Tostadores y POSIX

Michael González Harbour[1] (mgh@ctr.unican.es)
Departamento de Electrónica y Computadores, Universidad de Cantabria
39005 Santander, SPAIN

C. Douglass Locke (doug.locke@lmco.com)
Lockheed Martin Corporation
700 No. Frederick Av., Gaithersburg MD 20879, USA

Resumen

Contrariamente a la impresión general acerca del estándar POSIX de IEEE, el subconjunto más pequeño que se ha definido para las interfaces POSIX, llamado el Perfil de Sistema de Tiempo Real Mínimo, define interfaces para un sistema operativo de muy reducidas dimensiones y alta eficiencia. Un sistema operativo de este tipo, totalmente conforme al perfil estándar, se puede implementar de una forma suficientemente pequeña y eficiente como para ser usado en aplicaciones tales como control de electrodomésticos, control de máquinas industriales, controladores de redes, y otras aplicaciones empotradas pequeñas que no tienen hardware para gestión de memoria, ni terminales, ni dispositivos de almacenamiento magnético.

1. Introducción

Aunque es bien conocido que el POSIX, el estándar de Interfaz de Sistemas Operativos Portables de IEEE, está basado en las interfaces UNIX, es mucho menos sabido que el estándar POSIX en realidad describe un conjunto de interfaces de aplicación aplicables a una gran variedad de implementaciones de sistemas operativos. Debido a la bien merecida reputación del UNIX de ser un sistema operativo repleto de funciones con servicios tales como un mecanismos de protección bien desarrollados, sistema de ficheros jerárquico, gestión de procesos, y un cierto conjunto de funciones de seguridad, se suele suponer que un sistema conforme a POSIX debe ser demasiado voluminoso para ser usado como una plataforma para sistemas de tiempo real. Ciertamente, cuando se formó el Grupo de Trabajo de Tiempo Real en POSIX, hubo mucha discusión sobre si el estándar de tiempo real no sería más que una figura retórica.

Desde aquel entonces, han sucedido diversos acontecimientos en el esfuerzo de estandarización. El primer lugar, el Grupo de Trabajo de Tiempo Real de POSIX completó con éxito las Extensiones de tiempo Real (1003.1b) que definían un número significativo de extensiones del sistema operativo a nivel de proceso, que otorgaban a la implementación y a la aplicación la habilidad de presentar tiempos de respuesta acotados. En segundo lugar, el Grupo de Trabajo también completó la Extensión de Threads (1003.1c) que definía un

[1] La participación de Michael González en el estándar POSIX está financiada en parte por la *Comisión Interministerial de Ciencia y Tecnología* del Gobierno Español

conjunto de interfaces para crear múltiples threads (o procesos ligeros concurrentes) dentro de cada proceso POSIX. En tercer lugar, el grupo de trabajo está a punto de terminar el proceso de votación de cuatro Perfiles de Entornos de Aplicaciones de Tiempo Real (1003.13), que definen cuatro subconjuntos del POSIX necesarios para cuatro categorías distintas de aplicaciones, con diferentes características de tamaño y complejidad.

Durante las deliberaciones del grupo de trabajo, se le puso al más pequeño de estos perfiles, llamado el Perfil del Sistema de Tiempo Real Mínimo (1003.13-PSE51) el nombre informal del perfil del "tostador", debido a que se pensó para que pudiera dar soporte a implementaciones tan pequeñas que pudiesen ser usadas incluso para implementar un controlador de un electrodoméstico para tostar pan. Los sistemas que se ajustan a este perfil normalmente no tienen un terminal para interfaz con el usuario, ni sistema de ficheros o dispositivos de almacenamiento magnético, y tienen un procesador y capacidad de memoria muy reducidos. Normalmente no tienen hardware para gestión de memoria (MMU) o, si lo tuviesen, permanecería sin usar e inhabilitado. Las aplicaciones típicas (probablemente no tostadores de pan) incluirían el control de electrodomésticos (como microondas o televisores), terminales de puntos de venta, controladores de motores de automóviles, controladores de sensores industriales, y otras aplicaciones que no requiriesen el soporte complejo ofrecido por los perfiles más grandes.

Aún así, algunos implementadores de este tipo de sistemas pequeños tienen pocas expectativas de que un sistema operativo conforme a POSIX pueda ser suficientemente pequeño o ser lo suficientemente eficiente para ser usado en este tipo de aplicaciones, porque se supone que los diferentes perfiles serían implementados por la vía de reducir el tamaño de un sistema POSIX completo, de tipo UNIX. Puesto que una implementación típica de POSIX puede tener requisitos de memoria del orden de las decenas de megabytes, parecería altamente improbable que un sistema operativo conforme al modelo POSIX del "tostador" pudiera ser suficientemente pequeño como para ser útil.

El uso del perfil POSIX de Sistema de Tiempo Real Mínimo proporciona un alto grado de portabilidad entre diversas implementaciones de sistemas operativos conformes a POSIX. Puesto que una aplicación conforme al perfil mínimo se puede ejecutar en implementaciones conformes a cualquiera de los otros perfiles, es posible por ejemplo realizar las pruebas funcionales de la aplicación sobre una estación de trabajo, en lugar de hacer todas las pruebas sobre el hardware final o un emulador especializado.

En la sección 2 de este artículo, describiremos los perfiles POSIX de tiempo real con suficiente nivel de detalle como para que queden claras los objetivos y la funcionalidad básica de cada uno de ellos. Luego, en la sección 3 usaremos varios ejemplos para describir los requerimientos de servicios del sistema operativo impuestos por las aplicaciones para las que el perfil Mínimo de Tiempo Real está pensado, y mostraremos cómo se pueden cumplir estos requisitos con el perfil mínimo. En la sección 4, describiremos un conjunto de características de diseño para una implementación práctica pero mínima de un sistema operativo acorde con este perfil, y seguidamente en la sección 5 haremos algunas consideraciones sobre las prestaciones que se prevén alcanzar. Finalmente extraeremos algunas conclusiones.

2. El Perfil IEEE 1003.13 de Sistema de Tiempo Real Mínimo

2.1. La familia de estándares POSIX

POSIX es el acrónimo de Interfaz de Sistemas Operativos Portables (Portable Operating Systems Interface). 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.

Tabla I. Algunos de los estándares base POSIX

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

El POSIX es una familia de estándares en evolución, cada uno de los cuales cubre diferentes aspectos de los sistemas operativos. Algunos 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. Estas interfaces permiten al programa de aplicación invocar directamente los servicios del sistema operativo. El estándar no especifica cómo se implementan estos servicios, sino únicamente su semántica; de este modo, los implementadores pueden elegir la implementación que estimen más conveniente siempre que sigan la especificación de la interfaz. La mayoría de los estándares base están especificados para el lenguaje de programación C. La tabla I lista algunos de los estándares básicos POSIX que son más importantes para las aplicaciones 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 lenguajes de programación diferentes. Los lenguajes que se han usado hasta el momento son el Ada y el Fortran. La tabla II lista algunos de ellos.

