

Tutorial de Real Time Linux

Ismael Ripoll

Table of Contents

1 Tutorial del API de RTLinux	1
1.1 Consejos para leer este documento.....	1
1.2 Contenido.....	1
1.3 Documentación adicional.....	1
1.4 Usabilidad de esta documentación.....	2
2 Introducción a RTLinux	3
2.1 Historia y evolución	3
2.2 Relación entre RTLinux y Linux	3
2.3 Características	3
2.4 Operación de un S.O. clásico	3
2.5 Estándares UNIX.....	4
2.6 POSIX Threads (IEEE 1003.1b)	4
2.7 El papel de las interrupciones.....	5
2.8 Arquitectura de RTLinux	5
2.9 Módulos del núcleo	6
2.10 Referencias	6
3 Módulos del núcleo cargables	9
3.1 Introducción	9
3.2 Módulos cargables.....	9
3.3 Módulo "Hello World" (mhello.c)	9
3.4 Compilar módulos	10
3.5 Ejecutar módulos.....	10
3.6 Parámetros	10
3.7 Más sobre módulos.....	10
4 Creación y gestión de threads.....	13
4.1 Listado de funciones.....	13
4.2 La primera rt_task	13
4.3 Detalles de implementación	13
4.4 Creación y destrucción de threads.....	14
4.5 Atributos.....	14
4.6 PThreads periódicos, no portable	14
4.7 Cancelación	14
4.8 Ejemplo de cancelación.....	15
4.9 Pila de cancelación	15
4.10 Control de ejecución.....	16
4.11	16
5 Señales e Interrupciones.....	17
5.1 Introducción	17
5.2 Señales.....	17
5.3 Envío de señales	18
5.4 Listado de señales.....	18
5.5 Esquema global de gestión de interrupciones	18
5.6 Habilitar/inhabilitar interrupciones	18
5.7 Enmascarar interrupciones	19
5.8 Interrupciones hardware.....	19
5.9 Interrupciones software	20
6 Gestión del tiempo	21
6.1 Introducción	21
6.2 Visión global	21
6.3 Relojes	21
6.4 Soporte hardware.....	22
6.5 Estructuras de datos.....	23
6.6 Listado de funciones.....	23
6.7 Consultar el reloj	23
6.8 Esperas	23
6.9 Miscelánea de funciones	24
6.10 Modo de operación.....	24
6.11 Interrupción de reloj	24
6.12 Operar con tiempos	25
7 Sincronización: Mutex, Variables condición y Semáforos.....	27
7.1 Introducción	27

Table of Contents

7.2	Inversión de prioridad	27
7.3	Utilización	27
7.4	Listado de funciones.....	28
7.5	Semáforos.....	28
7.6	Atributos de semáforos.....	28
7.7	.._SPINLOCK_NP un caso especial	29
7.8	Techos de prioridad dinámicos	29
7.9	Variables condición.....	29
7.10	Semáforos generales.....	30
7.11	Funciones	31
8	Dispositivos POSIX y acceso a la Entrada/Salida.....	33
8.1	Introducción	33
8.2	Estructura global	33
8.3	/dev/mem.....	34
8.4	Registrar nuevos dispositivos.....	35
8.5	Programación de la E/S	35
8.6	Acceso a la memoria física.....	36
8.7	Programación de puertos desde procesos Linux	36
9	FIFO.....	37
9.1	Introducción	37
9.2	Programación desde procesos Linux.....	37
9.3	Copia de variables por FIFO	38
9.4	Listado de funciones.....	38
9.5	Crear FIFOS	38
9.6	Leer y escribir en fifos	38
9.7	Control de estado.....	38
9.8	Manejadores de FIFOS.....	39
9.9	API POSIX.4.....	39
9.10	open y close.....	39
9.11	Read y write	39
9.12	Colas de mensajes	39
9.13	40
9.14	40
10	Memoria compartida.....	43
10.1	Introducción	43
10.2	Memoria física contigua.....	43
10.3	Listado de funciones.....	43
10.4	Uso de mbuff desde Linux	44
10.5	Crear y liberar memoria	44
10.6	Utilización de mbuff desde RLinux	44
10.7	Memoria alta no usada	44
10.8	Memoria alta desde RTLinux.....	45
10.9	Memoria alta desde Linux.....	45
10.10	Bigphysarea.....	45
10.11	46
11	Herramientas de depuración	47
11.1	Introducción	47
11.2	Imprimir directo a consola	47
11.3	Imprimir mediante printk	47
12	FAQ.....	51
13	Licencia.....	53

1 Tutorial del API de RTLinux

alertwidth()

1.1 Consejos para leer este documento



Recuerda que estas páginas están especialmente diseñadas para *Konqueror*, *Netscape 6*, *Mozill*, *Opera*, etc. sobre *Linux*.

Desde esta documentación se hace referencia a los ficheros fuente del propio RTLinux. Para que los enlaces de la documentación funcionen correctamente es necesario que RTLinux esté instalado en el directorio [/usr/src/rtl](#) de la máquina local.









También es conveniente tener instalados los fuentes del núcleo de Linux en su lugar habitual: [/usr/src/linux](#).


1.2 Contenido

1. Conceptos generales

1. [Introducción](#) ()
2. [Módulos](#) ()

2. API de RT-Linux.

1. [Creación y gestión de pthreads](#) ()
2. [Señales e interrupciones](#) ()
3. [Gestión del tiempo](#) ()
4. [Sincronización](#) ()
5. [Dispositivos POSIX y acceso a la Entrada/Salida](#) ()
6. [Comunicación mediante FIFOs](#) ()
7. [Memoria compartida](#) ()
8. [Depuración](#) ()
9. [Mini-FAQ](#)

También puedes bajarte todo el tutorial en un solo fichero () de 350Kb, generado con [htmldoc](#).

Puedes encontrar la última versión de este documento en: <http://bernia.disca.upv.es/rtportal>

1.3 Documentación adicional

1.3.1 Linux

- Linux Kernel Module Programming Guide www.linuxhq.com/guides/LKMPG/ (versión [local](#))

1.3.2 RT Linux

- Antes de comenzar la implementación del API de PThreads en RTLinux, Victor Yodaiken esbozó las principales líneas de diseño en el documento [design.pdf](#).
- Documentos [Getting started with RT-Linux](#) y [FAQ](#) de Michael Barabanov, que acompañan los fuentes de RTLinux.
- Módulo de comunicaciones serie (RS232) para RTLinux: [rt_com](#).
- Artículo que describe una aplicación de control ejemplo desarrollada con RTLinux: [nerdcontrol](#).
- [Real Time and Embedded HOWTO](#), Herman Bruyninckx.

