POSIX-RT

Librería

Sistemas Informáticos de Tiempo Real
Francisco Pastor Gomis

1. Estándares POSIX

POSIX es el acrónimo de Interfaz de Sistemas Operativos Portables (Portable Operating Systems Interfaces). Es un estándar basado en el popular Sistema Operativo UNIX. Aunque el UNIX era un estándar industrial *de facto*, había suficientes diferencias entre las diferentes implementaciones de UNIX como para impulsar a los implementadores y usuarios a patrocinar la creación de un estándar internacional formal con el propósito de conseguir la portabilidad de las aplicaciones a nivel de código fuente. El POSIX está siendo desarrollado en el marco de la *Computer Society* de IEEE, con la referencia IEEE 1003, y también está siendo desarrollado a nivel de estándar internacional con la referencia ISO/IEC 9945.

El POSIX es una familia de estándares en evolución, cada uno de los cuales cubre diferentes aspectos de los sistemas operativos. Alguno de los estándares POSIX ya han sido aprobados, mientras que otros están siendo desarrollados todavía. Los podemos agrupar en tres categorías:

1. *Estándares base*: Definen la sintaxis y semántica de interfaces de servicios relacionados con diversos aspectos del sistema operativo. El estándar no especifica como se implementan estos servicios, sino únicamente su semántica. La mayoría de los estándares base están especificados para el lenguaje de programación C. La siguiente tabla lista algunos de los estándares básicos POSIX que son los más importantes para las aplicaciones de tiempo real.

1003.1	Servicios básicos del Sistema Operativo
1003.1a	Extensiones a los servicios básicos
1003.1b	Extensiones de tiempo real
1003.1c	Extensiones de threads
1003.1d	Extensiones adicionales de tiempo real
1003.1e	Seguridad
1003.1f	Sistema de ficheros en red (NFS)
1003.1g	Comunicaciones por red
1003.1h	Tolerancia a fallos
1003.1j	Extensiones de tiempo real avanzadas
1003.1m	Puntos de chequeo y reintento
1003.2	Shell y utilidades
1003.2b	Utilidades adicionales
1003.2d	Ejecución por lotes (batch)
1003.3	Métodos para probar la conformidad con POSIX
1003.21	Comunicaciones para sistemas distribuidos de tiempo real

2. *Interfaces en diferentes lenguajes de programación ("bindings")*: estos estándares proporcionan interfaces a los mismos servicios definidos en los estándares base, pero usando otros lenguajes de programación. Los lenguajes que se han usado hasta el momento son el Ada y el Fortran. En la siguiente tabla se muestran algunos de ellos:

1003.5	*Binding* de Ada para 1003.1
1003.5b	*Binding* de Ada para 1003.1b y 1003.1c
1003.5c	*Binding* de Ada para 1003.1g
1003.5f	*Binding* de Ada para 1003.21
1003.9	*Binding* de Fortran 77 para 1003.1

3. *Entornos de sistemas Abiertos*: Estos estándares incluyen una guía al entorno POSIX y perfiles de aplicación. Los perfiles son un subconjunto de los servicios POSIX que se requieren

para un determinado ámbito de aplicación. Representan un importante mecanismo para definir de forma estándar un conjunto bien definido de implementaciones de sistemas operativos, adecuadas para áreas de aplicación específica.

2. Estándares POSIX para tiempo real

Los estándares POSIX ya aprobados que son de interés para tiempo real son el 1003.1, 1003.1b y 1003.1c.

2.1 El estándar básico 1003.1

Define los servicios más básicos de un sistema operativo UNIX convencional, que se centran en dos objetos fundamentales: los *procesos*, que proporcionan la ejecución concurrente de programas en espacios de direcciones independientes; y los *ficheros*, que son objetos que representan distintas facilidades del sistema operativo (por ejemplo, datos, dispositivos de E/S, etc.) sobre las que se pueden realizar operaciones de lectura o escritura.