3) *Entorno de Sistemas Abiertos*. Estos estándares incluyen una guía al entorno POSIX y perfiles de aplicación. Los perfiles son subconjuntos de los servicios POSIX que se requieren para un determinado ámbito de aplicación. Los perfiles representan un importante mecanismo para definir de forma estándar un conjunto bien definido de

implementaciones de sistemas operativos, adecuadas para áreas de aplicación específicas.

Tabla II. Algunos de los *Bindings* POSIX

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

El POSIX incluye servicios de sistema operativo para muchos entornos de aplicación. Un área de gran importancia que está siendo estandarizada en POSIX es la de los sistemas de tiempo real. El propósito de las Extensiones de Tiempo Real es conseguir la portabilidad a nivel de código fuente de aplicaciones con requisitos temporales. Hasta ahora, esta portabilidad era imposible debido a las profundas diferencias entre las interfaces y modelos subyacentes de los diferentes sistemas operativos de tiempo real existentes. El objetivo de portabilidad se ha conseguido en el POSIX definiendo servicios del sistema operativo adicionales (1003.1b, 1003.1c) que se han diseñado específicamente para ser implementables con tiempos de respuesta acotados, y para proporcionar un comportamiento temporal de la aplicación predecible. La mayoría de los fabricantes de sistemas operativos de tiempo real incluyen ya o planean incluir en breve una interfaz POSIX a sus implementaciones.

Muchas aplicaciones de tiempo real, como las de los sistemas empotrados pequeños, tienen restricciones físicas especiales que requieren de sistemas operativos con una funcionalidad reducida. Por ejemplo, existen muchos sistemas que no tienen disco duro, o no tienen unidad hardware de gestión de memoria, y que tienen una cantidad de memoria muy pequeña. Para estos sistemas, es preciso que el estándar permita implementaciones que soporten sólo un pequeño subconjunto de las funciones POSIX. El grupo de trabajo de tiempo real del POSIX está abordando este problema, y está actualmente definiendo cuatro perfiles de aplicaciones de tiempo real, bajo el estándar 1003.13[2]: para sistemas empotrados pequeños, para controladores de tiempo real, para sistemas empotrados grandes, y para sistemas multipropósito con todas las facilidades pero con requisitos de tiempo real. Este artículo se centra en el menor de los perfiles, pensado para sistemas empotrados pequeños.

2.2 Repaso de los estándares base POSIX de tiempo real

Los estándares POSIX ya aprobados que son de interés para las aplicaciones de tiempo real son el 1003.1, 1003.1b, y 1003.1c. Han sido publicados conjuntamente como ISO/IEC Std. 9945-1:1996. A continuación describiremos brevemente los servicios básicos definidos en estos estándares:

- *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 (p.e. datos, dispositivos de I/O, 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

[2] Se espera que el estándar 1003.13 quede aprobado durante 1997

servicios de temporización muy primitivos, servicios de ficheros y directorios, entrada/salida, control de terminales y bases de datos del sistema para usuarios y grupos.

- *Las extensiones de tiempo real, 1003.1b*: El estándar define las extensiones básicas de tiempo real que se consideraron esenciales para dar soporte a aplicaciones con requisitos de tiempo real. Los servicios descritos 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. Incluidos entre estos servicios podemos encontrar: la sincronización de procesos mediante semáforos contadores; objetos de memoria compartida, que permiten a los procesos con espacios de direcciones independientes compartir información; colas de mensajes, que permiten el intercambio de eventos o datos entre procesos; entrada/salida asíncrona, que permite a la aplicación ejecutarse en paralelo con las operaciones de entrada/salida; y entrada/salida sincronizada, que permite un mayor grado de predecibilidad en las operaciones de entrada/salida sobre ficheros.

  - *Servicios que se necesitan para conseguir un comportamiento temporal predecible*: Incluyen la planificación expulsora de procesos mediante políticas basadas en prioridades fijas; la inhibición de memoria virtual para el espacio de direcciones de un proceso, con objeto de conseguir un tiempo de acceso a memoria predecible; señales de tiempo real, que proporcionan un comportamiento más predecible para las señales; y relojes y temporizadores, que permiten la gestión del tiempo desde la aplicación.

- *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 eficientes debido al requisito de espacios de direcciones independientes y al estado asociado al proceso, que es voluminoso. Un cambio de contexto entre procesos es complejo, y normalmente requiere reprogramar la unidad de gestión de memoria (MMU), entre otras cosas. Por el contrario, las tareas de un núcleo de tiempo real de alta eficiencia comparten el mismo espacio de direcciones y tienen un estado asociado pequeño. Por tanto, con el propósito de incrementar la eficiencia, se introdujeron en el POSIX los threads, o procesos ligeros, como un mecanismo de concurrencia de alta eficiencia. Bajo la Extensión de Threads, cada proceso POSIX puede tener múltiples flujos de control concurrentes, todos ellos compartiendo el mismo espacio de direcciones. El estándar proporciona servicios para la gestión, cancelación, planificación, y sincronización de threads.

Además de los estándares mencionados, existen más 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 de servidor esporádico, e información para implementación de ficheros de tiempo real. Los servicios definidos en 1003.1j incluyen primitivas adicionales de sincronización para multiprocesadores, gestión de memoria de diversos tipos, y los nuevos relojes monotónico y sincronizado.

2.3 Los 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. Por tanto, 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. Aunque hay muchas pequeñas diferencias entre los diferentes perfiles, las principales diferencias radican en la presencia o no de múltiples procesos, y la presencia o no de un sistema de ficheros jerárquico. La Tabla III muestra las características de los cuatro perfiles de tiempo real, de acuerdo con estas propiedades. Los servicios específicos definidos en el perfil mínimo de tiempo real, y por tanto pensados para sistemas empotrados pequeños, se muestran en la Tabla IV.

Tabla III. Principales características de los subconjuntos POSIX de tiempo real

Perfil	Nombre del perfil	Sistema de ficheros	Múltiples procesos	Múltiples threads	Plataforma típica
PSE51	Sistema de tiempo real mínimo	No	No	Si	Sistema 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 diskette 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

Tabla IV. Opciones y unidades de funcionalidad requeridas por el perfil mínimo

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 mensajes I/O sincronizada	Threads Atributo de tamaño de *stack* de thread Atributo de dirección del *stack* Planificación de threads por prioridades Protocolo de herencia de prioridad Protocolo de protección de prioridad

3. Requisitos Mínimos de Aplicaciones de Tiempo Real

En esta sección presentaremos los requisitos mínimos típicos de las aplicaciones de tiempo real empotradas, y mostraremos como el Perfil de Sistema de Tiempo Real Mínimo definido en el POSIX (P1003.13-PSE51) proporciona servicios que cubren estos requisitos.