1.3.3 Hardware del PC

- Manuales de Intel sobre la arquitectura Pentium®:
 - ♦ [Software Developer Manual Volume 1: Basic Architecture](#)
 - ♦ [Software Developer Manual Volume 2: Instruction Set Reference Manual](#)
 - ♦ [Software Developer Manual Volume 3: System Programming Guide](#)


- Prontuario sobre el hardware del PC: [hard0001](#).
- Mucha información sobre hardware del PC: www.us-e panorama.net

1.3.4 Estándares POSIX

- *"Programming for the Real World POSIX 1004"* de Bill O. Gallmeister. Ed: O'Really Associates, Inc..
- *Multithreaded Programming with Pthreads* de Bil Lewis y Daniel J. Berg. Ed: Sun Microsystems.
- El tutorial en línea *"Getting Started With POSIX Threads"* de Thomas Wagner and Don Towsley.
- OpenGroup www.opengroup.org. Propietario de la marca UNIX e impulsor de "Single UNIX Specification". En su web se pueden consultar todas las hojas de manual.

1.4 Usabilidad de esta documentación

Las presentaciones en transparencias se generan automáticamente a partir del fichero HTML que se utiliza para imprimir.

Se ha tratado de seguir los estándares  W3C lo más fielmente posible para obtener una presentación compatible con todos los navegadores del mercado. Las transparencias son compatibles con: HTML 4, DOM 2, ECMAScript 1.2 y CSS 1. Es posible ver correctamente las presentaciones con los siguientes navegadores: [Netscape 4.x](#), [Netscape 6](#), [Mozilla](#), [Opera](#), [Konqueror](#) y [SkipStone](#).

Desgraciadamente, Internet Explorer ® no sigue los estándares. Si bien Microsoft® ha participado en el diseño y estandarización del *DOM* (Document Object Model), sus productos no lo utilizan.



Linux is a trademark of Linus Torvalds.
RTLinux is a trademark of VJY Associates, LLC. FSMLabs is a service of VJY Associates, LLC of New Mexico.
© Jose Ismael Ripoll Ripoll (disca), Abril 2000 [Opencontent License](#)



2 Introducción a RTLinux

© Ismael Ripoll
iripoll@disca.upv.es
<http://bernia.disca.upv.es/~iripoll>

Permission is granted to copy, distribute and/or modify this document under the terms of the [OpenContent License](#).

2.1 Historia y evolución

- RTLinux nació del trabajo de Michael Barabanov y Victor Yodaiken en *New Mexico Tech*. Hoy en día continúan activamente con su desarrollo desde su propia empresa ([FSM Labs](#)) desde la que ofrecen soporte técnico.
- RTLinux se distribuye bajo la "GNU Public License". Recientemente Victor Yodaiken ha patentado la original arquitectura en la que se basa RTLinux.
- Actualmente funciona sobre arquitecturas PowerPC, i386, y está en desarrollo la versión para Alpha.
- A partir del código de Yodaiken, se está desarrollando otro proyecto liderado por P. Mantegazza llamado: "Real Time Application Interface" [RTAI](#)
- Esta documentación describe únicamente la principal versión de RTLinux (desarrollada por [FMS Labs](#)).
- Las primeras versiones de RTLinux ofrecían un API muy reducido sin tener en cuenta ninguno de los estándares de tiempo real: POSIX Real-Time extensions, PThreads, etc.
- A partir de la versión 2.0 Victor Yodaiken decide reconvertir el API original a otro que fuera "compatible" con el API de POSIX Threads. El documento [design](#) explica las líneas generales de la adaptación al estándar.
- Existe una versión para multiprocesadores, con la posibilidad de asignar tareas a procesadores.

2.2 Relación entre RTLinux y Linux

- Es importante no confundir la versión de RTLinux con la versión del núcleo de Linux.
 - ♦ RTLinux no es código independiente. Esto es, no es una nueva versión de Linux.
 - ♦ Parte de la distribución de RTLinux es una "parche" sobre el código de Linux. Y otra parte son módulos cargables.
 - ♦ Cada versión de RTLinux está diseñada para funcionar sobre una versión de Linux. Por ejemplo la versión 3 de RTLinux necesita linux-2.3.48 o superior.

2.3 Características

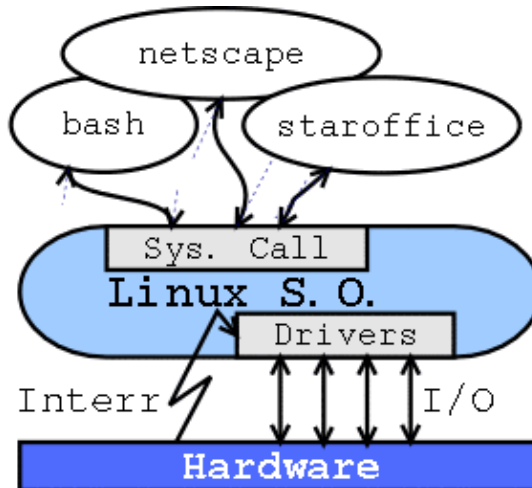
- Sistema operativo de tiempo real estricto.
- Extensiones para entorno multiprocesador SMP (x86).
- API "próximo" al de POSIX threads. Planificador expulsivo por prioridades fijas, señales, sistema de archivos POSIX (open, close, etc.) semáforos y variables condición.
- Depuración de código mediante GDB (GNU Debugger).
- Soporte para arquitecturas x86 y PPC.
- Acceso directo al hardware (puertos e interrupciones).
- Comunicación con procesos linux mediante memoria compartida y "tubos".
- Estructura modular para crear sistemas pequeños.
- Eficiente gestión de tiempos. En el peor caso se dispone de una resolución próxima al microsegundo (para un i486).
- Facilidades para incorporar nuevos componentes: relojes, dispositivos de E/S y planificadores.