Los servicios definidos en el POSIX 1003.1 incluyen:

- la gestión de procesos
- identificación y entorno de procesos
- notificación de eventos a través de señales
- algunos servicios de temporización muy primitivos
- servicios de ficheros y directorios
- entrada/salida
- control de terminales
- bases de datos para usuarios y grupos.

2.2 Las extensiones para tiempo real 1003.1b

El estándar define las extensiones para tiempo real que se consideran esenciales para dar soporte a aplicaciones con requisitos de tiempo real. Los servicios definidos en este estándar se pueden agrupar en dos categorías:

Servicios que facilitan la programación concurrente

Son necesarios porque la mayoría de las aplicaciones de tiempo real son concurrentes, y sus procesos cooperan estrechamente. Algunos de los servicios definidos son:

- sincronización de procesos mediante semáforos contadores
- objetos de memoria compartida: permite a los procesos con espacios de direcciones independientes compartir información
- colas de mensajes: permiten el intercambio de eventos o datos entre procesos

- entrada/salida asíncrona: permite a la aplicación ejecutarse en paralelo con las operaciones de entrada/salida
- entrada/salida sincronizada: permite un mayor grado de predicibilidad en las operaciones de entrada/salida sobre ficheros

Servicios que se necesitan para conseguir un comportamiento temporal predecible

Los servicios que se incorporan son:

- planificación expulsora de procesos mediante políticas basadas en prioridades fijas
- la inhibición de memoria virtual para el espacio de direcciones de un proceso, para poder conseguir un tiempo de acceso a memoria predecible
- señales de tiempo real, que proporcionan un comportamiento más predecible que las señales
- relojes y temporizadores que permiten la gestión del tiempo desde la aplicación.

2.3 La extensión de threads 1003.1c

Las unidades de concurrencia del POSIX 1003.1 son los procesos con espacios de direcciones independientes. En la práctica los procesos son relativamente voluminosos y poco eficaces debido al requisito de espacios de direcciones independientes y el estado asociado al procesos, que es voluminoso, haciendo muy costosa la realización de un cambio de contexto.

Por el contrario, las tareas de un núcleo de tiempo real de alta eficacia comparten el mismo espacio de direcciones y tienen un estado asociado pequeño.

Con el propósito de incrementar la eficiencia, se introdujeron en el POSIX los threads, o procesos ligeros, como un mecanismo de concurrencia de alta eficacia.

Bajo la Extensión de Threads, cada proceso POSIX puede tener múltiples flujos de control concurrentes, todos ellos compartiendo el mismo espacio de direcciones. Los servicios que se proporcionan para los threads incluyen:

- creación
- cancelación
- planificación
- sincronización

2.4 Otras extensiónes para tiempo real

Además de los estándares anteriores, existen otras extensiones de tiempo real que están siendo desarrolladas bajo los estándares P1003.1d y P1003.1j.

Los servicios definidos en estos estándares no se incluyeron en las primeras extensiones de tiempo real porque, aunque son importantes para muchas aplicaciones de tiempo real, no se consideraron esenciales.

Los servicios definidos en P1003.1d incluyen:

- el arranque rápido de procesos
- tiempos límite en servicios bloqueantes
- medida y limitación de tiempos de CPU
- planificación del servidor esporádico
- información para la implementación de ficheros de tiempo real.

Los servicios definidos en P1003.1j incluyen:

- primitivas para sincronización en multiprocesadores
- gestión de memoria de diversos tipos
- nuevos relojes monotónico y sincronizado

3. Subconjuntos POSIX de tiempo real

Los servicios del POSIX básico junto a las Extensiones de Tiempo Real y la Extensión de Threads contribuyen a formar un sistema operativo muy grande, que casi con toda probabilidad no puede ser implementado en computadores empotrados pequeños.

Para que el POSIX sea aplicable a este y otros entornos restringidos, se han definido cuatro subconjuntos o perfiles estándar del POSIX de tiempo real, que se orientan a cuatro plataformas comunes que han tenido éxito comercial.

