no es `NULL` se devuelve el valor anterior

Leer el estado de un temporizador

- Leer el valor de un temporizador:

```
int timer_gettime (timer_t timerid,
                   struct itimerspec *value);
```

  - significado del valor retornado en `value`:
    - `value.it_value`: tiempo que falta para que expire
    - `value.it_interval`: periodo del temporizador
- Leer el número de expiraciones no notificadas:

```
int timer_getoverrun (timer_t timerid);
```

  - el temporizador sólo mantiene una señal pendiente, aunque haya muchas expiraciones
  - el número de expiraciones no notificadas se puede saber mediante esta función

8.4 Ejemplo: Threads Periódicos con temporizadores

```
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <pthread.h>
#include <unistd.h>
#include <misc/error_checks.h>


struct periodic_data {
  struct timespec per;
  int sig_num;
};
```


Ejemplo: Threads Periódicos con temporizadores (cont.)

```
// Thread periódico que crea un temporizador periódico
void * periodic (void *arg)
{
  struct periodic_data my_data;
  int received_sig;
  struct itimerspec timerdata;
  timer_t timer_id;
  struct sigevent event;
  sigset_t set;

  my_data = * (struct periodic_data*)arg;

  // crea el temporizador
  event.sigev_notify = SIGEV_SIGNAL;
  event.sigev_signo = my_data.sig_num;
  CHKE( timer_create(CLOCK_MONOTONIC, &event, &timer_id) );

  // programa el temporizador (expiración periódica)
  timerdata.it_interval = my_data.per;
  timerdata.it_value = my_data.per;
  CHKE( timer_settime(timer_id, 0, &timerdata, NULL) );
```


Ejemplo: Threads Periódicos con temporizadores (cont.)

```
  // conjunto de señales esperadas por sigwaitinfo
  sigemptyset (&set);
  sigaddset(&set,my_data.sig_num);

  // La mascara de señales vendrá fijada por el padre

  // espera la expiración del temporizador y realiza el trabajo útil
  while (1) {
    CHKE( sigwait(&set, &received_sig) );

    printf("Thread con periodo sec=%d nsec=%d activo \n",
           my_data.per.tv_sec,my_data.per.tv_nsec);
  }
}
```

Ejemplo: Threads Periódicos con temporizadores (cont.)

```
// Programa principal, que crea dos threads periódicos
int main() {
  pthread_t t1,t2;
  sigset_t set;
  struct periodic_data per_params1,per_params2;

  // pone la máscara de señales
  sigemptyset(&set);
  sigaddset(&set, SIGRTMIN);
  sigaddset(&set, SIGRTMIN+1);
  CHK( pthread_sigmask(SIG_BLOCK, &set, NULL) );

  // crea el primer thread
  per_params1.per.tv_sec = 0; per_params1.per.tv_nsec = 500000000;
  per_params1.sig_num = SIGRTMIN;
  CHK( pthread_create (&t1, NULL, periodic, &per_params1) );

  // crea el segundo thread
  per_params2.per.tv_sec = 1; per_params2.per.tv_nsec = 500000000;
  per_params2.sig_num = SIGRTMIN+1;
  CHK( pthread_create (&t2, NULL, periodic, &per_params2) );
  sleep(30);
  exit(0);
}
```


© M. Aldea, M. González
oct-13

8.5 Relojes y temporizadores de tiempo de ejecución

Se asocian tanto a procesos como a threads:

- **permiten monitorizar el tiempo de ejecución o de CPU**
  - **todas las técnicas de análisis de tiempo real se basan en los tiempos de ejecución de peor caso**
- **permiten detectar un consumo excesivo de tiempo de CPU**
  - **mediante la interfaz de temporizadores**
  - **cuando un temporizador expira, envía una señal**

Esta limitación de tiempos de ejecución era habitual en los ejecutivos cíclicos

Los temporizadores de tiempo de CPU permiten el mismo grado de fiabilidad en sistemas POSIX