2.4 Operación de un S.O. clásico

- Desde un punto de vista clásico, el núcleo del S.O. es un programa (conjunto de funciones y estructuras de datos) que gestiona los recursos de la máquina.
- El núcleo se ejecuta estando el microprocesador en *modo supervisor*, esto es, el micro permite la ejecución de todas las instrucciones máquina, es posible acceder a todas las direcciones de memoria, programar los periféricos y capturar las interrupciones. Utilizando la terminología de Intel diríamos que el núcleo se ejecuta

en "Ring level 0".

- Los procesos "normales" tienen prohibido el uso de ciertas instrucciones consideradas privilegiadas. Un proceso "normal" no puede acceder a los puertos de E/S ni capturar interrupciones. El procesador está en *modo usuario*: "Ring level 3".
- En cualquier sistema operativo (S.O.) las aplicaciones se ejecutan bajo la supervisión del S.O. y sin la capacidad de acceder al hardware de la máquina.



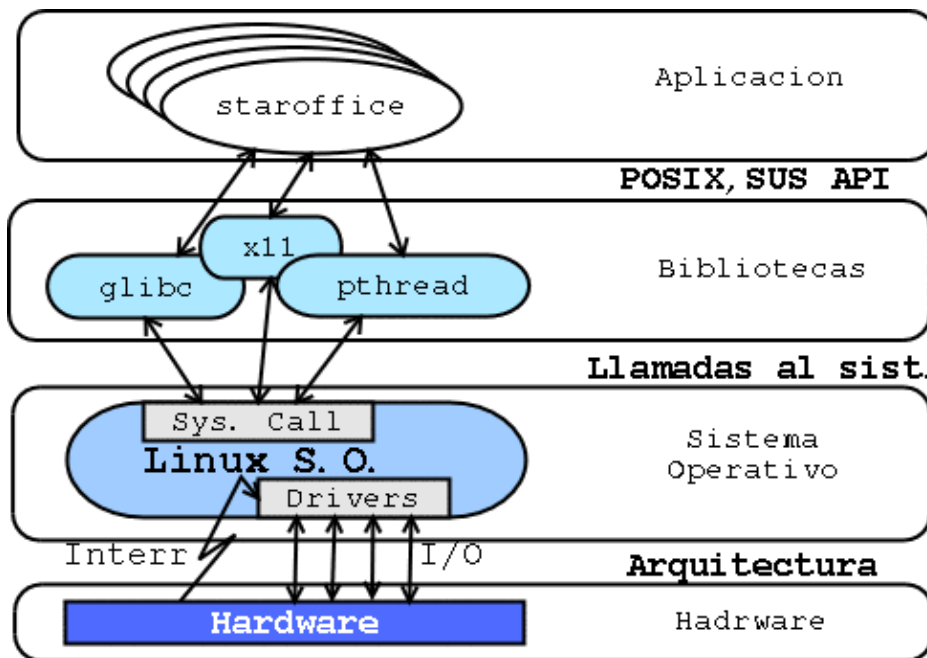
2.5 Estándares UNIX

- La historia de UNIX está jalonada por una serie de desarrollos más o menos propietarios, seguidos por intentos de unificación o estandarización.
- En cada momento, la empresa con mayor implantación ha sido la que ha "impuesto" sus nuevos desarrollos a UNIX, dejando anticuadas las versiones UNIX de la competencia.
- En los últimos años (finales de 1980) han aparecido varios estándares para definir el API de UNIX: POSIX, SVIDIII, X/Open, etc.
- Actualmente, el consorcio [OpenGroup](#) es el propietario de la marca UNIX. Este grupo está intentando unificar todos los estándares de UNIX bajo el nombre de "*Single UNIX Specification*" (SUS).
- Los estándares UNIX se centran en la definición del API del sistema utilizando el lenguaje "C". El objetivo es que un mismo código fuente pueda ser compilado en dos sistemas POSIX distintos sin necesidad de modificar el código.
- En UNIX no tiene sentido hablar de compatibilidad de código ejecutable.
- Tanto las especificaciones de tiempo real como la de threads recogidas en el estándar SUS son las definidas en el estándar POSIX 1003.1a y 1003.1b.
- Es posible acceder a las especificaciones en formato HTML de PThread desde la página [The Single UNIX Specification, Version 2 – 6 Vol Set for UNIX 98](#).

2.6 POSIX Threads (IEEE 1003.1b)

- Inicialmente Xavier Leroy desarrolló una biblioteca que ofrecía el API de PThreads para Linux. Esta implementación se conocía como LinuxThreads.
- Actualmente, LinuxThreads ha pasado a formar parte de la distribución estándar de la biblioteca de "C" de GNU, conocida como la "glibc2".
- Esta implementación de PThreads se basa en la llamada al sistema `clone(2)` que Linux ofrece para crear procesos.
- Utilizando esta biblioteca se pueden desarrollar sistemas de tiempo real blando, ya que las tareas de tiempo real (threads) son ejecutadas sobre un núcleo que no soporta tiempo real estricto.

La biblioteca PThread de glib2 no modifica el núcleo del S.O., sino que implementa el API de PThredas (1003.1b) sobre las llamadas al sistema existentes.