La principal diferencia de estos perfiles radica en la presencia o no de múltiplos procesos y la presencia o no de un sistema de ficheros jerárquicos. La siguiente tabla muestra las características de los cuatro perfiles de tiempo real, de acuerdo con estas propiedades.

Perfil	Nombre de perfil	Sistema de ficheros	Múltiples procesos	Múltiples threads	Plataforma típica
PSE51	Sistema de tiempo real mínimo	No	No	Si	Sistemas empotrado pequeño, sin MMU, sin disco, sin terminal.
PSE52	Controlador de tiempo real	Si	No	Si	Controlador industrial de propósito especial, sin MMU, pero con un disco o disquete y un terminal
PSE53	Sistema de tiempo real dedicado	No	Si	Si	Sistema empotrado grande, con MMU, pero sin disco
PSE54	Sistema de tiempo real multipropósito	Si	Si	Si	Computador grande con requisitos de tiempo real

Los servicios específicos definidos en el perfil mínimo de tiempo real, y por tanto pensados para sistemas empotrados pequeños, se muestran en la siguiente tabla:

Unidades de funcionalidad del POSIX 1003.1	Opciones del POSIX 1003.1b	Opciones del POSIX 1003.1c
Proceso único Señales I/O de dispositivos Soporte de lenguaje C	Inhibición de memoria virtual Idem para rangos de direcciones Semáforos Objetos de memoria compartida Señales de tiempo real Temporizadores Colas de mensjaes I/O sincronizada	Threads Atributo de tamaño de *stack* de thread Atributo de dirección del *stack* Planificación de threads por prioridad Protocolo de herencia de prioridad Protocolo de protección de prioridad

4. Servicios para un sistema mínimo

Vamos a ver cuales son los servicios más importantes que se incorporan en el perfil del sistema mínimo.

Estos servicios se centran en la concurrencia y en los servicios necesarios para su utilización (sincronización, comunicación, etc.). Se hace especial hincapié en la predicibilidad de los servicios y en los tiempos de ejecución de las tareas.

4.1 Concurrencia

La concurrencia se consigue mediante el uso de los threads. Un thread representa un flujo de control simple que se ejecuta concurrentemente a otros threads en el sistema.

Cada thread tiene su propio identificador y una serie de recursos, mucho más reducidos de los que posee un proceso, y que son:

- Una pila (stack)
- Una política y parámetros de planificación
- Datos específicos del thread: contexto, mascara de señales, lista de mutex, etc.

Ya que todos los threads de un proceso comparten el mismo espacio de direcciones, pueden ser construidos con un grado de eficiencia superior al de los procesos. Los procesos requieren la separación de espacios de direcciones y, por tanto, para realizar un cambio de procesos se debe hacer uso de la MMU.

En el perfil mínimo sólo hay un proceso, con múltiples threads. El sistema comienza por crear el thread principal, que ejecuta la función C `main()`. Este thread creará posteriormente al resto de los threads de la aplicación de forma dinámica.

Los principales servicios que proporciona el POSIX para la gestión de los threads son:

- Manipulación de atributos de thread
- Creación de threads
- Terminación y cancelación de threads

4.1.1 Manipulación de atributos de thread

Cada thread tiene un conjunto de atributos a los que normalmente se le asigna un valor cuando se crea un thread (aunque hay algunos atributos que pueden cambiar dinámicamente).

Los atributos que se utilizan para crear el thread se hallan almacenados en un objeto "opaco" del tipo `pthread_attr_t`. Hay servicios para inicializar o destruir uno de estos objetos, y una pareja de funciones asociada a cada atributo, una para poner el valor del atributo y otra para leerlo.

Las funciones para inicializar y destruir atributos son:

```
int pthread_attr_init(pthread_attr_t* __attr);
int pthread_attr_destroy(pthread_attr_t* __attr);
```

Los principales atributos definidos son el tamaño mínimo de la pila y el estado de devolución de recursos. Este último atributo define si el thread se crea en el estado independiente o *detached* (`PTHREAD_CREATE_DETACHED`), o en el estado sincronizado o *joinable* (`PTHREAD_CREATE_JOINABLE`); un thread independiente devuelve al sistema los recursos que usa cuando se termina; un thread sincronizado mantiene sus recursos cuando termina, y sólo los devuelve cuando otro thread ejecuta la función `pthread_join()`.