Puesto que una aplicación de tiempo real necesita interaccionar y/o controlar un entorno que cambia con el tiempo, la aplicación tiene requisitos de tiempo real impuestos por este entorno. Con objeto de garantizar que estos requerimientos temporales se cumplirán, el comportamiento temporal de la aplicación debe ser predecible. El hecho de que el entorno es complejo y generalmente tiene diversas partes que deben ser controladas o actuadas simultáneamente, conduce a una arquitectura software concurrente. Las tareas de este software concurrente deben de ser planificadas de una forma predecible, para poder mantener la garantía de cumplimiento de los requisitos temporales. Puesto que estas tareas necesitan cooperar entre sí y compartir recursos, se precisan primitivas de sincronización con características de tiempo real. Las acciones ejecutadas por un software de este tipo necesitan estar adecuadamente temporizadas mediante el uso de un reloj hardware a través de servicios de temporización proporcionados por el sistema operativo. A continuación, repasaremos cada uno de estos requerimientos con mayor nivel de detalle.

3.1 Concurrencia

La concurrencia en el perfil mínimo de tiempo real de POSIX 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 tiene sus propios recursos tales como una pila (*stack*), una política y parámetros de planificación, y datos específicos del thread. Tal como mencionamos anteriormente, ya que todos los threads de un proceso comparten el mismo espacio de direccionamiento, pueden ser construidos con un grado de eficiencia superior al de los procesos. Éstos requieren la separación de espacios de direcciones y, por tanto, del uso de una MMU. En el perfil mínimo sólo hay un proceso, con múltiples threads. El sistema comienza por crear el llamado 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 la manipulación de atributos de threads, creación de threads, terminación de threads, y servicios de cancelación de threads.

- *Manipulación de atributos de thread*: Cada thread tiene un conjunto de atributos a los que normalmente se les asigna un valor cuando se crea el thread (aunque hay algunos atributos que se pueden cambiar dinámicamente). Los atributos que se usan 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 (*pthread_attr_init*(), *pthread_attr_destroy*()), y hay una pareja de funciones asociada a cada atributo, una para poner el valor del atributo y otra para leerlo.

  Los principales atributos definidos son el tamaño mínimo de *stack*, la dirección del *stack*, 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*() para esperar a su terminación. Hay otros atributos que definen la política y parámetros de planificación, pero los explicaremos un poco más adelante.

- *Creación de Threads*: Una vez que el objeto de atributos de 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 deberemos especificar 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. La función de creación retorna un identificador de thread (del tipo *pthread_t*), que podrá ser usado en llamadas posteriores para solicitar algún servicio referido a ese thread en particular.

- *Terminación y cancelación de threads:* Un thread puede terminarse a sí mismo llamando a la función *pthread_exit*(), o si alcanza el final lógico de la función que el thread comenzó a ejecutar cuando se creó. También es posible terminar un thread llamando a *pthread_cancel*(), y especificando el identificador del thread deseado. En ambos casos, el thread ejecuta las operaciones de terminación que hayan sido registradas mediante llamadas a *pthread_cleanup_push*(). Por motivos de protección, es posible que un thread inhiba la cancelación desde otros threads, y que la vuelva a habilitar más adelante. La cancelación del thread puede verse retrasada hasta que la aplicación ejecute una llamada al sistema operativo declarada en el estándar como un punto de cancelación.

La tabla V muestra un ejemplo de un programa que crea dos threads que escriben mensajes en la pantalla de forma periódica. El periodo se le pasa a cada thread como parámetro. Debe observarse que este parámetro está siendo compartido por el thread principal y el nuevo thread, y por tanto, para evitar errores de inconsistencia, no debería de ser modificado después de crear el thread. Los threads del ejemplo se han creado en estado independiente. Por motivos de espacio, hemos omitido todo el código de detección y tratamiento de errores, que siempre deberá de incluirse en una aplicación real.

3.2 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 (o desalojante) por prioridades fijas. 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 por 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 por prioridades, que usa orden FIFO (el primero en llegar es el primer en ser atendido) para determinar el orden en el que se ejecutan los threads de la misma prioridad.

Tabla V. Ejemplo de creación de threads

```
#include <pthread.h>
/* Thread que escribe periódicamente en la pantalla */
void * periodic (void *arg)
{
          int period; /*en segundos*/

          period = *((int *)arg);
          while (1) {
                printf("En el thread con periodo %d\n",period);
                sleep (period);
          }
}
/* Programa principal, que crea dos threads periódicos */
main ()
{
          pthread_t th1,th2;
          pthread_attr_t attr;
          int period1=2;
          int period2=3;

          /* Crear el objeto de atributos y poner el estado de devolución de
recursos*/
          pthread_attr_init(&attr);
          pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
          /* Crear los threads */
          pthread_create(&th1,&attr,periodic,&period1);
          pthread_create(&th2,&attr,periodic,&period2);
          ...
}
```

- SCHED_RR: este es el nombre de la política de planificación expulsora por prioridades que usa un esquema cíclico para planificar threads de la misma prioridad. Después de que un thread con esta política ha ejecutado durante una rodaja temporal de duración fija, se planifica, si existe, al próximo thread de la misma prioridad.

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 (si se permite la asignación dinámica de threads a procesadores, las reglas de planificación se interpretarán en una forma definida por la implementación). Existe una tercera política de planificación (SCHED_OTHER) con un comportamiento definido por la implementación, y que se definió para permitir la compatibilidad con implementaciones preexistentes. Puede parecer un poco 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 definidas 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 SCHED_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. El nivel de prioridad mínimo y máximo se pueden consultar (*sched_get_priority_min*(), *sched_get_priority_max*()). Los atributos de política de planificación y prioridad forman parte del objeto de atributos de 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 (mediante las funciones *pthread_setschedparam*() y *pthread_getschedparam*()).

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_params*, 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 los 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.

La tabla VI muestra un ejemplo que hace uso de las operaciones de planificación de threads. El programa muestra el mismo programa principal que se usó en el ejemplo de la tabla V, y que creaba dos threads periódicos, pero definiendo los atributos de planificación. Observar que el valor del atributo *inheritsched* se ha situado para que los atributos de planificación sean explícitos. el primer thread se crea con la prioridad mínima para la política SCHED_FIFO, mientras que el segundo thread tiene el siguiente valor de prioridad, y tiene por tanto una prioridad mayor.