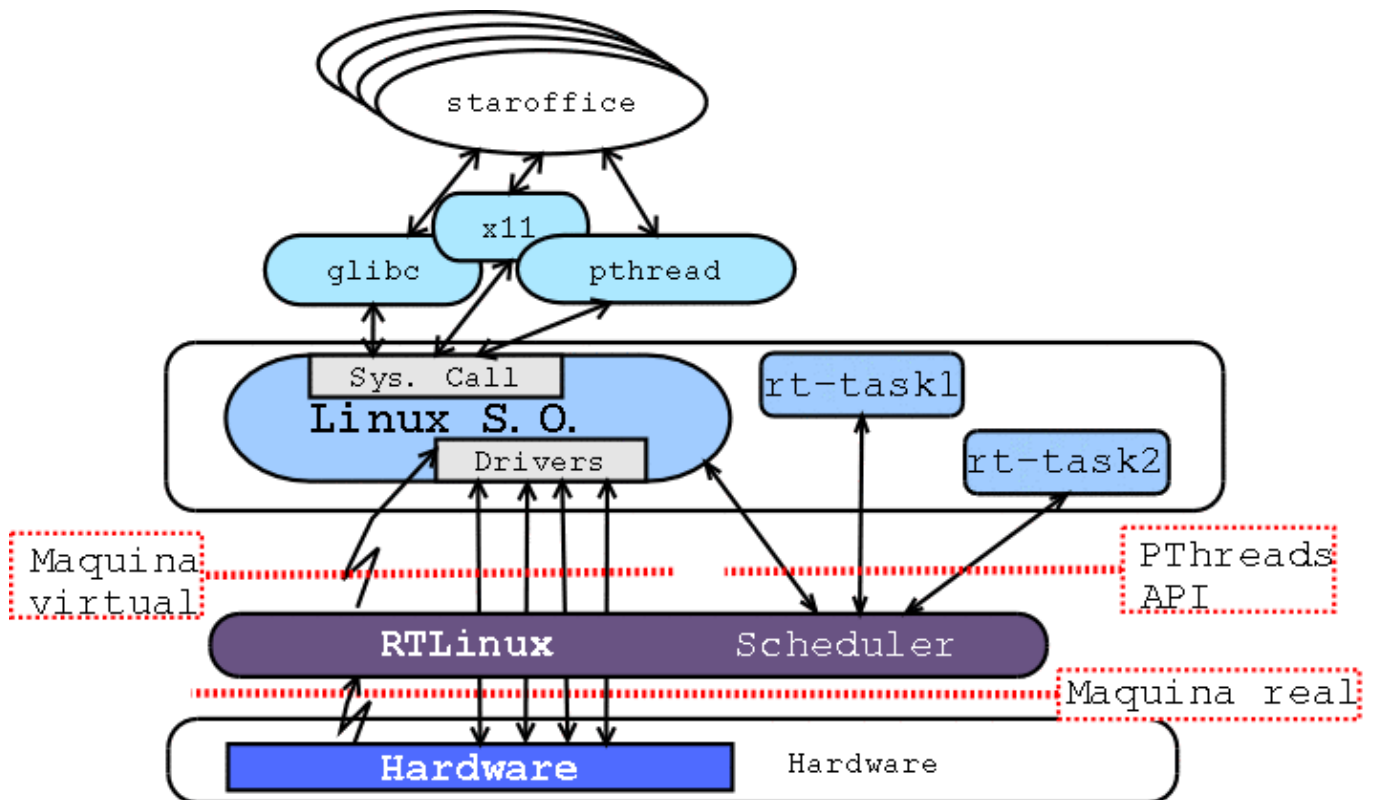
2.7 El papel de las interrupciones

Si sólo hay un procesador ¿Cómo puede el núcleo controlar el funcionamiento del resto de procesos?

- Cuando un proceso "normal" está en ejecución el S.O. está parado. Suponiendo un sistema monoprocesador, en un instante dado sólo puede haber un proceso ejecutándose, o bien un proceso "normal" o el núcleo, pero no los dos a la vez.
- Una vez el ordenador ha arrancado, la única forma de tomar el control el núcleo es mediante las interrupciones.
- Cada vez que se produce una interrupción el microprocesador deja lo que está haciendo y pasa a ejecutar la rutina de tratamiento de interrupción. Estas rutina (función) pertenece al núcleo.

2.8 Arquitectura de RTLinux

- A diferencia de otras aproximaciones para diseñar un S.O. de tiempo real, RTLinux no añade nuevas llamadas al sistema ni modifica ninguna de las ya existentes. Tampoco es una biblioteca para el programador.
- RTLinux se sitúa entre el hardware y el propio sistema operativo, creando una máquina virtual para que Linux pueda seguir funcionando.
- RTLinux toma el control de todas las interrupciones, e implementa un gestor de interrupciones por software.
- Las tareas RT-Linux se ejecutan utilizando el Run Time Support (RTS) de RTLinux.



- Las tareas de tiempo real (rt-task):

- ♦ Comparten el mismo espacio de memoria que el núcleo, por lo que pueden acceder a todas las variables y funciones de éste, aunque se podrían producir interbloqueos o condiciones de carrera.
- ♦ No pueden hacer uso de las llamadas al sistema de Linux.
- ♦ Se ejecutan en modo supervisor, esto es, pueden ejecutar cualquier instrucción de procesador y tienen acceso a todos los puertos de entrada/salida.
- ♦ Las páginas de memoria de datos y programa no pueden sufrir intercambio con disco (Swap-out).

2.9 Módulos del núcleo

- Para poner en ejecución una rt-task se tiene que utilizar el sistema de módulos cargables de Linux.
- Los módulos son "trozos de sistema operativo" que se pueden insertar y extraer en tiempo de ejecución.
- Un módulo es un fichero objeto, obtenido a partir de un fuente en "C" compilado pero no enlazado (linkado).
- Existen varias guías que explican la programación de módulos, una de ellas es la de Ori Pomerantz: [Linux Kernel Module Programming Guide](#)
- Estudiaremos la programación con módulos en el tema del API.

2.10 Referencias

- *Multithreaded Programming with Pthreads* de Bil Lewis y Daniel J. Berg. Ed: Sun Microsystems.
- Documento *"Getting started with RT-Linux"* de Michael Barabanov, que acompaña los fuentes de RTLinux.
- El libro *"Programming for the Real World POSIX 1004"*.
- El tutorial en línea *"Getting Started With POSIX Threads"* de Thomas Wagner and Don Towsley.
- OpenGroup www.opengroup.org. Propietario de la marca UNIX e impulsor de "Single UNIX Specification".
- Linux Kernel Module Programming Guide www.linuxhq.com/guides/LKMPG/



3 Módulos del núcleo cargables

© Ismael Ripoll
iripoll@disca.upv.es
<http://bernia.disca.upv.es/~iripoll>

Permission is granted to copy, distribute and/or modify this document under the terms of the [OpenContent License](#).

3.1 Introducción

- Las `rtl-tasks` pueden acceder a todas las variables y funciones del núcleo de Linux. Pero es poco recomendable ya que se pueden producir interbloqueos o condiciones de carrera.
- Desde las funciones de inicialización y finalización (`init_module` y `cleanup_module`) se puede hacer uso de todos los recursos del núcleo de forma segura. Estas funciones no son de tiempo real.
- A efectos prácticos, el núcleo de Linux se puede considerar como otra tarea de tiempo real (`rtl-task`) pero que es planificada con la mínima prioridad.