Las funciones relacionadas con estos atributos son:

```
int pthread_attr_setstacksize    (pthread_attr_t* __attr, size_t __stacksize);
int pthread_attr_getstacksize    (pthread_attr_t* __attr, size_t* __stacksize);
int pthread_attr_setdetachstate  (pthread_attr_t* __attr, int __detachstate);
int pthread_attr_getdetachstate  (pthread_attr_t* __attr);
```

Si los valores de los atributos no se inicializan el sistema considera unos valores por defecto.

4.1.2 Creación de threads

Una vez que el objeto de atributos del thread ha sido creado y que cada uno de los atributos individuales ha sido inicializado al valor deseado, podemos crear el thread mediante la función `pthread_create()`.

En esta llamada se debe indicar, además de los objetos thread y atributos, una función C que será ejecutada por el nuevo thread, y el parámetro con el que esta función será llamada, del tipo `void*`. Este parámetro se puede usar para pasar información al nuevo thread.

```
int pthread_create      (pthread_t* __thread,
                         pthread_attr_t* __attr,
                         void* (*__func)(void*),
                         void* __arg);
```

Si el tipo de la función o del parámetro no coinciden con los especificados en la llamada anterior, se puede realizar la correspondiente conversión de tipos para evitar los errores del compilador.

Otras funciones relacionadas con un thread ya creado son:

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

Devuelve el objeto asociado al thread que llama a la función

```
int pthread_detach( pthread_t  handle );
```

Convierte al thread que se hace referencia en dettached.

4.1.3 Terminación y cancelación de threads

Un thread puede terminarse a sí mismo llamando a la función `pthread_exit()`, o si alcanza el final lógico que el thread comenzó a ejecutar cuando se creó.

También es posible terminar un thread llamando a `pthread_cancel()`, y especificando el objeto asociado al thread que se quiere terminar.

En ambos casos, el thread ejecuta las operaciones de terminación que hayan sido registradas mediante llamadas a `pthread_cleanup_push()`.

Las funciones relacionadas con la creación y cancelación de threads son:

```
int pthread_join(pthread_t handle, void **return_value);
int pthread_cancel(pthread_t __thread);
int pthread_exit(void *__exit_value);
int pthread_cleanup_push(void (*__routine)(void *), void *__arg);
int pthread_cleanup_pop( int execute );
```

4.2 Planificación

La teoría de planificación proporciona diversas soluciones para conseguir un comportamiento temporal predecible junto a un alto nivel de utilización de la CPU.

Una de las soluciones más simples y mejor conocidas es la planificación expulsora con prioridades fija. En esta política cada thread tiene una prioridad asignada, y el planificador elige para ejecución aquel thread que tiene mayor prioridad, de entre los que están listos para ejecutar.

El POSIX especifica la planificación expulsora con prioridades fijas para los threads y también para los procesos, con dos variaciones que pueden ser seleccionadas a nivel de cada thread, como dos políticas de planificación diferentes.:

`SCHED_FIFO`: Este es el nombre de la política de planificación expulsora con prioridades que usa orden FIFO para determinar el orden en el que se ejecutan los threads de la misma prioridad.

`SCHED_RR`: Este es el nombre de la política de planificación expulsora con prioridades que usa un esquema cíclico para planificar threads de la misma prioridad.

Hay que tener en cuenta que las políticas de planificación definidas sólo tienen un comportamiento temporal predecible en monoprocesadores, o en sistemas multiprocesadores en los que la asignación de threads a procesadores sea estática.

Existe una tercera política de planificación (`SCHED_OTHER`) con un comportamiento definido por la implementación, que se definió para permitir la compatibilidad con implementaciones preexistentes.

