Bloque I: Principios de sistemas operativos

Notas:

Tema 6. Sistemas de tiempo real y sistemas empotrados

- Concepto de sistema de tiempo real
- Concepto y características de sistemas empotrados
- Políticas de planificación para tiempo real
- Protocolos de sincronización de tiempo real
- Análisis de sistemas de tiempo real
- Relojes y temporizadores POSIX

1. Concepto de sistema de tiempo real

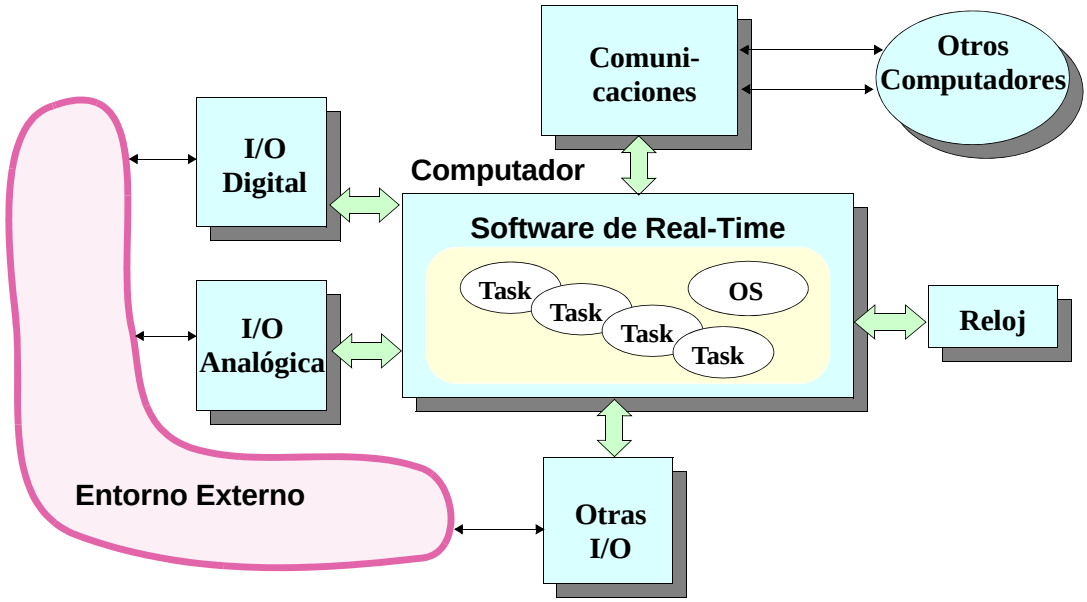
Un sistema de tiempo real es una combinación de uno o varios computadores, dispositivos hardware de entrada/salida, y software de propósito especial, en la que:

- **hay una fuerte interacción con el entorno**
- **el entorno cambia con el tiempo**
- **el sistema controla o reacciona de forma simultánea a diferentes aspectos del entorno**

Como resultado:

- **se imponen requerimientos temporales sobre el software**
- **el software es de naturaleza concurrente**

Elementos de un sistema de tiempo real



Definición de sistema de tiempo real

"En las aplicaciones de tiempo real, el funcionamiento correcto no sólo depende de los resultados del cálculo, sino también del instante en el que se generan estos resultados."

Algunos requerimientos de tiempo real frecuentes:

- **Realizar actividades periódicas:**
  - **solicitadas a intervalos periódicos**
  - **deben completar un trabajo antes de un plazo**
- **Responder a eventos aperiódicos:**
  - **solicitados a intervalos irregulares**
  - **en algunos casos, completar el trabajo antes de un plazo**
  - **alcanzar un requerimiento de tiempo de respuesta promedio**

¿Qué es importante en sistemas de tiempo real?

Los criterios para tiempo real difieren de los de sistemas de tiempo compartido

	Tiempo Compartido	Sist. de Tiempo Real
Capacidad	Potencia de cálculo	Planificabilidad
Respuesta Temporal	Rápida respuesta en pro-medio	Respuesta de peor caso garantizada
Sobrecarga	Reparto Equitativo	Estabilidad

- ***Planificabilidad*: habilidad para cumplir todos los plazos**
- ***Respuesta de peor caso*: no basta el caso promedio**
- ***Estabilidad*: en una sobrecarga, se deben de cumplir al menos los plazos de todas las tareas críticas**

Sistemas operativos de tiempo real

En el pasado, muchos sistemas de tiempo real no necesitaban sistema operativo.