3.2 Módulos cargables

- Un módulo es un fichero objeto que se puede "enlazar" y "des-enlazar" en el núcleo de Linux en tiempo de ejecución.
- Con los módulos se agiliza enormemente el desarrollo de software crítico ya que no es necesario crear un nuevo núcleo y rearrancar la máquina cada vez que hacemos una prueba.
- Un módulo es un programa "C" sin función `main()`, y que obligatoriamente ha de tener las funciones `init_module` y `cleanup_module`.
- Cada módulo se compila para una versión de núcleo concreta y normalmente sólo se puede cargar sobre esta versión, aunque es posible forzar a cargar un módulo antiguo sobre un núcleo más moderno.
- Los módulos que el núcleo puede cargar suelen residir en el directorio `/lib/modules/[uname -r]/`.
- Los módulos se crean (como cualquier fichero objeto) con el compilador de "C" de GNU: `gcc`.
- Para trabajar con módulos se dispone de las siguientes utilidades del sistema:

<code>insmod:</code>	Instala en el núcleo un módulo.
<code>rmmod:</code>	Extrae del núcleo un módulo.
<code>modinfo:</code>	Muestra información sobre el módulo.
<code>modprobe:</code>	Automatiza/facilita la gestión de módulos.
<code>depmod:</code>	Determina las dependencias entre módulos.
<code>lsmod:</code>	Lista los módulos cargados.

3.3 Módulo "Hello World" ([mhelloworld.c](#))

```
#include <linux/module.h>
#include <linux/kernel.h>

static int datos=0;

MODULE_AUTHOR("Ismael Ripoll");
MODULE_DESCRIPTION("Modulo ejemplo Hello World");
MODULE_PARM(datos,"i");

int init_module(void) {
    printk("Hello World, parámetro %d \n", datos);
    return 0;
}

void cleanup_module(void) {
    printk("Bye World\n");
}
```

3.4 Compilar módulos

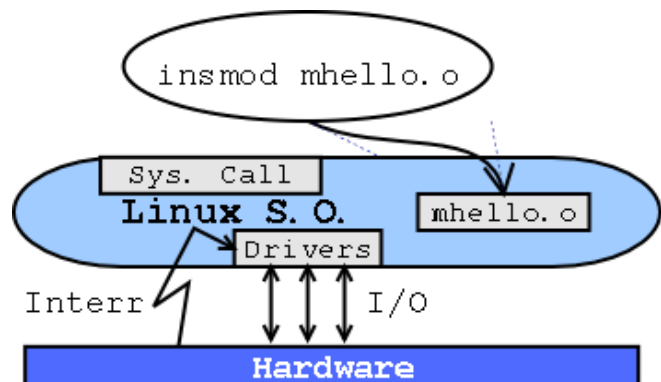
- Para compilar el `mhello.c` se tiene que utilizar la siguiente orden:

```
gcc -c -O2 -fomit-frame-pointer -DMODULE -D__KERNEL__ mhello.c
```

 - ♦ "-c": El compilador compila y ensambla el programa pero no lo enlaza, esto es, únicamente se generará el código objeto.
 - ♦ "-O2 -fomit-frame-pointer": genera código optimizado.
 - ♦ "-DMODULE -D__KERNEL__": define las macros `MODULE` y `__KERNEL__`, utilizadas por los ficheros de cabecera del núcleo para generar el código apropiado.
- Tras ejecutar esta orden obtendremos el fichero `mhello.o`

3.5 Ejecutar módulos

- Podemos insertar el módulo con la orden `insmod mhello.o`. Es necesario ser "root" para poder cargar y descargar módulos.



- Para ver todos los mensajes enviados por el núcleo (llamadas a `printk`) utilizamos la orden `dmesg`. Si estamos en consola de texto, entonces los mensajes aparecen también en pantalla conforme se generan.
- Para extraer el módulo `rmmod mhello`. Observa que en este caso no hay que poner la extensión ".o":

```
# insmod mhello.o datos=90; rmmod mhello; dmesg | tail -2
```

```
Hello World, parámetro 90
```

```
Bye World
```

3.6 Parámetros

- El método para pasar parámetros a un módulo durante la carga es el siguiente:
 1. La variable que recibirá el parámetro se ha de declarar de tipo `static`.
 2. Se convierte en parámetro mediante la macro `MODULE_PARM`.
 3. Cuando se realiza la carga del módulo, se añade el nombre del parámetro y el valor separados por el símbolo "=". En el código fuente del módulo:

```
static int datos=0;
MODULE_PARM(datos,"i");
```
- La orden `modinfo` informa de los parámetros que puede recibir un módulo sin necesidad de inspeccionar al código fuente.

```
# modinfo -p mhello.o
```

```
datos int
```

3.7 Más sobre módulos

- Una vez un módulo se ha cargado en el núcleo, todas sus funciones y variables públicas son accesibles desde nuevos módulos que se carguen después.
- La mayoría del API de RTLinux está dividido en varios módulos opcionales con el objetivo de poder diseñar sistemas de tiempo real lo más ajustados a las necesidades. *Si una aplicación no necesita semáforos entonces no cargamos el módulo que ofrece semáforos (`rtl_mutex.o`).*
- Por otra parte, antes de poder poner en marcha una aplicación de tiempo real necesitaremos cargar los módulos del API que nuestra aplicación vaya a necesitar.

- La orden `ksyms` lista todos los símbolos del núcleo, tanto los iniciales como los pertenecientes a módulos cargados.

4 Creación y gestión de threads

Ismael Ripoll
iripoll@disca.upv.es
<http://bernia.disca.upv.es/~iripoll>

Permission is granted to copy, distribute and/or modify this document under the terms of the [OpenContent License](#).