Puede parecer extraño que la política de planificación sea un atributo de cada thread, y no un atributo global del sistema. Ello se debe a que las dos políticas de planificación son compatibles entre sí y,

de hecho, son idénticas excepto para el tratamiento de los threads de la misma prioridad. En cualquier caso, la política recomendada para sistemas de tiempo real estricto es `SHCED_FIFO`.

El estándar requiere como mínimo 32 niveles distintos de prioridad, que es un número suficiente para conseguir altos niveles de utilización incluso para números muy elevados de threads.

Los atributos de política de planificación y prioridad forman parte del objeto de atributos del thread que se usará posteriormente para crear un thread. Asimismo, es posible modificar o consultar estos atributos dinámicamente, después de que el thread haya sido creado, usando las funciones:

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

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

Los atributos de planificación definidos en el estándar y que son útiles para aplicaciones de un solo proceso son los siguientes:

- Política de planificación (*schedpolicy*): para tiempo real debemos elegir `SCHED_FIFO` o `SCHED_RR`.
- Parámetros de planificación (*schedparam*): esta es una estructura extensible del tipo , `struct sched_param` que de momento sólo tiene un campo definido, `sched_priority`, que representa la prioridad del thread.
- Herencia de atributos de planificación (*inheritsched*): este atributo define si los atributos de planificación usados para crear el thread se heredan del thread padre (`PTHREAD_INHERIT_SCHED`) o son definidos en el objeto de atributos de thread (`PTHREAD_EXPLICIT_SCHED`). Es extremadamente importante colocar este atributo al valor `PTHREAD_EXPLICIT_SCHED` si deseamos que el nuevo thread tenga una política de planificación o una prioridad distinta de la del thread padre.

Las funciones relacionadas con estos atributos de thread son:

```
int pthread_attr_setschedpolicy(pthread_attr_t *__handle, int __sched_policy);
int pthread_attr_getschedpolicy(pthread_attr_t *__handle, int *__sched_policy);
int pthread_attr_setinheritsched(pthread_attr_t *__handle, int __inherit_sched);
int pthread_attr_getinheritsched(pthread_attr_t *__handle, int *__inherit_sched);
int pthread_attr_setschedparam(pthread_attr_t *handle,
                               const struct sched_param *sched_param );
int pthread_attr_getschedparam(pthread_attr_t *handle,
                               struct sched_param *sched_param );
```

4.3 Sincronización

El estándar 1003.1c proporciona dos primitivas de sincronización par threads: el mutex para el acceso mutuamente exclusivo a recursos compartidos, y las variables condicionales para la sincronización de espera.

Ambas primitivas se utilizan a través de objetos `pthread_mutex_t` y `pthread_cond_t` respectivamente. Cada uno de estos objetos tiene un objeto de atributos asociado que se usa al crear el objeto, en una forma similar a la de los atributos de creación de threads.

El mutex es el objeto de sincronización usado para la exclusión mutua. Tiene dos operaciones definidas:

- *Bloquear*: si el mutex está libre, se bloque y el thread que invoca la operación se convierte en el propietario del mutex; en caso contrario, el thread se suspende y se añade a la cola de prioridad de los threads que están esperando en ese mutex

- *Liberar*: si hay threads esperando en la cola, se activa al que esté en la primera posición, y se convierte en el nuevo propietario del mutex; en caso contrario, el mutex queda libre; sólo el propietario del mutex está autorizado a invocar esta operación

El perfil de sistema de tiempo real mínimo requiere que se de soporte tanto al protocolo de herencia de prioridad como al de protección de prioridad, para los mutex. Ambos protocolos evitan el efecto conocido como inversión de prioridad no acotada, que es la causa de retrasos de muy larga duración en el tiempo de respuesta de threads de alta prioridad.

El uso de uno de estos protocolos es, por tanto, un requisito imprescindible en sistemas de tiempo estricto.

El protocolo que muestra mejor tiempo de respuesta en el peor caso es el de protección de prioridad (también llamado de techo de prioridad), mientras que el mejor tiempo de respuesta promedio corresponde al protocolo de herencia de prioridad.