Hoy en día, muchas aplicaciones requieren servicios de sistema operativo tales como:

- programación concurrente, redes de comunicación, sistema de ficheros, etc.

El comportamiento temporal de un programa depende fuertemente del comportamiento del sistema operativo.

Definición de tiempo real en sistemas operativos, según POSIX: "La habilidad del sistema operativo para proporcionar el nivel de servicio requerido con un tiempo de respuesta acotado."

2. Concepto y características de sistemas empotrados

Un computador (y su software) se considera empotrado si:

- es un componente integral de un sistema mayor,
- se usa para controlar, monitorizar o procesar la información de ese sistema,
- y usa dispositivos hardware especiales

La mayoría de los sistemas de tiempo real son empotrados

Al revés no; por ejemplo, en estos sistemas no importa el tiempo de respuesta de peor caso:

- llave electrónica
- muñeco que reacciona al entorno con gestos y sonidos

3. Políticas de planificación para tiempo real

Planificadores en tiempo de compilación:

- conocidos como ejecutivos cíclicos (ver “la solución cíclica”, en el capítulo 2)
- predecibilidad mediante un plan estático
- la estructura lógica depende de los requisitos temporales
- difícil de mantener

Planificadores en tiempo de ejecución:

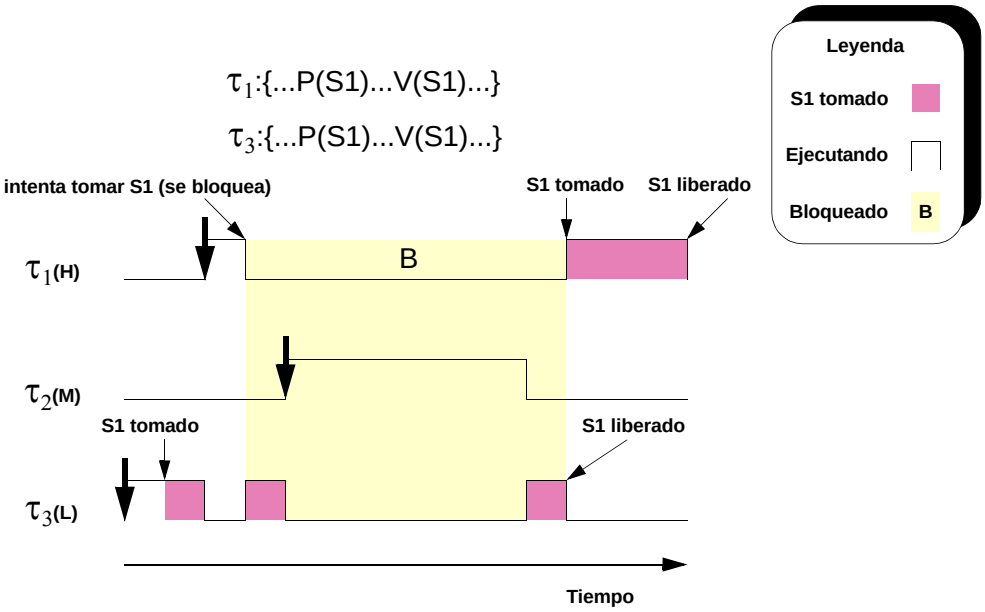
- Basados en prioridades
- Expulsores o no, prioridades fijas o dinámicas
- Predecibilidad mediante métodos analíticos
- Separan la estructura lógica de los requisitos temporales

Planificación por prioridades fijas

La planificación expulsora por prioridades fijas es la más popular:

- El comportamiento temporal es más fácil de entender y predecir
- El tratamiento ante sobrecargas es fácil de prever
- Existen técnicas analíticas completas
- Requerido por estándares de sistemas operativos y lenguajes concurrentes

4. Sincronización de tiempo real: Inversión de prioridad



Protocolos de sincronización

Los protocolos de sincronización de tiempo real evitan la inversión de prioridad no acotada