Tabla VI. Ejemplo del uso de atributos de planificación de threads

```
#include <sched.h>
#include <pthread.h>
/* Programa principal que crea dos threads periódicos */
main ()
{
          pthread_t th1,th2;
          pthread_attr_t attr;
          int period1=2;
          int period2=3;
          struct sched_param my_params;

          /* Crear el objeto de atributos y colocar cada atributo */
          pthread_attr_init(&attr);
          pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
          pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED);
          pthread_attr_setschedpolicy(&attr,SCHED_FIFO);
          my_params.sched_priority= sched_get_priority_min(SCHED_FIFO);
          pthread_attr_setschedparams(&attr,&my_params);

          /* Crear los threads */
          pthread_create(&th1,&attr,periodic,&period1);

          my_params.sched_priority= (sched_get_priority_min(SCHED_FIFO)+1);
          pthread_attr_setschedparams(&attr,&my_params);
          pthread_create(&th2,&attr,periodic,&period2);
          ...
}
```

3.3 Sincronización

El estándar 1003.1c proporciona dos primitivas de sincronización para threads: el mutex para el acceso mutuamente exclusivo a recursos compartidos, y las variables condicionales para la sincronización de espera. Ambas primitivas se usan a través de objetos del tipo *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:

- *tomar o bloquear*: si el mutex está libre, se bloquea 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 le 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 [Sha, et al, 1990], 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 real estricto. El protocolo que muestra mejor tiempo de respuesta en el peor caso es el de protección de prioridad (también llamado protocolo de techo de prioridad

en Ada 95 y protocolo de techo de prioridad emulado en algunas referencias de la bibliografía), mientras que el mejor tiempo de respuesta promedio corresponde al protocolo de herencia de prioridad.

Los atributos de creación del 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.

De los atributos de arriba, sólo el techo de prioridad se puede cambiar dinámicamente, después de que el mutex haya sido creado.

La tabla VII muestra un ejemplo de un módulo de programa que representa un objeto compartido abstracto, con operaciones para crearlo, modificarlo, y leerlo. Las operaciones de modificación y lectura usan un mutex para conseguir la exclusión mutua. Se usa el protocolo de protección de prioridad, y el techo de prioridad del mutex se especifica como un parámetro en la llamada de creación. Debe ser colocado a un valor de prioridad mayor o igual que la más alta de las prioridades de los threads que pueden usar ese recurso compartido.

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 reactivan uno o más 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 condicional 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.

La tabla VIII muestra un ejemplo del uso de una variable condicional. El ejemplo implementa un *buffer* que puede ser usado por un conjunto de threads productores y consumidores, para el intercambio de mensajes. Las operaciones del *buffer* usan una cola que está construida como una máquina abstracta de datos, y cuyas operaciones están especificadas en el fichero `buffer_queue.h`. El *buffer* usa un mutex para garantizar la exclusión mutua durante el acceso a la información interna del *buffer*, y una variable condicional para despertar al consumidor (o consumidores) cuando hay datos disponibles en el *buffer*. Por sencillez del

Tabla VII. Implementación de un objeto compartido abstracto

```
/*****************************************************************/
/* Fichero shared_object.h: Prototipo del objeto*/
typedef struct {
 ...
} shared_t;

void sh_create (int prio_ceiling, const shared_t *initial_value);
     /* sh_create debe terminar antes de que el objeto sea usado */
void sh_modify (const shared_t *new_value);
void sh_read (shared_t *current_value);

/*****************************************************************/
/* Fichero shared_object.c: Implementación del objeto */
#include <sched.h>
#include <pthread.h>
#include "shared_object.h"

shared_t object;
pthread_mutex_t the_mutex;

void sh_create (int prio_ceiling, const shared_t *initial_value)
{
      pthread_mutexattr_t attr;

      pthread_mutexattr_init(&attr);
      pthread_mutexattr_setprotocol (&attr,PTHREAD_PRIO_PROTECT);
      pthread_mutexattr_setprioceiling (&attr,prio_ceiling);
      pthread_mutex_init(&the_mutex,&attr);
      object=*initial_value;
      pthread_mutexattr_destroy(&attr);
}

void sh_modify (const shared_t *new_value)
{
    pthread_mutex_lock(&the_mutex);
    object=*new_value;
    pthread_mutex_unlock(&the_mutex);
}

void sh_read (shared_t *current_value)
{
    pthread_mutex_lock(&the_mutex);
    *current_value=object;
    pthread_mutex_unlock(&the_mutex);
}
```

ejemplo, hemos construido en *buffer* con una operación de extracción bloqueante para los consumidores, pero una operación de inserción no bloqueante para los productores (si el *buffer* está lleno, la operación de inserción falla). Podríamos haber usado una segunda variable condicional para proporcionar una operación de inserción bloqueante. Como podemos ver en el ejemplo, una operación de inserción correcta toma el mutex, inserta el mensaje en la cola, señaliza la condición para despertar a un consumidor, si existe, que esté esperando a un mensaje y, finalmente, libera el mutex. La operación de extracción toma el mutex para evaluar el predicado lógico (si la cola está vacía o no). Si la cola está vacía, el thread se suspende en la condición invocando la operación de espera. La llamada a la función *pthread_cond_wait*() está incluida en el interior de un lazo *while* porque la operación de señalización puede despertar a más de un thread[3]; por tanto, si un thread consumidor se encuentra después de despertarse que la cola está todavía vacía (porque otro thread se le ha adelantado a realizar la operación de extracción), entonces continúa esperando. Cuando el thread encuentra que hay

[3] Este comportamiento se especificó en el estándar para permitir una implementación más eficiente de las variables condicionales en sistemas multiprocesadores.

Tabla VIII. Implementación de un *buffer* abstracto

```
/*****************************************************************/
/* Fichero one_way_buffer.h: Prototipo del objeto*/
typedef struct {
   ...
} buf_data_t;
#define BUFFER_OK    0
#define BUFFER_FULL -1

void buf_create (int prio_ceiling);
     /* buf_create debe completarse antes de que el objeto sea usado */
int  buf_insert (const buf_data_t *message);
     /* buf_insert retorna BUFFER_OK o BUFFER_FULL, y nunca se bloquea */
void buf_extract (buf_data_t *message);
     /* buf_extract se bloquea si el buffer está vacío */

/*****************************************************************/
/* Fichero buffer_queue.h: Prototipo del objeto*/
#include "one_way_buffer.h"
#define MAX_DATA 100

void queue_insert  (const buf_data_t *message);
void queue_extract (buf_data_t *message);
int  queue_isfull  (void);
int  queue_isempty (void);

/*****************************************************************/
/* Fichero one_way_buffer.c: Implementación del objeto */
#include <sched.h>
#include <pthread.h>
#include "buffer_queue.h"

pthread_mutex_t the_mutex;
pthread_cond_t the_condition;

void buf_create (int prio_ceiling)
{
      pthread_mutexattr_t attr;

      pthread_mutexattr_init(&attr);
      pthread_mutexattr_setprotocol (&attr,PTHREAD_PRIO_PROTECT);
      pthread_mutexattr_setprioceiling (&attr,prio_ceiling);
      pthread_mutex_init(&the_mutex,&attr);
      pthread_mutexattr_destroy(&attr);
      /* Initialize the condition variable with default attributes */
      pthread_cond_init(&the_condition,NULL);
}

int  buf_insert (const buf_data_t *message)
{
    int ret_value;

    pthread_mutex_lock(&the_mutex);
    if (queue_isfull()) ret_value=BUFFER_FULL;
    else {
        ret_value=BUFFER_OK;
        queue_insert(message);
        pthread_cond_signal(&the_condition);
    }
    pthread_mutex_unlock(&the_mutex);
    return (ret_value);
}

void buf_extract (buf_data_t *message)
{
    pthread_mutex_lock(&the_mutex);
    while (queue_isempty()) pthread_cond_wait(&the_condition,&the_mutex);
    queue_extract(message);
    pthread_mutex_unlock(&the_mutex);
}
```

un mensaje disponible en el *buffer*, lo extrae de la cola con el mutex tomado, y luego libera el mutex.