Los atributos de creación de mutex que son útiles en aplicaciones de un solo proceso son:

- *protocolo (protocol)*: los valores permitidos son `PTHREAD_NONE` para sistemas sin requisitos de tiempo real, y `PTHREAD_PRIO_INHERITANCE` y `PTHREAD_PRIO_PROTECT`, para los protocolos de herencia de prioridad y protección de prioridad, respectivamente.

- *techo de prioridad (priority ceiling)*: es el valor de prioridad heredado por el propietario del mutex si se elige el protocolo de protección de prioridad.

Las variables condicionales son objetos de sincronización que se usan para que un thread pueda suspenderse en espera de que otro thread lo reactive. Las operaciones asociadas a las variables condicionales son:

- *esperar*: el thread se suspende hasta que otro thread señaliza la variable condicional.

- *señalizar*: se reactiva sólo uno de los threads que están suspendidos a la espera de la variable condicional; no tiene efecto si no hay ningún thread esperando en la variable condicional.

- *broadcast*: todos los threads suspendidos en la variable condicional se reactivan.

La variable condicional se usa siempre conjuntamente a un mutex que se suele usar para proteger la evaluación de un predicado lógico que determina si el thread debe esperar o no.

La operación de espera se invoca con el mutex tomado por el thread que invoca la operación, y mientras este thread permanece suspendido el mutex se libera automáticamente, para dar la oportunidad al thread que señaliza de modificar las variables protegidas, y por tanto afectar al resultado del predicado lógico.

Una vez que la variable condición se señaliza, el thread que espera toma el mutex de nuevo, de forma automática y antes de que la operación de espera termine. Esto permite al thread que estaba esperando reevaluar el predicado lógico mientras mantiene tomado el mutex que protege su estado.

Las funciones relacionadas con las operaciones sobre mutex y variables condición son:

```
int pthread_mutexattr_init(pthread_mutexattr_t *mu_attr_h);
int pthread_mutexattr_destroy(pthread_mutexattr_t *mu_attr_h);
int pthread_mutexattr_setprotocol     (pthread_mutexattr_t *handle,
                                       pthread_protocol_t protocol);
int pthread_mutexattr_getprotocol     ( pthread_mutexattr_t *handle,
                                       pthread_protocol_t *protocol);
int pthread_mutexattr_setprio_ceiling ( pthread_mutexattr_t *handle,
                                       int prio_ceiling);
int pthread_mutexattr_getprio_ceiling (pthread_mutexattr_t *handle,
                                       int *prio_ceiling);

int pthread_mutex_init(pthread_mutex_t *handle, pthread_mutexattr_t *mu_attr_h);
int pthread_mutex_destroy(pthread_mutex_t *handle );
int pthread_mutex_lock(pthread_mutex_t *handle );
int pthread_mutex_unlock(pthread_mutex_t *handle );
int pthread_mutex_trylock(pthread_mutex_t *mu_h );

int pthread_condattr_init( pthread_condattr_t *handle);
int pthread_condattr_destroy( pthread_condattr_t *handle);

int pthread_cond_init(pthread_cond_t *handle, pthread_condattr_t *cv_attr_h);
int pthread_cond_destroy(pthread_cond_t *handle );
int pthread_cond_wait(pthread_cond_t *cv_h,  pthread_mutex_t *mu_h );
int pthread_cond_signal(pthread_cond_t *handle );
int pthread_cond_broadcast(pthread_cond_t *handle );
int pthread_cond_timedwait            (pthread_cond_t *cv_h,
                                       pthread_mutex_t *mu_h,
                                       const struct timespec *abstime );
```

4.4 Temporización

Un requisito general en sistemas de tiempo real es el poder medir el tiempo y ejecutar acciones específicas periódicamente, o en un instante de tiempo concreto, o cuando ha transcurrido un cierto intervalo de tiempo. Estos requisitos se pueden cubrir usando los servicios de relojes y temporizadores del POSIX 1003.1b, que se describen a continuación.