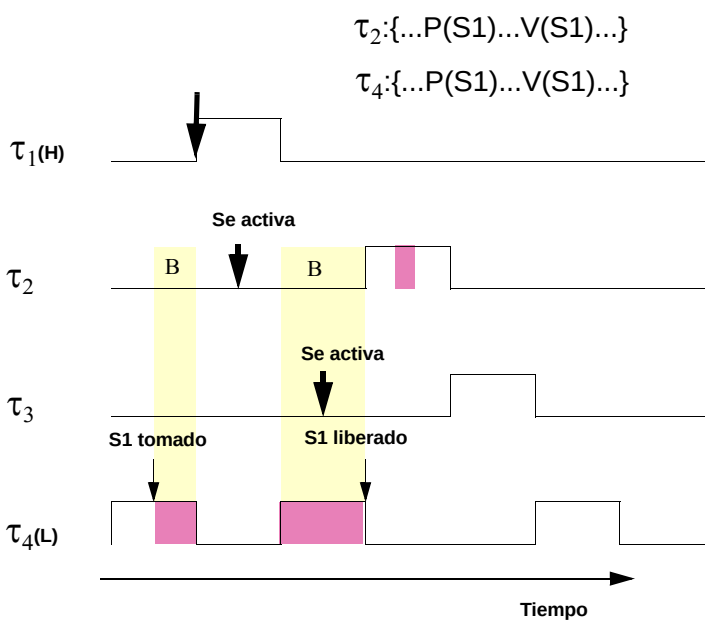
Los más importantes:

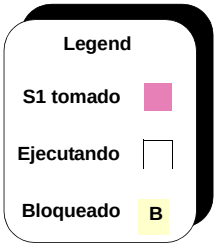
- **Herencia de prioridad**
- **Techo de prioridad inmediato (o protección por prioridad)**

Ambos hacen que la inversión de prioridad, o retraso que una tarea sufre a causa de tareas de prioridad inferior:

- **sea función de la duración de una o varias secciones críticas**
- **no sea función de la duración de tareas completas**

Techo de prioridad inmediato





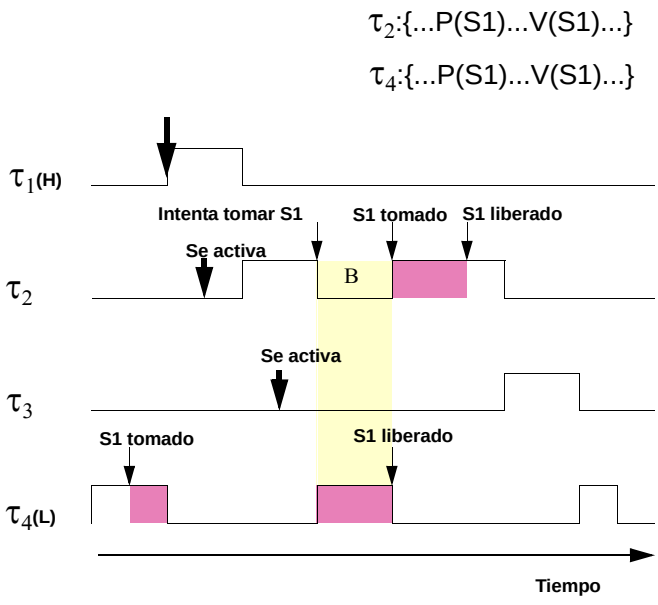
Techo de prioridad de un recurso:

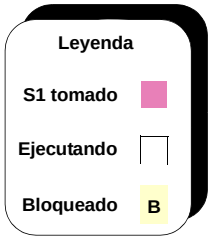
- la mayor prioridad de las tareas que lo usan

Regla:

- la sección crítica se ejecuta al techo de prioridad

Herencia de prioridad





Regla:

- las tareas heredan las prioridades de aquellas tareas que bloquean

Protocolos de sincronización en POSIX

Atributos de inicialización de mutexes (adicionales a *pshared*):

- *protocol*: protocolo utilizado
  - `PTHREAD_PRIO_NONE`: no hay herencia de prioridad
  - `PTHREAD_PRIO_INHERIT`: herencia de prioridad
  - `PTHREAD_PRIO_PROTECT`: protección de prioridad (techo de prioridad inmediato)
- *prioceiling*: techo de prioridad
  - valor entero; se puede modificar en ejecución
  - se usa para la protección de prioridad

Estos atributos se almacenan en el objeto de atributos

Atributos de planificación de los mutex

```
#include <pthread.h>

int pthread_mutexattr_getprotocol
      (const pthread_mutexattr_t *attr,
       int *protocol);

int pthread_mutexattr_setprotocol
      (pthread_mutexattr_t *attr,
       int protocol);

int pthread_mutexattr_getprioceiling
      (const pthread_mutexattr_t *attr,
       int *prioceiling);

int pthread_mutexattr_setprioceiling
      (pthread_mutexattr_t *attr,
       int prioceiling);
```