3.4 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 describimos brevemente 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 ms. Esto implica que el diseñador o diseñadora de la aplicación debe seleccionar cuidadosamente la implementación POSIX a usar, de modo que soporte al menos el nivel de resolución de reloj requerido por su aplicación. El estándar define 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, es bien conocido que una operación de retraso relativo no sirve para crear acciones periódicas. Por esta razón se está estandarizando en el POSIX P1003.1j una función de retraso basado en una hora absoluta. Mientras tanto, la activación periódica de threads se puede conseguir mediante el uso de temporizadores, aunque con la desventaja de que el número total de temporizadores (y por tanto de threads periódicos) está limitado.

- *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[4] 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.

La tabla IX muestra un ejemplo de la implementación de threads periódicos mediante temporizadores. La primera expiración del temporizador se hace igual al periodo del thread, y el resto de las expiraciones se programan para ocurrir periódicamente, también con el periodo del thread. Este periodo y el número de señal a utilizar para la notificación de las

[4] Una señal es un mecanismo usado por el sistema operativo para notificar a un proceso o a un thread de que ha ocurrido un evento concreto. Las señales se pueden enmascarar o desenmascarar usando una máscara asociada a cada thread. Las señales enmascaradas se pueden aceptar (es decir, podemos esperar a que ocurran) mediante una llamada a una operación *sigwait*().

expiraciones del temporizador se le pasan al thread como parámetros. El thread espera a la señal mediante la función *sigwait*(), que requiere que la señal esté enmascarada en ese thread, y en todos los demás threads del proceso[5]. Puesto que la máscara de señales se hereda del thread padre durante la creación, la mejor solución es que todas aquellas señales que van a ser usadas en operaciones *sigwait*() sean enmascaradas por el thread principal, y que posteriormente ningún thread toque su máscara de señales.

Tabla IX. Ejemplo de un thread periódico implementado mediante un temporizador

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

/* Tipo de datos para el parámetro del thread */
struct periodic_data {
    struct timespec per;
    int sig_num;
};

/* Thread 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;

    /* Copiar el parámetro de entrada en una variable local */
    my_data = * (struct periodic_data*)arg;
    /* Crear el temporizador */
    event.sigev_notify = SIGEV_SIGNAL;
    event.sigev_signo = my_data.sig_num;
    timer_create (CLOCK_REALTIME, &event, &timer_id);
    /* Armar el temporizador */
    timerdata.it_interval = my_data.per;
    timerdata.it_value = my_data.per;
    timer_settime(timer_id, 0, &timerdata, NULL);
    /* Preparar el conjunto de señales usado en la llamada a sigwait() */
    /* La máscara de señales se heredará del thread padre */
    sigemptyset (&set);
    sigaddset(&set,my_data.sig_num);