- *Relojes*: El POSIX define un reloj de ámbito de sistema, identificado por el símbolo CLOCK_REALTIME. Aunque la resolución de este reloj puede ser tan fina como un nanosegundo, el estándar sólo impone a la implementación una resolución de al menos 20 mseg. Se definen funciones para consultar o modificar la hora tanto del reloj CLOCK_REALTIME

como de cualquier otro reloj definido por la implementación que esté disponible (*clock_gettime*(), *clock_settime*()).

- *Retraso de alta resolución*: La función *nanosleep*() proporciona a un thread la capacidad de suspenderse a sí mismo hasta que transcurre un determinado intervalo temporal, medido según el reloj CLOCK_REALTIME. Aunque la función *nanosleep*() es útil para crear acciones temporizadas de forma relativa, no es útil para crear acciones periódicas.

- *Temporizadores*: Son objetos capaces de notificar a la aplicación del paso de un intervalo de tiempo o del momento en el que un reloj alcanza una determinada hora. La notificación que ocurre cada vez que el temporizador expira consiste en enviar al proceso una señal especificada por el usuario, o crear y ejecutar un thread. El instante de la primera expiración se puede especificar como un intervalo relativo o una hora absoluta; además, el temporizador se puede programar para expirar periódicamente después de la primera expiración.

En un sistema mínimo sólo existe un único reloj, el CLOCK_REALTIME.

Algunas funciones relacionadas con los temporizadores son:

```
struct timespec
{
    long tv_sec;
    long tv_nsec;
};

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
int clock_settime(int clock_id, const struct timespec *tp);
int clock_gettime(int clock_id, struct timespec *tp);

struct sigevent
{
    int sigev_notify;
    int sigev_signo;
};

struct itimerspec
{
    struct timespec it_interval;
    struct timespec it_value;
};

int timer_create(int clock_id, struct sigevent* event, timert_t* timer_id);
int timer_settime(timer_t timer_id, int flags, const struct itimerspec* spec,
                  struct itimerspec *oldspec);
int timer_gettime(timer_t timer_id, struct itimerspec *spec);
```

En la implementación de threads en Linux, en lugar de la función `nanosleep` se ha implementado la función no portable:

```
int pthread_delay_np( const struct timespec *interval );
```

4.5 Señales

Las señales son el mecanismo utilizado en el POSIX para notificar a un proceso o a un thread de que ha ocurrido un evento, como por ejemplo una expiración de un temporizador, un fallo hardware, o un evento generado por la aplicación.

Cada thread tiene su propia máscara de señales que permite al thread controlar la forma y momento en el que el sistema operativo le puede entregar una señal.

En el sistema operativo mínimo, las señales se pueden generar en tres lugares:

- Interrupciones del temporizador
- Interrupciones de entrada/salida
- Interrupciones del ejecutivo

Una señal se puede generar para un thread específico, o para un proceso, en cuyo caso cualquier thread que exprese interés en la señal (bien desenmascarando la señal o invocando a una operación para `sigwait()` esperarla) la puede recibir. Como el sistema mínimo tiene un solo proceso, todos los threads del sistema son candidatos a recibir señales enviadas a ese único proceso.

Algunas funciones relacionadas con las señales son:

```
int sigwait( const sigset_t *sigset, int *signal );
int pthread_sigmask( int how, const sigset_t *newmask, sigset_t *prev );
int pthread_kill( pthread_t handle, int sig );
```

4.6 Funciones no reentrantes

En la utilización de threads de POSIX hay que prestar atención a la utilización de funciones de la librería estándar de C y de variables globales que no sean reentrantes o que no estén protegidas (por ejemplo las funciones de memoria dinámica y la variable errno).

En algunas implementaciones se incluyen las versiones reentrantes de las funciones más utilizadas, como pueden ser las funciones de memoria dinámica.

Con macros de C (#define) se puede sustituir la llamada a las funciones normales por las reentrantes. En este caso, hay que prestar especial atención en no utilizar ninguna función de la librería (ya compilada) que haga uso de alguna función no reentrante.

5. Índice