5. Análisis de sistemas de tiempo real

El análisis “Rate Monotonic” (RMA) es una teoría para analizar el comportamiento temporal de aplicaciones de tiempo real basadas en prioridades fijas

Proporciona directrices para hacer asignación óptima de prioridades

Aporta tests de utilización, que nos permiten hacernos una idea aproximada sobre el cumplimiento de los plazos

Permite obtener los tiempos de respuesta exactos de las tareas

Ayuda a identificar “cuellos de botella”, para mejorar el diseño

Principios básicos del RMA

Dos conceptos ayudan a construir la situación de peor caso en el caso de tareas periódicas independientes:

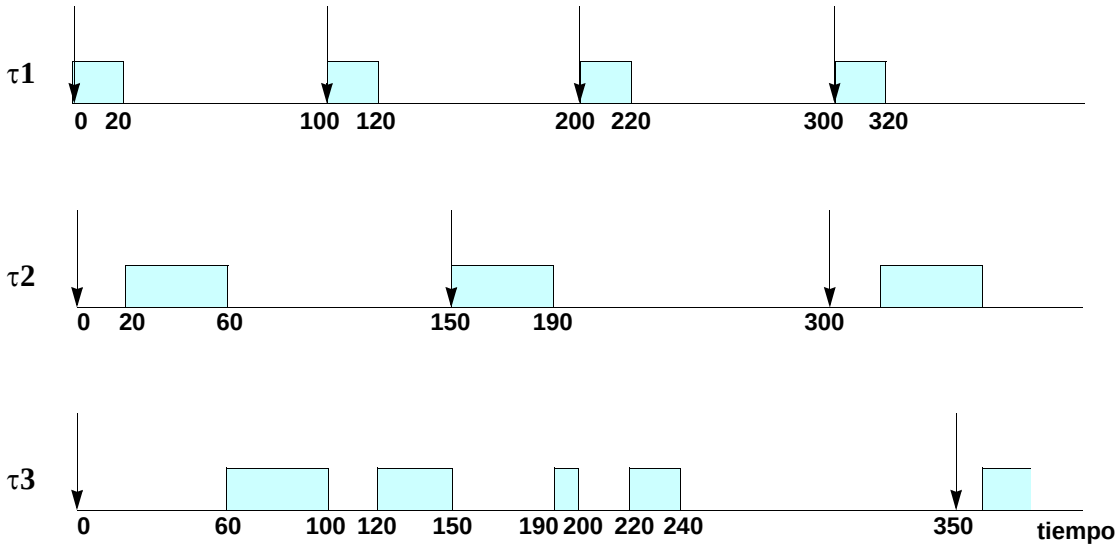
- *Instante crítico*. El tiempo de respuesta de peor caso de todas las tareas se obtiene si las activamos todas a la vez
- *Basta comprobar el primer plazo*. Después de un instante crítico, si una tarea cumple su primer plazo cumplirá todos los plazos posteriores

Basándose en estos conceptos se obtienen estos resultados:

- Asignación óptima de prioridades
- Test de utilización
- Test exacto, o de tiempos de respuesta

Ejemplo de sistema periódico

Ejemplo de instante crítico



Prioridades óptimas

Para un conjunto de tareas con estas restricciones:

- **periódicas,**
- **independientes,**
- **plazos = periodos,**
- **planificación expulsora por prioridades fijas,**

la asignación óptima de prioridades es la "*rate monotonic*":

- **mayor prioridad a menor periodo**

***Óptimo* significa que si el sistema es planificable (cumple sus plazos) con una asignación determinada, también lo es con la asignación óptima.**

Test de utilización

***Test de utilización*: Si hay *n* tareas con las restricciones anteriores y con la asignación de prioridades "rate monotonic", serán planificables si**

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq U(n) = n(2^{1/n} - 1)$$

U(1) = 1.0	U(4) = 0.756	U(7) = 0.728
U(2) = 0.828	U(5) = 0.743	U(8) = 0.724
U(3) = 0.779	U(6) = 0.734	U(∞) = 0.693

Este test es condición suficiente, pero no necesaria, para la planificabilidad

Plazos menores al periodo

Asignación óptima "deadline monotonic":

- mayor prioridad al que tiene menor plazo
- la asignación "rate monotonic" es un caso particular