© M. Aldea, M. González
oct-13

Interfaz para relojes de tiempo de CPU

Constantes del tipo `clockid_t` que identifican el reloj del thread o proceso actual:

```
CLOCK_THREAD_CPUTIME_ID
CLOCK_PROCESS_CPUTIME_ID
```

Funciones para obtener el identificador de reloj de un proceso o thread cualquiera:

```
#include <pthread.h>
#include <time.h>
int clock_getcpuclockid(pid_t pid,
                        clockid_t *clock_id);

int pthread_getcpuclockid(pthread_t thread_id,
                          clockid_t *clock_id);
```


© M. Aldea, M. González
oct-13

8.6 Ejemplo: limitación del tiempo de ejecución

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


// Argumento de los threads
struct periodic_data {
  struct timespec period;
  struct timespec wcet;
  int id;
  int prio;
  pthread_t * tid;
};
```



```
// Thread que actúa como vigilante
// Baja la prioridad de un thread si excede el tiempo de ejecución
// (La mascara de señales se hereda del padre)
void * vigilante (void *arg) {
  sigset_t set;
  siginfo_t info;

  // se pone a la máxima prioridad
  CHK( pthread_setschedprio(pthread_self(),
                          sched_get_priority_max(SCHED_FIFO)) );

  // lazo de espera de señales
  sigemptyset(&set);
  sigaddset(&set,SIGRTMIN);
  while (1) {
    // espera la señal
    CHKE( sigwaitinfo(&set, &info) );

    // cuando llega la señal baja la prioridad del thread implicado
    CHK( pthread_setschedprio
            (*((struct periodic_data *)info.si_value.sival_ptr)->tid,
            sched_get_priority_min(SCHED_FIFO)) );
  }
}
```



```
// Thread periódico que crea un timer de tiempo de ejecución
void * periodic (void *arg) {
  struct periodic_data my_data;
  struct itimerspec timerdata;
  timer_t cpu_timer_id;
  struct sigevent event;
  struct sched_param sch_param; int policy;
  struct timespec next_time,zero={0,0};

  timerdata.it_interval = zero; timerdata.it_value = my_data.wcet;

  my_data = * (struct periodic_data*)arg;

  // pone la prioridad al valor correcto
  CHK( pthread_setschedprio(pthread_self(), my_data.prio) );

  // crea el temporizador de tiempo de ejecución
  event.sigev_notify = SIGEV_SIGNAL;
  event.sigev_signo = SIGRTMIN;
  event.sigev_value.sival_ptr = &my_data;
  CHKE( timer_create(CLOCK_THREAD_CPUTIME_ID, &event,
                     &cpu_timer_id) );
```

```
   // lazo de activaciones periódicas
   clock_gettime(CLOCK_MONOTONIC, &next_time);
   while (1) {
     // arma el temporizador de tiempo de ejecución
     CHKE( timer_settime(cpu_timer_id, 0, &timerdata, NULL) );
     // simula la realización de trabajo útil
     printf("Thread con Id=%d comienza\n",my_data.id);
     eat((float)my_data.wcet.tv_sec +
        ((float)my_data.wcet.tv_nsec) / 1.0e9 - 0.1);
     // mostramos el final de la ejecución, con la prioridad actual
     CHK( pthread_getschedparam(pthread_self(),&policy,&sch_param) );
     printf("Thread con Id=%d termina con prio=%d\n",
            my_data.id,sch_param.sched_priority);
     // pone la prioridad al valor correcto, por si acaso se la
     // hubiera bajado el vigilante
     CHK( pthread_setschedprio(pthread_self(), my_data.prio) );
     // espera al próximo periodo
     incr_timespec(&next_time, &my_data.period);
     CHK( clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                          &next_time, NULL) );
   }
 }
```



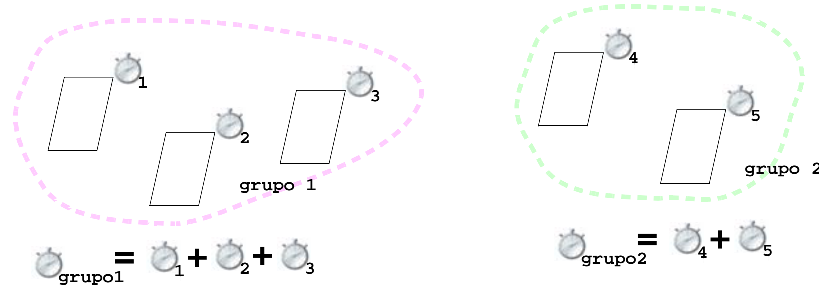
```
 // Programa principal, que crea dos threads periódicos
 int main () {
   pthread_t tv,t1,t2;
   sigset_t set;
   struct periodic_data per_params1,per_params2;
   struct sigaction sigact;