4.1 Listado de funciones

Creación/destrucción

`pthread_create`
`pthread_exit`

Atributos

`pthread_attr_init`
`pthread_attr_destroy`
`pthread_attr_setschedparam`
`pthread_attr_getschedparam`
`sched_get_priority_max`
`sched_get_priority_min`
`pthread_attr_setstacksize`

Cancelación

`pthread_cancel`
`pthread_setcancelstate`
`pthread_setcanceltype`
`pthread_testcancel`

Threads periódicos (no estándar)

`pthread_make_periodic_np`
`pthread_setperiod_np`
`pthread_wait_np`

Atributos (no estándar)

`pthread_attr_setfp_np`
`pthread_attr_getfp_np`
`pthread_attr_setcpu_np`
`pthread_attr_getcpu_np`

Control de ejecución (no estándar)

`pthread_suspend_np`
`pthread_wakeup_np`

4.2 La primera `rt_task`

[[hello.c](#)]

- En el directorio de [examples](#) de la distribución podemos encontrar este ejemplo de tarea de tiempo real mínima.
- Todas las funciones relacionadas con la creación y destrucción de threads están declaradas en el fichero de cabecera [include/rtl_sched.h](#).
- POSIX define tanto las funciones del API, como los nombres de los ficheros de cabecera donde están declaradas. Para conseguir la compatibilidad, RTLinux ha creado el directorio [include/posix](#) que contiene todos los ficheros de cabecera requeridos por POSIX. Estos ficheros de cabecera estándar suelen contener líneas para incluir los ficheros donde realmente se hacen las declaraciones.
- En el proceso de instalación de RTLinux se crea el fichero [rtl.mk](#), fichero que contiene todas las declaraciones necesarias para poder compilar los programas para RTLinux (módulos). Este fichero se tiene que incluir en el `Makefile` para poder compilar sin problemas nuestros programas.
- Para crear un programa para RTLinux es conveniente crear un fichero `Makefile` en el que incluya el fichero `rtl.mk` (mediante la directiva: `include /usr/src/rtl/rtl.mk`) y luego se incluyan las dependencias de nuestros fuentes.
- Nota: los `Makefiles` de los ejemplos no incluyen `rtl.mk` ya que estos son invocados desde el [Makefile](#) principal que es el que contiene todas las macros y directivas para compilar correctamente.

4.3 Detalles de implementación

- Aunque RTLinux implementa muchas funciones POSIX, en ocasiones estas funciones no se ajustan completamente a la semántica definida en el estándar.
- En algún caso, la función de RTLinux no implementará todas las opciones que define POSIX (Por ejemplo `sched_setscheduler` solo ofrece la política `SCHED_FIFO`) o sólo se podrán utilizar con ciertos parámetros. En estos casos, la llamada fallará con el correspondiente código de retorno. Otra situación más sutil, consiste en que ciertas funciones sólo se pueden ejecutar desde el núcleo de Linux, otras sólo por `rt-tasks` y otras se pueden ejecutar indistintamente.

- Las funciones marcadas como "**No RTL segura**" son aquellas que no pueden ejecutarse por tareas de tiempo real ni por manejadores de interrupciones de tiempo real.

4.4 Creación y destrucción de threads

`pthread_create(*thread, *attr, *start, *arg)`

Inicia la ejecución de un thread. Los atributos de creación por defecto (`attr=NULL`) son: `stack_size=8000`, `sched_priority=0`, `cpu=current`, `use_fp=FALSE`. El nuevo thread comienza su ejecución inmediatamente. **No RTL segura**.

`pthread_exit(retval)`

Termina la ejecución del thread que invoca esta llamada, retornando como valor de terminación `retval`.

`pthread_join(thread, retval)`

Suspende la ejecución del thread que invoca esta llamada hasta que `thread` termina su ejecución. En la implementación actual todos los threads son creados `PTHREAD_CREATE_JOINABLE` sin posibilidad de ser modificado.

4.5 Atributos

`pthread_attr_init(*attr), pthread_attr_init_destroy(*attr)`

Los atributos de creación de tareas se pasan como una estructura de tipo `pthread_attr_t`. El acceso a los campos de esta estructura se realiza mediante funciones simulando la forma de operar de la programación orientada a objetos.

`pthread_attr_[get/set]schedparam(*attr, *param)`

Establece la prioridad de la nueva `rtl-task`. Por ahora (V3) sólo está disponible la política `SCHED_FIFO`. El rango de prioridades válido se obtiene de funciones `sched_get_priority_min` y `sched_get_priority_max`. La mínima prioridad es 0 y la máxima es 100000 (realmente la máxima es `MAXINT`).

`pthread_attr_[set/get]fp_np(*attr, flag)`

Para reducir el tiempo de cambio de contexto por defecto RTLinux no salva a memoria el estado de coprocesador matemático. Si queremos que las `rtl-tasks` puedan utilizar el coprocesador entonces es necesario llamar a esta función (Ver ejemplo [fp](#)).

`pthread_attr_[set/get]cpu_np(*attr, cpu)`

Asigna el nuevo thread que se cree con `attr` al procesador `cpu` (si el sistema es SMP). La numeración de los procesadores es la que aparece en el fichero [/proc/cpuinfo](#).

`pthread_attr_setstacksize(*attr, size)`

Todos los threads comparten el mismo espacio de memoria, pero cada uno ha de tener su propia pila.

4.6 PThreads periódicos, no portable

`pthread_make_periodic_np(thread, start_time, period)`

Permite "despertar" un thread de forma periódica. El propio thread ha de suspenderse voluntariamente al final de cada activación y RTLinux lo reactiva cada periodo. La unidad de tiempo es el nanosegundo.

`pthread_setperiod_np(thread, *itime)`

Equivalente a la función anterior pero utilizando una estructura del tipo `itimerspec` para especificar el instante de inicio y el periodo.

`pthread_wait_np()`

Suspende la ejecución del thread que la invoca. Es necesario que el thread tenga un funcionamiento periódico. El propio programa [hello.c](#) es una tarea periódica.

4.7 Cancelación

`pthread_cancel(thread)`

La cancelación es el mecanismo por el cual un thread invoca la terminación de otro. El thread que invoca esta llamada no se suspende. El `thread` puede terminar o no en función de su estado de "cancelación".

`pthread_setcancelstate(state, *oldstate)`

Establece el estado de cancelación del thread que invoca esta llamada. Los posibles valores del `state` son: `PTHREAD_CANCEL_ENABLE` (estado por defecto) y `PTHREAD_CANCEL_DISABLE`. En el estado "DISABLE" este thread ignorará todas las peticiones de cancelación.

`pthread_setcanceltype(type, *oldtype)`

Una vez un thread acepta ser cancelado, la cancelación se puede realizar de dos formas:

1. Inmediatamente nada más otro thread llame a la función `pthread_cancel`.
2. Diferida, de forma que sólo en ciertos punto de la ejecución del thread se pueda terminar (ver `testcancel`). De esta forma se puede intentar dejar el sistema en un estado estable antes del finalizar (liberar semáforos, cerrar, ficheros, etc.). Opción por defecto.