Criterios para análisis

- instante crítico: ***válido***
- basta comprobar el primer plazo: ***válido***
- test de utilización: ***no es válido***
- test exacto o de tiempos de respuesta: ***válido***

Cálculo de tiempos de bloqueo

Cuando hay sincronización de acceso mutuamente exclusivo hay que añadir a la tarea *i* el *tiempo de bloqueo* B_i:

- es el tiempo máximo de retraso que sufre una tarea por culpa de todas las demás tareas de prioridad inferior

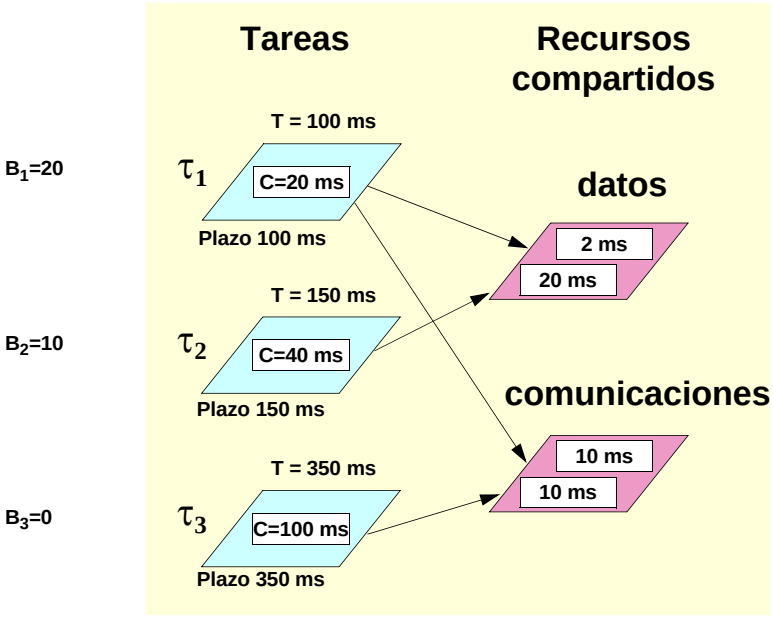
Para el protocolo de techo de prioridad inmediato:

- B_i es la duración máxima de todas las secciones críticas cuyo techo es $\geq P_i$ (prioridad de la tarea *i*)

El test de utilización cambia y debe hacerse para cada tarea *i*:

$$\frac{C_1}{T_1} + \ldots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \leq U(i) = i(2^{1/i} - 1)$$

Ejemplo



Modelo de análisis bajo techo inmediato de prioridad

$\frac{C_1}{T_1} + \frac{B_1}{T_1} \leq U(1)$	$\frac{20}{100} + \frac{20}{100} = 0{,}40 < 1{,}0$
$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{B_2}{T_2} \leq U(2)$	$\frac{20}{100} + \frac{40}{150} + \frac{10}{150} = 0{,}534 < 0{,}828$
$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} < U(3)$	$\frac{20}{100} + \frac{40}{150} + \frac{100}{350} = 0{,}753 < 0{,}779$

6. Relojes y temporizadores POSIX

Reloj:

- objeto que mide el paso del tiempo

Resolución de un reloj:

- el intervalo de tiempo más pequeño que el reloj puede medir

La Época:

- las 0 horas, 0 minutos, 0 segundos del 1 de enero de 1970, UTC (Coordinated Universal Time)
- segundos desde la Época = sec + min*60 + hour*3600 + yday*86400 + (year-70)*31536000 + ((year-69)/4)*86400

Relojes y temporizadores POSIX (cont.)

Reloj del sistema:

- mide los segundos transcurridos desde la Época
- se usa para marcar las horas de creación de ficheros, etc.
- es global al sistema

Reloj de Tiempo Real

- reloj que mide el tiempo transcurrido desde la Época
- se usa para timeouts y para crear temporizadores
- puede coincidir o no con el reloj del sistema
- es global al sistema
- resolución máxima: 20 ms. ¡Precaución!
- resolución mínima: 1 nanosegundo

Relojes y temporizadores POSIX (cont.)

