

Modelos y Herramientas de Diseño de Tiempo Real

Máster en Ingeniería Informática

Material complementario: POSIX y MaRTE OS

Página Web (Moodle):

<https://moodle.unican.es/course/view.php?id=16319>

Curso 2022-2023

Threads: conceptos básicos

Thread:

- un flujo de control simple perteneciente a un proceso
- tiene un identificador de thread (*tid*)
- el *tid* sólo es válido para threads del mismo proceso
- tiene su propia política de planificación, y los recursos del sistema necesarios, tales como su propio stack, etc
- todos los threads de un proceso comparten un único espacio de direccionamiento

Proceso en una implementación multi-thread:

- un espacio de direccionamiento con uno o varios threads
- inicialmente contiene un solo thread: el thread principal

Threads: conceptos básicos (cont.)

Creación de un proceso (*fork*):

- se crea con un único thread
- si el proceso que llama es multi-thread sólo puede hacer llamadas “seguras”, hasta realizar un *exec*

Ejecución de un programa:

- todos los threads del proceso original se destruyen
- se crea también con un único thread

Servicios bloqueantes:

- en una implementación multithread, sólo se suspende el thread que invoca el servicio

Threads: conceptos básicos (cont.)

Los threads tienen dos estados posibles para controlar la devolución de recursos al sistema:

- “*detached*” o independiente: `PTHREAD_CREATE_DETACHED`
 - cuando el thread termina, devuelve al sistema los recursos utilizados (tid, stack, etc)
 - no se puede esperar la terminación de este thread con `pthread_join()`
- “*joinable*” o sincronizado: `PTHREAD_CREATE_JOINABLE`
 - cuando el thread termina, mantiene sus recursos
 - los recursos se liberan cuando el thread termina, y otro thread llama a `pthread_join()`
 - este es el valor por defecto

Creación de threads

Para crear un thread es preciso definir sus atributos en un objeto especial

El objeto de atributos

- debe crearse antes de usarlo: *pthread_attr_init()*
- puede borrarse: *pthread_attr_destroy()*
- se pueden modificar o consultar atributos concretos del objeto (pero no los del thread, que se fijan al crearlo)

Los atributos definidos son:

- tamaño de stack mínimo (opcional)
- dirección del stack (opcional)
- control de devolución de recursos ("detach state")

Atributos de creación: interfaz

```
#include <pthread.h>
int pthread_attr_init (pthread_attr_t *attr);
int pthread_attr_destroy (pthread_attr_t *attr);

int pthread_attr_setstacksize (pthread_attr_t *attr,
                               size_t stacksize);
int pthread_attr_getstacksize (const pthread_attr_t *attr,
                               size_t *stacksize);

int pthread_attr_setstackaddr (pthread_attr_t *attr,
                               void *stackaddr);
int pthread_attr_getstackaddr (const pthread_attr_t *attr,
                               void **stackaddr);

int pthread_attr_setdetachstate (pthread_attr_t *attr,
                                 int detachstate);
int pthread_attr_getdetachstate (const pthread_attr_t *attr,
                                 int *detachstate);
```

Llamada para creación de threads

Interfaz:

```
int pthread_create (pthread_t *thread,  
                    const pthread_attr_t *attr,  
                    void *(*start_routine) (void *),  
                    void *arg);
```

Esta función:

- crea un nuevo thread con atributos especificados por *attr*.
- devuelve el *tid* del nuevo thread en *thread*
- el thread se crea ejecutando *start_routine(arg)*
- si *start_routine* termina, es equivalente a llamar a *pthread_exit* (observar que esto difiere del thread *main*)

Ejemplo de creación de threads

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

// Thread que pone periódicamente un mensaje en pantalla
// el periodo se le pasa como parámetro
void *periodic (void *arg) {
    int period;

    period = *((int *)arg);
    while (1) {
        printf("En el thread con periodo %d\n",period);
        sleep (period);
    }
}
```


Ejemplo de creación de threads (cont.)

```
// Programa principal que crea dos threads periódicos
int main ()
{
    pthread_t th1, th2;
    pthread_attr_t attr;
    int period1=2
    int period2=3;
    // Crea el objeto de atributos
    if (pthread_attr_init(&attr) != 0) {
        printf("error de creación de atributos\n");
        exit(1);
    }
    // Crea los threads
    if (pthread_create(&th1, &attr, periodic, &period1) != 0) {
        printf("error de creación del thread uno\n");
        exit(1);
    }
}
```

Ejemplo de creación de threads (cont.)