Estos dos tipos se denominan respectivamente: `PTHREAD_CANCEL_ASYNCHRONOUS` y `PTHREAD_CANCEL_DEFERRED`.

`pthread_testcancel()`

Comprueba si hay pendiente alguna petición de cancelación y si es así termina el thread que la invoca. Sólo tiene sentido si el thread está en estado `PTHREAD_CANCEL_ENABLE` y tipo `PTHREAD_CANCEL_DEFERRED`.

- Si el estado de cancelación es `PTHREAD_CANCEL_DEFERRED` además forzando la llamada `pthread_testcancel()`, también se comprueba si hay una petición de terminación pendiente en las llamadas a las siguientes funciones:

<pre>pthread_suspend_np pthread_wait_np usleep pthread_cond_wait</pre>	<pre>pthread_cond_timedwait_hrt pthread_cond_timedwait pthread_testcancel pthread_join</pre>
--	--

- A la lista de funciones que chequean si se tiene que cancelar el thread habrá que añadir algunas otras funciones relacionadas con la gestión de señales cuando éstas estén implementadas.
- Un proceso con cancelación diferida que llame a una función que contenga alguna llamada a alguna de las funciones anteriores producirá la terminación del thread.
- Consejo: *Intentar evitar la cancelación diferida en la medida de lo posible.*

4.8 Ejemplo de cancelación

```
void *start_routine(void *arg){
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    pthread_make_periodic_np(pthread_self(),
                             gethrtime(), 500000000);

    while (1) {
        pthread_wait_np ();
        pthread_testcancel();

        // Sabemos que esta ejecución no se abortará.
        HAZ_MUCHO_TRABAJO(); // Sólo computo.

        pthread_testcancel();
    }
}
```

4.9 Pila de cancelación

- En POSIX existe la posibilidad de declarar funciones de cancelación que se ejecutan en caso de terminación (normal o anormal) del thread.
- Por ejemplo, el sistema puede quedar en un estado inestable si un thread es cancelado estando dentro de una sección crítica (ver semáforos). Podemos instalar una función de "limpieza" para liberar el semáforo.
- Es posible instalar tantas funciones de terminación como se quiera. Si se instalan dos funciones de terminación, éstas se ejecutan en orden FILO (primera en ser instalada última en ser ejecutada).
- Es obligatorio "desinstalar" todas las funciones de terminación en orden inverso a como se instalaron.
- El API de Pthread supone que las funciones de cancelación se gestionan mediante una pila de funciones de terminación.
- Para insertar una función en la pila de terminación se utiliza la función (realmente es una macro de "C"): `pthread_cleanup_push(func, arg)`, y para eliminarla `pthread_cleanup_pop(exec)`.

- Como se puede observar, las funciones se tienen que "desapilar" en orden inverso a como fueron instaladas. Y en el mismo nivel sintáctico.
- La propia implementación de variables condición hace uso de la pila de cancelación: [schedulers/rtl_mutex.c](#)

4.10 Control de ejecución

`pthread_suspend_np(thread)`

Envía la señal `RTL_SIGNAL_SUSPEND` a un thread, suspendiendo la ejecución del mismo.

Para volver a activarlo hay que enviarle la señal `RTL_SIGNAL_WAKEUP` o llamar a `pthread_wakeup_np`.

`pthread_wakeup_np(thread)`

Envía la señal `RTL_SIGNAL_WAKEUP` a un thread, reanudando su ejecución.

- Utilizando estas funciones junto con manejadores de interrupciones podemos asociar tareas con interrupciones.

4.11

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

pthread_t thread;

void * start_routine(void *arg)
{
    struct sched_param p;
    p . sched_priority = 1;
    pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);

    pthread_make_periodic_np (pthread_self(), gethrtime(), 500000000);

    while (1) {
        pthread_wait_np ();
        rtl_printf("I'm here; my arg is %x\n", (unsigned) arg);
    }
    return 0;
}

int init_module(void) {
    return pthread_create (&thread, NULL, start_routine, 0);
}