Temporizador:

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

Relojes de Tiempo Real

Se define el tipo `timespec` para definir el tiempo con alta resolución:

```
struct timespec {
     time_t tv_sec; // segundos
     long tv_nsec;} // nanosegundos
```

- tiempo = $tv\_sec*10^9+tv\_nsec$
- $0 \leq tv\_nsec < 10^9$

Se define el tipo `clockid_t`, cuyos valores identifican relojes

Se define el reloj de tiempo real `CLOCK_REALTIME` como una constante del tipo `clockid_t`

La máxima resolución permisible es de 20 ms

Funciones para Manejar Relojes

Cambiar la hora:

```
#include <time.h>
int clock_settime
    (clockid_t clock_id,
     const struct timespec *tp);
```

Leer la hora:

```
int clock_gettime
    (clockid_t clock_id,
     struct timespec *tp);
```

Leer la resolución del reloj:

```
int clock_getres
    (clockid_t clock_id,
     struct timespec *res);
```


Temporizadores

Crear un temporizador:

```
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

Temporizadores (cont.)

Borrar un temporizador

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

Armar un temporizador:

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

- la estructura `itimerspec` tiene:
  - `struct timespec it_interval`: periodo
  - `struct timespec it_value`: tiempo de expiración
- 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`

Temporizadores (cont.)

Armar un temporizador (cont.):

- si el temporizador ya estaba armado, se rearma
- `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`
- si `value.it_interval` > 0 el temporizador es periódico después de la primera expiración
- 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

Temporizadores (cont.)

Leer el valor de un temporizador:

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

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

Funciones *sleep*

Dormir un proceso o thread:

```
unsigned int sleep
    (unsigned int seconds);
```

- el proceso o thread se suspende hasta que transcurren los segundos indicados, o se ejecuta un manejador de señal
- la función devuelve 0 si duerme el intervalo solicitado, y el número de segundos que faltan para completar el intervalo si vuelve a causa de una señal.

Sleep de alta resolución

Relativo:

```
int nanosleep (const struct timespec *rqtp,
               struct timespec *rmtp);
```

- `*rqtp` es el tiempo a suspenderse
- si retorna por una señal, el tiempo restante va en `*rmtp`

Absoluto o relativo, con especificación del reloj:

```
int clock_nanosleep
      (clockid_t clock_id,
       int flags, const struct timespec *rqtp,
       struct timespec *rmtp);
```

- `clock_id` es el identificador del reloj a usar
- `flags` especifica opciones; si la opción `TIMER_ABSTIME` está, es absoluto; si no, relativo

Ejemplo: Threads Periódicos

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

static int error_status=-1;

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

Ejemplo (cont)

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

  my_data = * (struct periodic_data*)arg;
  event.sigev_notify = SIGEV_SIGNAL;
  event.sigev_signo = my_data.sig_num;
  if (timer_create (CLOCK_REALTIME, &event, &timer_id) == -1) {
    perror("error de creacion del timer\n");
    pthread_exit((void *)&error_status);
  }
  timerdata.it_interval = my_data.per;
  timerdata.it_value = my_data.per;
```


Ejemplo (cont)

```
  if (timer_settime(timer_id, 0, &timerdata, NULL) == -1) {
    perror("error en timer_settime\n");
    pthread_exit((void *)&error_status);
  }

  sigemptyset (&set);
  sigaddset(&set,my_data.sig_num);

  // La mascara de señales vendrá fijada por el padre

  while (1) {
    if (sigwaitinfo(&set,&received_sig) == -1) {
      perror("sigwait error");
      pthread_exit((void *)&error_status);
    }
    printf("Thread con periodo sec=%ld nsec=%ld activo\n",
    my_data.per.tv_sec,my_data.per.tv_nsec);
  }
}
```

Ejemplo (cont)

```
// Programa principal, que crea dos threads periódicos

int main ()
{
  pthread_t t1,t2;
  sigset_t set;
  struct periodic_data per_params1,per_params2;

  sigemptyset(&set);
  sigaddset(&set,SIGRTMIN);
  sigaddset(&set,SIGRTMIN+1);
  sigprocmask(SIG_BLOCK, &set, NULL);
```


Ejemplo (cont)

```
  per_params1.per.tv_sec=0;
  per_params1.per.tv_nsec=500000000;
  per_params1.sig_num=SIGRTMIN;
  if (pthread_create (&t1,NULL,periodic,&per_params1) != 0) {
    printf("Error en creacion de thread\n");
  }

  per_params2.per.tv_sec=1;
  per_params2.per.tv_nsec=500000000;
  per_params2.sig_num=SIGRTMIN+1;
  if (pthread_create (&t2,NULL,periodic,&per_params2) != 0) {
    printf("Error en creacion de thread\n");
  }
  sleep(30);
  exit(0);
}
```