   adjust();

   // configura la señal para que tenga comportamiento de tiempo real
   sigact.sa_handler=SIG_DFL;
   sigact.sa_flags=SA_SIGINFO;
   CHKE (sigaction(SIGRTMIN, &sigact , NULL) );

   // pone la máscara de señales
   sigemptyset(&set);
   sigaddset(&set, SIGRTMIN);
   CHK( pthread_sigmask(SIG_BLOCK, &set, NULL) );
```



```
   // crea el thread vigilante
   CHK( pthread_create (&tv, NULL, vigilante, NULL) );

   // crea el primer thread periódico
   per_params1.period.tv_sec=2; per_params1.period.tv_nsec=500000000;
   per_params1.wcet.tv_sec=0; per_params1.wcet.tv_nsec=500000000;
   per_params1.id=1;
   per_params1.tid=&t1;
   per_params1.prio = sched_get_priority_min(SCHED_FIFO)+3;
   CHK( pthread_create (&t1, NULL, periodic, &per_params1) );

   // crea el segundo thread periódico
   per_params2.period.tv_sec=7; per_params2.period.tv_nsec=0;
   per_params2.wcet.tv_sec=2; per_params2.wcet.tv_nsec=500000000;
   per_params2.id=2;
   per_params2.tid=&t2;
   per_params2.prio=sched_get_priority_min(SCHED_FIFO)+2;
   CHK( pthread_create (&t2, NULL, periodic, &per_params2) );

   // permite ejecutar un rato a los threads
   sleep(30);
   exit(0);
 }
```

8.7 Relojes de tiempo de ejecución para grupos de threads

Miden el tiempo de ejecución consumido por un grupo de threads



La interfaz 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:**

```
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*

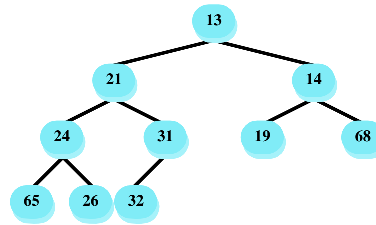
```
int marte_getgroupcpuclockid(
        const marte_thread_set_t set,
        clockid_t *clock_id);
```

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

8.8 Gestión de eventos temporales en MaRTE OS

Programación del temporizador hardware

- Modelo "despertador": se programa para el siguiente evento
- Mejores prestaciones que con el modelo de interrupción periódica



Cola de eventos temporales

- Todos los eventos (*timers*, activación threads, etc.) ordenados por instante de activación
- Montículo binario (`O(log n)`)

Gestión de eventos temporales en MaRTE OS

Manejador de la interrupción del Timer

```
procedure Intr_Timer is
begin
  while(evento más urgente ha expirado)
    Extrae evento más urgente;
    case (Tipo evento) is
    when FIN_SUSPENSION =>
      Activa tarea;
    when EXPIRA_TIMER    =>
      Lanza señal;
    when ... =>
      ...;
    end case;
  end while;
  reprograma temporizador para
                  el evento más urgente;
  llama al planificador;
end Intr_Timer;
```

MaRTE OS

Lista de eventos temporales ordenados por urgencia

Interrupción

PIT o Local-APIC

Temporizador del PC

Hardware (PC)