```
if (pthread_create(&th2,&attr,periodic,&period2) != 0) {  
    printf("error de creación del thread dos\n");  
    exit(1);  
}  
// Les deja ejecutar un rato y luego termina  
sleep(30);  
printf("thread main terminando\n");  
exit (0);  
}
```

Terminación de threads

Función para terminación de threads:

```
#include <pthread.h>  
void pthread_exit (void *value_ptr);
```

Esta función termina el thread y hace que el valor apuntado por *value_ptr* esté disponible para una operación *join*

- se ejecutan las rutinas de cancelación pendientes
- al terminar un thread es un error acceder a sus variables locales
- cuando todos los threads de un proceso se terminan, el proceso se termina (como si se hubiera llamado a *exit*)

Terminación de threads (cont.)

Se puede esperar (*join*) a la terminación de un thread cuyo estado es sincronizado (*joinable*), liberándose sus recursos:

```
#include <pthread.h>
int pthread_join (pthread_t thread,
                  void **value_ptr);
```

También se puede cambiar el estado del thread a “detached”, con lo que el thread, al terminar, libera sus recursos:

```
#include <pthread.h>
int pthread_detach (pthread_t thread);
```

Existen también funciones para cancelar threads, habilitar o inhibir la cancelación, etc. (ver el manual).

Ejemplo

```
#include <pthread.h>
#include <stdio.h>
#define MAX    500000
#define MITAD  250000

typedef struct {
    int *ar;
    long n;
} subarray;

// Thread que incrementa n componentes de un array
void * incrementer (void *arg) {
    long i;

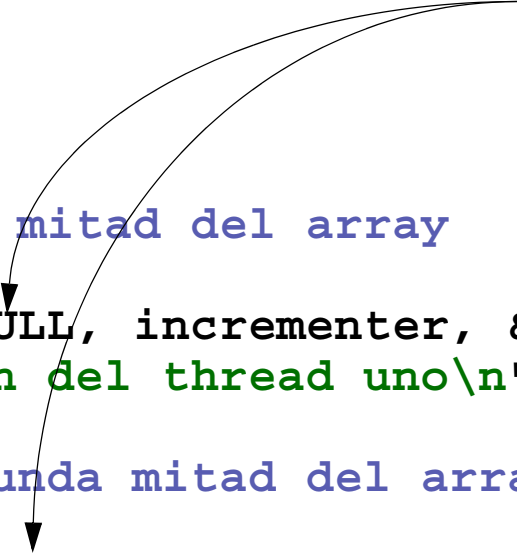
    for (i=0; i< ((subarray *)arg)->n; i++) {
        ((subarray *)arg)->ar[i]++;
    }
    pthread_exit(NULL);
}
```

Ejemplo (cont.)

```
// programa principal que reparte el trabajo de incrementar
// los componentes de un array entre dos threads
int main() {
    int ar [MAX];
    pthread_t th1, th2;
    subarray sb1, sb2;
    void *st1, *st2;
    long suma=0, i;

    sb1.ar = &ar[0]; // primera mitad del array
    sb1.n = MITAD;
    if (pthread_create(&th1, NULL, incrementer, &sb1) != 0) {
        printf("error de creacion del thread uno\n"); exit(1);
    }
    sb2.ar = &ar[MITAD]; // segunda mitad del array
    sb2.n = MITAD;
    if (pthread_create(&th2, NULL, incrementer, &sb2) != 0) {
        printf("error de creacion del thread dos\n"); exit(1);
    }
}
```

atributos por defecto



Ejemplo (cont.)

```
// sincronizacion de espera a la finalizacion

if (pthread_join(th1, &st1) != 0) {
    printf ("join error\n"); exit(1);
}
if (pthread_join(th2, (void **)&st2) != 0) {
    printf ("join error\n"); exit(1);
}
printf ("main termina; status 1=%d; status 2=%d\n",
        (int) st1, (int) st2);

for (i=0; i<MAX; i++) {
    suma=suma+ar[i];
}
printf ("Suma=%d\n", suma);
exit(0);
}
```

Identificación de threads

Identificación del propio thread:

```
pthread_t pthread_self(void);
```

Comparación de *tids*:

```
int pthread_equal (pthread_t t1, pthread_t t2);
```


Planificación de threads

Atributos de planificación:

- Son parte del objeto de atributos que se utiliza al crear el thread
- Ámbito de contención (`contentionscope`); valores:
 - ámbito de sistema: `PTHREAD_SCOPE_SYSTEM`
 - ámbito de proceso: `PTHREAD_SCOPE_PROCESS`
- Herencia de atributos de planificación (`inheritsched`); muy importante, ya que si hay herencia no se hace caso del resto de atributos de planificación; valores:
 - hereda los del padre: `PTHREAD_INHERIT_SCHED`
 - usa los del objeto `attr`: `PTHREAD_EXPLICIT_SCHED`

Planificación de threads (cont.)

Atributos de planificación (cont.)

- Política de planificación (`schedpolicy`); valores:
 - `SCHED_FIFO`
 - `SCHED_RR`
 - `SCHED_OTHER`
 - `SCHED_EDF` (de MaRTE OS)
- Parámetros de planificación(`schedparam`); es del tipo:

```
struct sched_param {  
    int sched_priority;  
    . . .  
}
```

Funciones para atributos de planificación

```
#include <pthread.h>

int pthread_attr_setscope
    (pthread_attr_t *attr,
     int contentionscope);

int pthread_attr_getscope
    (const pthread_attr_t *attr,
     int *contentionscope);

int pthread_attr_setinheritsched
    (pthread_attr_t *attr,
     int inheritsched);

int pthread_attr_getinheritsched
    (const pthread_attr_t *attr,
     int *inheritsched);
```

Funciones para atributos de planificación (cont.)

```
int pthread_attr_setschedpolicy  
    (pthread_attr_t *attr,  
     int policy);
```

```
int pthread_attr_getschedpolicy  
    (const pthread_attr_t *attr,  
     int *policy);
```

```
int pthread_attr_setschedparam  
    (pthread_attr_t *attr,  
     const struct sched_param *param);
```

```
int pthread_attr_getschedparam  
    (const pthread_attr_t *attr,  
     struct sched_param *param);
```

Funciones para el cambio dinámico de atributos de planificación

Para prioridades fijas FP

```
#include <pthread.h>

int pthread_getschedparam
    (pthread_t thread,
     int *policy, struct sched_param *param);

int pthread_setschedparam
    (pthread_t thread,
     int policy,
     const struct sched_param *param);
```

Ejemplo: planificación de threads FP

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sched.h>
#include <string.h>
#include "load.h"

// datos transferidos a cada thread periódico

struct periodic_params {
    float period, execution_time;
    int id;
};
```

Ejemplo (cont.)

```
// Thread pseudo-periódico
// Pone un mensaje en pantalla, consume tiempo de CPU,
// y pone otro mensaje
```

```
void * periodic (void *arg)
{
    struct periodic_params params;

    params = *(struct periodic_params*) arg;
    while (1) {
        printf("Thread %d starts\n", params.id);
        eat(params.execution_time);
        printf("Thread %d ends\n", params.id);
        sleep ((int)params.period);
    }
}
```

Ejemplo (cont.)

```
// Programa principal, que crea dos threads periódicos  
// con periodos y prioridades diferentes
```

```
int main()  
{  
    pthread_attr_t attr;  
    pthread_t t1,t2;  
    struct periodic_params t1_params, t2_params;  
    struct sched_param sch_param;  
    int err;  
  
    adjust();  
  
    // Crea objeto de atributos  
    if (pthread_attr_init (&attr) != 0) {  
        printf("Error en inicialización de atributos\n");  
        exit(1);  
    }  
}
```


Ejemplo (cont.)

```
// Coloca cada atributo

// El atributo inheritsched es muy importante
if (pthread_attr_setinheritsched (&attr,0) != 0) {
    printf("Error en atributo inheritsched\n");
    exit(1);
}
if (pthread_attr_setdetachstate
    (&attr,PTHREAD_CREATE_DETACHED) != 0) {
    printf("Error en atributo detachstate\n");
    exit(1);
}
if (pthread_attr_setschedpolicy (&attr, SCHED_FIFO) != 0) {
    printf("Error en atributo schedpolicy\n");
    exit(1);
}
sch_param.sched_priority =
    (sched_get_priority_max(SCHED_FIFO) - 5);
```

Ejemplo (cont.)

```
if (pthread_attr_setschedparam (&attr,&sch_param) != 0) {
    printf("Error en atributo schedparam\n");
    exit(1);
}

// Preparar el argumento del thread y crear el thread
t1_params.period = 2.0;
t1_params.execution_time = 1.0;
t1_params.id = 1;
if ((err=pthread_create(&t1,&attr,periodic,&t1_params)) != 0) {
    printf("Error en creacion del thread 1: %s\n",strerror(err));
    exit(1);
}
```