    /* Comenzar el lazo periódico */
    while (1) {
        ... do useful work;
        sigwait(&set,&received_sig);
    }
}
```

3.5 Entrada/salida

La entrada/salida básica se puede hacer bajo el perfil mínimo de dos formas distintas:

- *Entrada/Salida mapeada en memoria*: Puesto que el perfil mínimo no requiere la protección de memoria, si el espacio de direcciones lógicas coincide con el espacio de

[5] Si no, la señal se entrega a un thread que no la haya enmascarado.

direcciones físicas, es posible hacer entrada/salida directamente sobre dispositivos mapeados en memoria, leyendo o escribiendo en las posiciones de memoria apropiadas. Sin embargo, esta aproximación no es portable. Para conseguir la portabilidad, la dirección del dispositivo se puede obtener a partir del descriptor del fichero del dispositivo mediante la función *mmap*(). Esta aproximación es válida tanto si los espacios de direcciones lógico y físico coinciden, como si no.

- *Entrada/salida mediante dispositivos*: La entrada/salida puede programarse también de forma portable usando ficheros y servicios básicos de entrada/salida. Aunque el perfil de sistema de tiempo real mínimo no incluye un sistema de ficheros jerárquico, sí permite abrir ficheros asociados a dispositivos de I/O (*open*()), usarlos para leer o escribir (*read*(), *write*()), y cerrarlos (*close*()). Puesto que estos ficheros de dispositivo no pueden ser creados por la aplicación, deberán de ser creados durante la configuración del sistema. Los drivers de dispositivos en sí no serán portables, ya que de momento ningún estándar POSIX define servicios que permitirían al driver acceder a los recursos del sistema, tales como dispositivos hardware, o interrupciones. Existe un esfuerzo de estandarización a nivel industrial, llamado Uniform Driver Interface (UDI) [UDI, 1995], que trata de estandarizar una interfaz que permitiría a los diseñadores de drivers de I/O escribirlos de forma portable.

4. Consideraciones de diseño de un sistema operativo mínimo

En esta sección describiremos las principales características del diseño de un sistema operativo sencillo conforme al perfil de sistema de tiempo real mínimo, basadas en nuestra experiencia en la construcción de sistemas operativos de tiempo real. El propósito de la discusión es reforzar nuestra aseveración de que es posible construir un sistema operativo conforme al perfil mínimo que pueda ser usado de forma efectiva y eficiente en el tipo de sistemas empotrados pequeños para los que se diseñó el perfil del "tostador".

En lugar de comenzar con un sistema operativo de tipo UNIX con todas sus funciones, y luego eliminar servicios para cumplir con el perfil, comenzaremos con un diseño clásico de un ejecutivo de tiempo real, asegurándonos de suministrar todas las funciones especificadas en el perfil. A los efectos de este artículo, describiremos el sistema operativo en términos de un conjunto de módulos que gestionan las funciones más importantes, con una descripción de las estructuras de datos primordiales que son necesarias para construir todos los servicios.

Para comenzar, asumiremos que nuestro sistema operativo reducido no va a proporcional protección de memoria entre la aplicación y el sistema operativo, puesto que el procesador puede que carezca de funciones de gestión de memoria, o que éstas estén inhibidas. Asimismo, asumiremos que el modelo de memoria es continuo, es decir, no nos preocuparemos de segmentos o páginas de memoria.

En este modelo simplificado de memoria, las funciones de inhibición de memoria virtual se implementan como operaciones nulas, ya por definición la memoria virtual está inhibida siempre. Aún así, debemos suministrar estas funciones, e incluso aconsejar su uso desde la aplicación, con objeto de que ésta sea portable a otras plataformas que sí dispongan de memoria virtual (por ejemplo durante la depuración y prueba del programa). Otra

consecuencia de nuestro modelo de memoria es que la función *mmap*() no realiza realmente ningún mapeado de memoria, pero puede ser usada para retornar la dirección asociada con un dispositivo de entrada/salida mapeada en memoria, dado el descriptor de fichero asociado a ese dispositivo.

En el perfil mínimo, puesto que sólo hay un proceso, no se requieren las funciones de nivel de proceso, como la función *fork*(). Asimismo, debido a que sólo hay un proceso, se usará para la planificación de threads el ámbito de contienda a nivel de proceso (PTHREAD_-SCOPE_PROCESS) definido en el estándar [ISO/IEC Std. 9945-1:1996].

Consistentemente con los requisitos de planificación del POSIX, sólo implementaremos las políticas de planificación SCHED_FIFO y SCHED_RR. La política SCHED_OTHER será idéntica a SCHED_FIFO. Supondremos que, aunque las aplicaciones planificadas mediante un ejecutivo cíclico son posibles bajo este estándar, la mayoría de las aplicaciones empotradas de tiempo real estricto y no estricto se construirán con arquitecturas basadas en planificación por prioridades [Locke, 1992], como las que proporciona el estándar POSIX.

4.1 Estructuras de datos básicas del sistema operativo

Las estructuras de datos básicas de un sistema operativo mínimo serían:

- Una cola de threads activos o listos para ejecutar, en orden decreciente de prioridad.
- Una cola de threads bloqueados, que no están dispuestos para ejecutar, en orden arbitrario.
- Una cola de eventos temporales, ordenada crecientemente por la hora a la que debe ocurrir cada evento.

En el instante de inicialización, sólo hay un único thread conocido por el sistema operativo, el thread principal proporcionado al configurar el sistema, que será responsable de crear a otros threads necesarios para la aplicación. Este thread está en la cola de threads activos. La cola de threads bloqueados y la cola de eventos temporales están vacías. Además, puesto que el estándar permite que el número máximo de threads del sistema esté acotado, el tamaño de todas estas colas es acotado. Todos los registros de thread y de eventos temporales están preasignados desde el instante de configuración del sistema, y las colas se implementan encadenando estos registros en el orden apropiado para cada cola. Por tanto, toda la memoria del sistema operativo se puede preasignar durante la configuración del sistema, incluyendo el espacio de *stack* para cada thread.

4.2 Gestores de interrupciones y módulo de planificación

Un sistema de tiempo real mínimo se puede diseñar en base a un conjunto de gestores de interrupción para uno de los posibles eventos que puedan ocurrir. Para una aplicación mínima, el conjunto de gestores de interrupción requeridos es:

- Temporización
- Entrada/Salida
- Servicio del Ejecutivo (llamadas al sistema)

Se podrían proporcionar otros gestores para operaciones hardware especializadas. La estructura de todas las interrupciones es la misma. Cuando ocurre una interrupción, el estado del thread en ejecución se salva, se ejecutan las acciones específicas asociadas a esa interrupción y, finalmente, se invoca al planificador del sistema operativo para restaurar el estado de ejecución del thread de más alta prioridad (posiblemente el thread que estaba siendo ejecutado) de los que están listos para ejecutar.

El procesado de interrupción específico a la interrupción del temporizador hardware consiste en eliminar el evento que está a la cabeza de la cola de eventos temporales. Si el evento correspondía a una operación *nanosleep*(), el thread asociado se mueve de la cola de threads bloqueados a la de threads activos. Si es preciso generar una señal, se ejecuta la acción apropiada a esa señal (ver la sección 4.3). Si el evento es debido a que se ha sobrepasado el tiempo límite de un servicio del sistema (como *pthread_cond_timedwait*()), el thread correspondiente se mueve de la cola de threads bloqueados a la de threads activos (por supuesto, con el valor de retorno apropiado). Finalmente, el temporizador hardware se reprograma para causar una nueva interrupción en el instante correspondiente a la nueva cabeza de la cola de eventos temporales, si existe.

El procesado de interrupción específico a las interrupciones de entrada/salida consiste en invocar el driver de I/O asociado, para que gestione las transferencias de datos necesarias y reactive a los threads afectados si es necesario. Si es preciso generar una señal, se ejecutará la acción asociada a esa señal (ver la sección 3.4).