Ejemplo (cont.)

```
// Prepara atributos y parámetros para el segundo thread
sch_param.sched_priority =
    (sched_get_priority_max(SCHED_FIFO) - 6);
if (pthread_attr_setschedparam (&attr,&sch_param) != 0) {
    printf("Error en atributo schedparam\n");
    exit(1);
}
t2_params.period = 7.0;
t2_params.execution_time = 3.0;
t2_params.id = 2;
if ((err=pthread_create(&t2,&attr,periodic,&t2_params)) != 0) {
    printf("Error en creacion de thread 2: %s\n",strerror(err));
    exit(1);
}

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

Funciones para atributos de planificación en EDF

```
#include <pthread.h>

int pthread_attr_setreldeadline
    (pthread_attr_t *attr,
     const struct timespec *reldeadline);

int pthread_attr_getreldeadline
    (const pthread_attr_t *attr,
     struct timespec *reldeadline);

int pthread_attr_setpreemptionlevel
    (pthread_attr_t *attr,
     unsigned short int preemptionlevel);

int pthread_attr_getpreemptionlevel
    (const pthread_attr_t *attr,
     unsigned short int *preemptionlevel);
```

Funciones para cambio dinámico atributos de planificación en EDF

```
#include <pthread.h>

int pthread_setdeadline
(pthread_t thread,
 const struct timespec *deadline,
 clockid_t clock_id,
 int immediate);

int pthread_getdeadline
(pthread_t thread,
 clockid_t clock_id,
 struct timespec *deadline);
```

Es necesario establecer el plazo absoluto con el que se debe planificar el thread al despertarse justo antes de suspenderlo.

Mutexes en POSIX

Atributos de inicialización:

- *pshared*: indica si es compartido o no entre procesos:
 - PTHREAD_PROCESS_SHARED
 - PTHREAD_PROCESS_PRIVATE
- hay otros atributos relativos a la planificación, que veremos más adelante

Estos atributos se almacenan en un objeto de atributos

Objetos de atributos para mutex

```
#include <pthread.h>
```

```
int pthread_mutexattr_init  
    (pthread_mutexattr_t *attr);
```

```
int pthread_mutexattr_destroy  
    (pthread_mutexattr_t *attr);
```

```
int pthread_mutexattr_getpshared  
    (const pthread_mutexattr_t *attr,  
     int *pshared);
```

```
int pthread_mutexattr_setpshared  
    (pthread_mutexattr_t *attr,  
     int pshared);
```

Inicializar y destruir un mutex

Inicializar un Mutex:

```
int pthread_mutex_init (pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *attr);
```

Destruir un mutex:

```
int pthread_mutex_destroy (  
    pthread_mutex_t *mutex);
```


Tomar y liberar un mutex

Tomar un mutex y suspenderse si no está libre:

```
int pthread_mutex_lock (  
    pthread_mutex_t *mutex);
```

Tomar un mutex, sin suspenderse:

```
int pthread_mutex_trylock (  
    pthread_mutex_t *mutex);
```

Liberar un mutex

```
int pthread_mutex_unlock (  
    pthread_mutex_t *mutex);
```

Ejemplo: come-tiempos

```
////////////////////////////////////  
///                                LOAD.H                                ///  
////////////////////////////////////  
//      La función eat() permite consumir el tiempo      ///  
//      de CPU indicado por for_seconds                    ///  
  
//      La función adjust() debe ser llamada una vez      ///  
//      antes de llamar a eat(). No es necesario          ///  
//      llamarla posteriormente.                           ///  

```

```
void eat (float for_seconds);
```

```
void adjust (void);
```

Ejemplo con un mutex

```
#include <pthread.h>
#include <stdio.h>
#include "load.h"
struct shared_data {
    pthread_mutex_t mutex;
    int a,b,c,i;
} data;
// Este thread incrementa a, b y c 20 veces
void * incrementer (void *arg)
{
    for (data.i=1;data.i<20;data.i++) {
        pthread_mutex_lock(&data.mutex);
        data.a++; eat(0.1);
        data.b++; eat(0.1);
        data.c++; eat(0.1);
        pthread_mutex_unlock(&data.mutex);
    }
    pthread_exit (NULL);
}
```

Ejemplo con un mutex (cont.)

```
// Este thread muestra el valor de a, b y c
// hasta que i vale 20
void * reporter (void *arg)
{
    do {
        pthread_mutex_lock(&data.mutex);
        printf("report a=%d, b=%d, c=%d\n", data.a, data.b, data.c);
        pthread_mutex_unlock(&data.mutex);
        eat(0.2);
    } while (data.i < 20);
    pthread_exit (NULL);
}

// Programa principal: crea el mutex y los threads
// Luego, espera a que los threads acaben
int main()
{
    pthread_mutexattr_t mutexattr;
    pthread_t t1, t2;
```

Ejemplo con un mutex (cont.)