El procesado de interrupción específico al servicio del ejecutivo consiste en llamar a la función apropiada del sistema operativo, para ejecutar el servicio solicitado. Esto incluye todas las llamadas al sistema tales como *open*() *o pthread_create*()). Las llamadas al sistema pueden implicar el bloqueo momentáneo de las colas del sistema mientras se ajustan, por ejemplo para añadir o eliminar un thread a la cola de threads activos. Si es preciso generar una señal, se ejecutarán las acciones asociadas a esa señal (ver la sección 4.3).

4.3 Módulo de gestión de 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 nuestro sistema operativo mínimo, las señales se pueden generar en tres lugares: interrupciones del temporizador, interrupciones de entrada/salida, e 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 *sigwait*() para esperarla) la puede recibir. Por supuesto, nuestra implementación tiene un solo proceso, por lo que todos los threads del sistema son candidatos a recibir señales enviadas a ese único proceso.

Las acciones asociadas a la generación de una señal son las siguientes: En el momento de la generación de la señal, se elige para su entrega a un thread que tenga la señal desenmascarada. En nuestra implementación, se le entregará al primer thread que se encuentre. La acción asociada con la entrega de la señal es elegida por la aplicación con

antelación a la generación de la señal de entre tres posibilidades: ignorar la señal, ejecutar la acción por defecto (que usualmente implica que la aplicación completa se termina), o ejecutar un manejador de señal. En este último caso, el gestor de interrupción que generó la señal restaura el estado de la aplicación, y se invoca al manejador de señal.

Cuando se genera una señal y el thread la tiene enmascarada, si el thread estaba esperando bloqueado en una operación *sigwait*(), el thread se mueve de la cola de threads bloqueados a la de threads activos. En caso contrario se dice que la señal está pendiente, y la señal es añadida a la cola de señales pendientes correspondiente a su número de señal. El tamaño de esta cola está acotado. La señal permanece pendiente hasta que la máscara de señales del thread se cambia, o hasta que el thread ejecuta una operación *sigwait*().

4.4 Módulo de creación y terminación de threads

La creación de threads se consigue mediante una llamada a la función *pthread_create*(). En nuestra implementación, esta llamada causa que un registro de thread que esté libre se inicialice y se asigne al nuevo thread, y que este thread se añada a la cola de threads activos. El registro contiene el estado inicial del thread, sus atributos, y un enlace con el thread padre. Cuando el thread se termina, su referencia en la cola de threads activos o bloqueados se elimina. Cuando los recursos usados por el thread se devuelven al sistema (en el momento de la terminación para un thread independiente, o cuando se ejecuta una operación *pthread_join*() sobre el thread, si es sincronizado), el registro del thread se marca como libre y puede ser reutilizado posteriormente para la creación de un nuevo thread.

4.5 Módulo de sincronización

La sincronización entre threads se realiza mediante los mutex, variables condicionales, semáforos, y colas de mensajes. Tanto los mutex como las variables y los semáforos sin nombre se crean en el espacio de variables de la aplicación, normalmente usando variables estáticas. Las colas de mensajes y los semáforos con nombre son alojados estáticamente por el sistema operativo durante la configuración, aunque sin ningún nombre asociado. Durante la ejecución del sistema los semáforos con nombre se inicializan mediante la función *sem_open*(), momento en el que se les asigna un nombre; similarmente, a las colas de mensajes se les asigna el nombre al ser inicializadas mediante *mq_open*().

4.5.1 Los mutex

Cuando la operación *pthread_mutex_lock*() se invoca sobre un mutex libre, el mutex se bloquea y el thread que invoca la operación se convierte en el propietario del mutex. Si el mutex tiene el atributo de protocolo de protección de prioridad, el thread propietario hereda el techo de prioridad del mutex. Si la prioridad del mutex cambia debida a este efecto de herencia (la prioridad instantánea de un thread es el máximo de su prioridad base y de todas las prioridades heredadas), el thread se mueve al lugar apropiado a su nueva prioridad, dentro de la cola de threads activos.

Cuando la operación *pthread_mutex_lock*() se invoca sobre un mutex bloqueado, el thread que invoca la operación se mueve de la cola de threads activos a la de threads bloqueados, y se

añade además a la cola de espera de ese mutex, en orden de prioridad. Si el mutex tiene el atributo de protocolo de herencia de prioridad, el thread que en ese momento es el propietario del mutex hereda la prioridad base (y las prioridades heredadas) del thread que se acaba de suspender. Si como resultado de esta herencia la prioridad instantánea del thread propietario del mutex se incrementa, este thread se inserta en la cola de threads activos en la misma posición que tenía el thread recién suspendido.

Cuando se ejecuta la operación *pthread_mutex_unlock*(), todas las prioridades heredadas por el thread que invoca la operación como consecuencia de haber sido el propietario de ese mutex dejan de tener efecto, y la prioridad del thread se restaura a un nuevo valor en caso necesario. Luego, se comprueba la lista de threads que están en espera de tomar el mutex. Si no hay threads en espera, el mutex se libera. En caso contrario, el primero de los threads que está esperando se elimina de la cola del mutex y se pasa a la cola de threads activos. Este thread se convierte en el nuevo propietario del mutex y su prioridad se sitúa al valor apropiado, dependiendo del protocolo de sincronización utilizado. Si el mutex tiene el atributo de herencia de prioridad, el nuevo propietario del mutex se inserta en la cola de threads activos en la misma posición que tenía el thread que invocó la operación, si es que la prioridad del nuevo propietario es mayor.

4.5.2 Variables Condicionales

Cuando se ejecuta la operación *pthread_cond_wait*(), el mutex asociado se libera en la forma indicada arriba, el thread se añade a la cola de espera de la variable condicional, en orden de prioridad, y se mueve de la cola de threads activos a la de threads bloqueados. Cuando se ejecuta *pthread_cond_signal*(), si hay threads esperando en la cola, el primero de ellos se elimina de la cola y se invoca una operación de tomar el mutex, en su nombre. Como resultado, si el mutex estaba ya tomado, el thread recién despertado se añade a la cola de threads que están en espera del mutex, en orden de prioridad. Si el mutex no estaba tomado, el thread recién despertado se convierte en su propietario, y se mueve a la cola de threads activos.

4.5.3 Semáforos

Aunque los mecanismos preferidos para la sincronización de threads son los mutex y las variables condicionales, el perfil del "tostador" incluye los semáforos contadores definidos en el POSIX 1003.1b para aumentar la portabilidad de las aplicaciones. Los semáforos contadores se pueden usar tanto para sincronización de exclusión mutua, como para la sincronización de espera. Hay dos tipos de semáforos: con nombre y sin nombre. Los semáforos sin nombre son variables definidas en el espacio de la aplicación e identificados mediante un puntero, mientras que los semáforos con nombre residen en el espacio de datos del sistema operativo y están identificados por un nombre. La operación *sem_open*() inicializa un semáforo con nombre y traduce este nombre a un puntero que se usa luego para referirse al semáforo. Puesto que el número de semáforos está acotado, se puede preasignar un array para los semáforos durante la configuración del sistema. Luego, cuando el usuario solicita la inicialización de un semáforo con nombre, el nombre se asocia a uno de los semáforos no usados. La operación *sem_unlink*() libera el espacio del semáforo, que podrá ser reutilizado posteriormente.

4.5.4 Colas de Mensajes

Las colas de mensajes se incluyeron en el perfil del "tostador" porque representan un paradigma usual para el diseño de software de tiempo real. Mediante las colas de mensajes, los threads pueden intercambiarse mensajes con eventos o información. En las colas de mensajes POSIX cada mensaje tiene una prioridad asociada, lo que permite enviar mensajes con diferentes grados de urgencia. Al igual que para los semáforos, el número de colas de mensajes está acotado, por lo que podemos crear un array de registros de colas de mensajes con espacio para almacenar todos los datos de cada cola de mensajes, a excepción de los propios mensajes. El espacio requerido para los mensajes se determina en el instante en el que la aplicación crea la cola de mensajes, y por tanto puede ser alojado fácilmente en ese momento. Tanto el tamaño máximo de cada mensaje como el número máximo de mensajes son especificados por la aplicación, por lo que es sencillo implementar el gestor del espacio de mensajes con bloques de tamaño fijo y, por tanto, con tiempo de respuesta y tamaño predecibles.