```
adjust();
data.a = 0; data.b = 0; data.c = 0; data.i = 0;
pthread_mutexattr_init(&mutexattr);
if (pthread_mutex_init(&data.mutex,&mutexattr) != 0) {
    printf ("mutex_init\n"); exit(1);
}

if (pthread_create (&t1,NULL,incrementer,NULL) != 0) {
    printf ("pthread_create\n");
    exit(1);
}
if (pthread_create (&t2,NULL,reporter,NULL) != 0) {
    printf ("pthread_create\n");
    exit(1);
}
if (pthread_join(t1,NULL) != 0) printf ("in join\n");
if (pthread_join(t2,NULL) != 0) printf ("in join\n");
exit (0);
}
```

Sincronización de espera mediante semáforos

Semáforo Contador:

- es un recurso compartible con un valor entero no negativo
- cuando el valor es cero, el semáforo no está disponible
- se utiliza tanto para sincronización de acceso mutuamente exclusivo, como de espera
- operaciones que se aplican al semáforo:
 - *esperar*: si el valor es 0, el proceso o thread se añade a una cola; si es >0 , se decrementa
 - *señalizar*: si hay procesos o threads esperando, se elimina uno de la cola y se le activa; si no, se incrementa el valor
- existen semáforos con nombre, y sin nombre

Semáforos contadores

Inicializar un semáforo sin nombre:

```
#include <semaphore.h>
int sem_init (sem_t *sem, int pshared,
               unsigned int value);
```

- inicializa el semáforo al que apunta **sem**
- si **pshared** = 0 el semáforo sólo puede ser usado por threads del mismo proceso
- si **pshared** no es 0, el semáforo se puede compartir entre diferentes procesos
- **value** es el valor inicial del semáforo

Semáforos contadores (cont.)

Destruir un semáforo sin nombre:

```
int sem_destroy (sem_t *sem);
```

Abrir/inicializar un semáforo con nombre:

```
sem_t *sem_open (const char *name, int oflag,  
                [,mode_t mode, unsigned int value]);
```

- devuelve un puntero al semáforo
- **name** debe tener el formato **"/nombre"**
- **oflag** indica las opciones:
 - **O_CREAT**: si el semáforo no existe, se crea; en este caso se requieren los parámetros **mode**, que indica permisos, y **value**, que es el valor inicial
 - **O_EXCL**: si el semáforo ya existe, error

Semáforos contadores (cont.)

Cerrar un semáforo con nombre:

```
int sem_close (sem_t *sem);
```

Borrar un semáforo con nombre:

```
int sem_unlink (const char *name);
```

Esperar en un semáforo:

```
int sem_wait (sem_t *sem);
```

- decrementa el semáforo si está libre; si no, se suspende hasta que se libera, o se ejecuta un manejador de señal

```
int sem_trywait (sem_t *sem);
```

- decrementa el semáforo si está libre; si no, retorna indicando un error

Semáforos contadores (cont.)

Señalizar un semáforo:

```
int sem_post (sem_t *sem);
```

Leer el valor de un semáforo:

```
int sem_getvalue (sem_t *sem, int *sval);
```

Ejemplo: espera entre procesos

```
#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include "load.h"
int main()
{
    sem_t *sem; pid_t childpid; int i;
    adjust();
    if ((sem = sem_open ("/my_sem", O_CREAT, S_IRUSR|S_IWUSR, 0))
        == SEM_FAILED) {
        // valor inicial =0 para sincronización de espera
        perror("error in sem_open\n");
    }
    if ((childpid = fork()) == -1) {
        perror("fork failed\n");
    }
}
```

Ejemplo de espera (cont.)

```
// El proceso hijo señala el semáforo 20 veces
if (childpid == 0) {
    for (i=0;i<20;i++) {
        eat(1.0);
        printf("child process writes i=%d\n",i);
        sem_post(sem);
    }
    sem_close(sem); exit(0);
} else {
// El proceso padre espera en el semáforo 20 veces
    for (i=0;i<20;i++) {
        eat(0.5);
        sem_wait(sem);
        printf("parent process writes i=%d\n",i);
    }
    sem_close(sem);
    sem_unlink("/my_sem"); exit(0);
}
}
```

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);
```

Relojes y temporizadores en 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 = $\text{sec} + \text{min} * 60 + \text{hour} * 3600 + \text{yday} * 86400 + (\text{year} - 70) * 31536000 + ((\text{year} - 69) / 4) * 86400$

Relojes y temporizadores en 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 en 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*109+tv_nsec  
- 0≤tv_nsec<109
```

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`

El valor máximo de la 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);
```

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

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

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);
}
```