4.6 Relojes y temporizadores

El único reloj requerido por el estándar POSIX.1b es el reloj de tiempo real (CLOCK_-REALTIME). Cuando la aplicación solicita conocer la hora, se lee el reloj y se devuelve su valor. Los temporizadores son creados por la aplicación usando registros de temporizador prealojados, ya que el número máximo de temporizadores es limitado. El instante en el que el temporizador expira —y en el que por tanto hay que generar un evento (p.e. una señal)— se añade a la cola de eventos temporales en el orden apropiado. Durante la gestión de la interrupción asociada a este evento, si el temporizador se había programado para expirar periódicamente, un nuevo evento temporal se inserta en la cola de eventos temporales en el lugar adecuado.

4.7 Entrada/salida

Los dispositivos de entrada/salida disponibles (y sus nombres) se definen al configurar el sistema operativo y son estáticos. Por tanto, la ejecución de cualquiera de las operaciones de entrada/salida (*open*(), *close*(), *read*(), *write*()) implica el acceso directo a los dispositivos. Los drivers de dispositivo requeridos se configuran igualmente de forma estática. Las operaciones de control sobre el dispositivo se pueden realizar por la aplicación mediante llamadas a las funciones *read*() o *write*(). Alternativamente se puede usar una función no estándar, pero común en muchas implementaciones, *ioctl*(), para el control de los dispositivos con un mayor nivel de flexibilidad.

5. Características de rendimiento del sistema operativo mínimo

Sin implementar el sistema operativo descrito en la sección 4 sobre una plataforma hardware concreta, no es posible definir de forma precisa las prestaciones del sistema operativo, pero podemos considerar el diseño descrito y hacer algunas consideraciones acerca de su rendimiento. Hay tres consideraciones principales respecto a nuestro sistema operativo para sistemas empotrados pequeños:

5.1 Tamaño de memoria

En primer lugar, el tamaño del sistema operativo debe de ser pequeño. Aunque el hardware actual proporciona memorias de tamaño mucho mayor que antiguamente, en sistemas de bajo coste es todavía de gran importancia minimizar el tamaño de memoria asignado al sistema operativo. Como regla general el tamaño de un sistema operativo para el tipo de aplicaciones para las que está pensado el perfil de sistema de tiempo real mínimo debería de estar comprendido entre 100 y 500 KB o menos.

Diseños como el descrito en la sección 4 se han construido para muchas aplicaciones de tiempo real durante años. No es inusual que estos sistemas operativos tengan tamaños de código de 50 KB o menos, sin incluir el espacio de datos. El espacio para los datos se determina en función de las necesidades de la aplicación en términos del número de tareas, mutex, etc., pero frecuentemente no supera los 50 KB. Por tanto, un sistema operativo de este tipo se puede construir en un espacio total de unos 100 KB.

5.2 Duración de los diferentes servicios

En segundo lugar, el número de instrucciones ejecutadas para cada interrupción y para cada llamada al sistema debe ser pequeño y predecible. Debe de ser posible definir el tiempo de ejecución de peor caso de cada servicio y de cada interrupción de una forma estrictamente acotada.

Considerando el diseño descrito arriba para nuestro sistema operativo mínimo, podemos observar que la mayoría de las operaciones, especialmente aquellas ejecutadas con mayor frecuencia, son bien $\boldsymbol{O}(1)$ u $\boldsymbol{O}(n)$, donde n es el número de threads activos o el número de threads bloqueados, y por tanto siempre acotado. El bloqueo de los mutex y la espera a variables condicionales son también $\boldsymbol{O}(n)$ si el mutex está tomado, y $\boldsymbol{O}(1)$ si está libre. La operación de liberación de un mutex es siempre $\boldsymbol{O}(1)$ para mutex con el protocolo de herencia de prioridad, y $\boldsymbol{O}(n)$ para los mutex con el protocolo de protección de prioridad.

5.3 Planificabilidad

En tercer lugar, todas las operaciones ejecutadas por el sistema operativo deben de ser planificadas de forma acorde con las prioridades de la aplicación. Esto implica que el sistema operativo deberá de enmascarar las interrupciones en contadas ocasiones, y sólo por intervalos de tiempo de duración extremadamente corta. Las operaciones de bloqueo asociadas al mantenimiento de las colas del sistema operativo también deben de ser cortas y acotadas.

Usando una cola de prioridad para la cola de threads activos, y minimizando los tiempos de bloqueo de las colas y de inhibición de las interrupciones, se consigue un sistema operativo que puede ser usado tanto para aplicaciones de tiempo real estricto como no estricto, manteniendo siempre la máxima planificabilidad posible.

6. Conclusiones

A pesar de tener sus raíces en las interfaces de sistemas UNIX de complejidad elevada, el Perfil de Sistema de Tiempo Real Mínimo del POSIX 1003.13 proporciona algo único hasta el momento: una interfaz estándar para programas de aplicación que es capaz de proporcionar un nivel de portabilidad nunca conseguido anteriormente para sistemas empotrados pequeños. Aunque nunca se puede pretender que una aplicación de tiempo real pueda ser portada sin cambios de una plataforma a otra, las aplicaciones construidas de acuerdo a este estándar requerirán cambios sólo para las interfaces asociadas directamente al hardware y a los drivers de dispositivos, pero no requerirán cambios en todo lo que respecta a su concurrencia, sincronización, temporización, y funciones de acceso a los recursos del sistema.

Este alto nivel de portabilidad se produce sin requerir una sobrecarga excesiva, tiempos impredecibles, o utilizaciones de memoria altas. Las funciones se han diseñado cuidadosamente para ser implementables de forma eficiente, y con una pérdida mínima de planificabilidad. A medida que las implementaciones comerciales vayan apareciendo, es razonable esperar que las aplicaciones construidas para cumplir este estándar se convertirán en una parte dominante del mercado de las aplicaciones pequeñas de tiempo real empotradas.

7. Referencias y lecturas aconsejadas

Burns, A. and Wellings A., 1996. “Real-Time Systems and Programming Languages”. Second Edition. Addison-Wesley.

Gallmeister, B.O., 1995. “*POSIX.4: Programming for the Real World*”. O´Reilly Associates, Inc.

IEEE Standards Project P1003.1j, 1995 “*Draft Standard for Information Technology -Portable Operating System Interface (POSIX)- Part 1: Realtime System API Extension*”. Draft 5. The Institute of Electrical and Electronics Engineers.

IEEE Standards Project P1003.13, 1997. “*Draft Standard for Information Technology -Standardized Application Environment Profile- POSIX Realtime Application Support (AEP)*”. Draft 8. The Institute of Electrical and Electronics Engineers.

IEEE Standards Project P1003.1d, 1997. “*Draft Standard for Information Technology -Portable Operating System Interface (POSIX)- Part 1: Realtime System API Extension*”. Draft 10.0. The Institute of Electrical and Electronics Engineers.

ISO/IEC Standard 9945-1:1996. “*Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]*”. Institute of Electrical and electronic Engineers.

Klein, M.H, Ralya, T., Pollak, B., Obenza, R. and González Harbour. M, 1993. “*A Practitioner's Handbook for Real-Time Analysis*”. Kluwer Academic Pub.

Locke, C. D., 1992, "Fixed Priority vs. Cyclic Executive: A Software Engineering Perspective", *Real-Time Systems Journal*, Kluwer Academic Pub.

Nichols, B., Buttlar, D. and Proulx Farrell, J, 1996. "*Pthreads Programming*". O’Reilly & Associates, Inc.

PASC, 1997. "*POSIX status report*". `http://www.pasc.org/standing/sd11.html`

Project UDI, 1995. “*Uniform Driver Interface. Environment Specification*”. http://www3.sco.com/Products/layered/develop/devspecs/udi/

Sha, L., Rajkumar, R. and Lehoczky, J.P., 1990. “Priority Inheritance Protocols: An approach to Real-Time Synchronization”. *IEEE Trans. on Computers*, September 1990.