void cleanup_module(void) {
    pthread_cancel (thread);
    pthread_join (thread, NULL);
}
```

5 Señales e Interrupciones

Ismael Ripoll
iripoll@disca.upv.es
<http://bernia.disca.upv.es/~iripoll>

Permission is granted to copy, distribute and/or modify this document under the terms of the [OpenContent License](#).

5.1 Introducción

- En la programación clásica de UNIX, las señales se utilizan como un mecanismo de "comunicación" asíncrono. Si consideramos que un proceso es la abstracción de procesador, entonces las señales hacen el papel de las interrupciones.
- RTLinux hace un uso extensivo del concepto de señal/interrupción. Se utilizarán para comunicarse rt-tasks entre sí, modificar el estado de ejecución de rt-tasks, atender interrupciones hardware y disparar eventos en el núcleo de Linux.
- Actualmente toda la gestión de las interrupciones se realiza mediante funciones no estándar. El objetivo de los desarrolladores de RTLinux consisten en gestionar la interrupciones como señales. Todavía en fase de desarrollo.

5.2 Señales

- Al igual que sucede con las interrupciones, un proceso puede establecer a priori la forma en la que se atienden las señales. Cada señal puede estar en uno de los siguientes estados:
 - ♦ **Ignorada:** El proceso no recibe estas señales
 - ♦ **Capturada:** Cada vez que llega una señal se ejecuta una función manejadora asociada.
 - ♦ **Opción por defecto:** Dependiendo de la señal puede ser suspender o terminar la ejecución del proceso, ignorarla o causar un volcado de memoria (core).
 - ♦ **Bloqueada:** Las señales no se "entregan" al proceso hasta que se desbloqueen.
- Aunque una señal es un evento asíncrono a la ejecución de un thread, la recepción de la señal no se produce de forma inmediata a su envío. Si un thread de alta prioridad envía una señal a otro de menor prioridad, entonces el thread de menor prioridad sólo podrá atender (recibir) la señal cuando se ponga en ejecución. En otras palabras, cuando se le envía una señal a un thread, éste no toma el procesador inmediatamente para atender la señal, sino que la señal queda pendiente hasta que el thread toma el procesador.
- Las señales que representan interrupciones físicas sí que son atendidas inmediatamente, siempre y cuando la interrupción concreta no esté bloqueada ni inhabilitadas a nivel de procesador.
- En los sistemas operativos clásicos, el número de posibles señales es limitado, suele estar entre 16 y 64. RTLinux dispone de 1024 (RTL_SIGIRQMAX) distribuidas en varias categorías (interrupciones, predefinidas de sistema, entre pthreads y para Linux).
- El núcleo de RTLinux captura todas las interrupciones hardware y se las reenvía al núcleo de Linux.
 - ♦ Las señales serán el interfaz de RTLinux para el manejo y captura de interrupciones. Determinado rango de señales (de RTL_SIGIRQMIN a RTL_SIGIRQMIN + NR_IRQS) son señales que representarán interrupciones hardware.
 - ♦ Por ejemplo, instalar un manejador para la señal RTL_SIGIRQMIN equivale capturar la interrupción cero, esto es, la interrupción de reloj.
 - ♦ Puesto que el mecanismo de gestión de interrupciones mediante la interfaz de señales no está completamente implementado, presentará sólo la interfaz que ahora está disponible.
- En la distribución de RTLinux hay hojas de manual para todas las funciones no estándar de gestión de interrupciones.

5.3 Envío de señales

- Por ahora sólo está implementada la función para enviar señales. (La función para capturar señales, `sigaction`, también está en parte implementada).

`pthread_kill(thread, signo)`

Envía la señal `signo` a `thread`. La señal enviada no surtirá efecto hasta que el proceso destinatario se ponga en ejecución. La función retorna inmediatamente.

Si el thread no existe o el número de señal está fuera de rango, entonces devuelve un error (número distinto de cero).

5.4 Listado de señales

`RTL_SIGNAL_NULL`

Señal que no causa ninguna acción sobre el thread destino. Se utiliza para comprobar si el thread destino todavía está en ejecución, si es un thread válido.

`RTL_SIGNAL_SUSPEND`

Suspende la ejecución del thread que recibe esta señal.

`RTL_SIGNAL_WAKEUP`

Despierta el thread.

`RTL_SIGNAL_CANCEL`

Enviar esta señal a un thread es equivalente a invocar la función `pthread_cancel(thread)`.

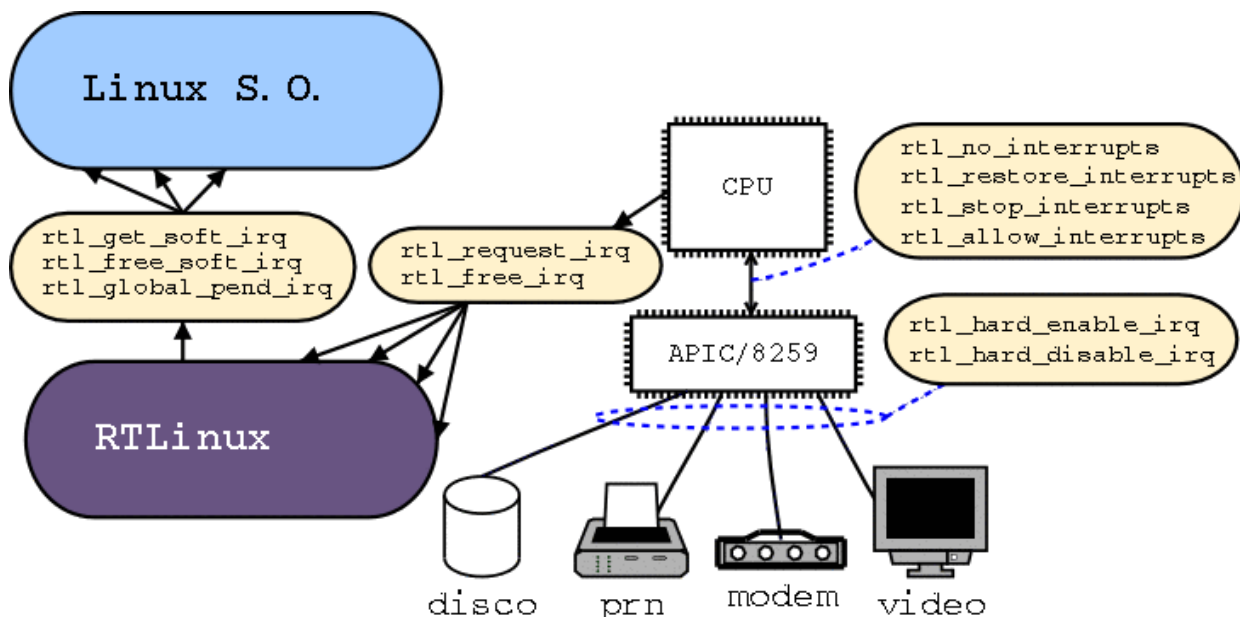
`RTL_LINUX_MIN_SIGNAL...`

Equivale a enviar una interrupción software al núcleo de Linux. `RTL_LINUX_MIN_SIGNAL` representa la interrupción 0. El thread destinatario de esta señal sólo puede ser el thread que representa el núcleo de Linux: `rtl_get_linux_thread(rtl_getcpuid())`.

`RTL_SIGNAL_KILL`

Termina la ejecución. *¡¡ La terminación del thread no se produce cuando se envía esta señal, sino que se produce cuando se recibe !!*

5.5 Esquema global de gestión de interrupciones

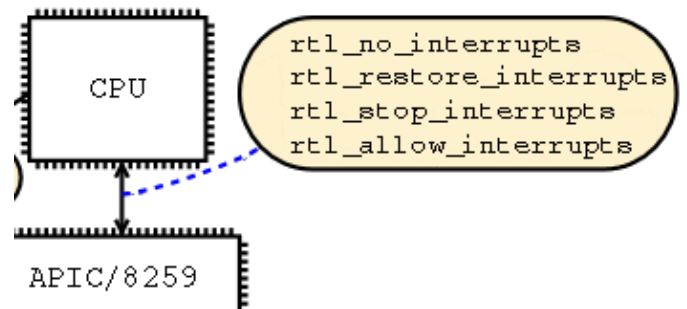


5.6 Habilitar/inhabilitar interrupciones

- **Macros** declaradas en `include/i386/rtl_sync.h`:

`rtl_no_interrupts(rtl_irqstate_t estado)`

Macro que salva los flags del procesador en la variable `estado` e inhabilita las interrupciones (`cli`). `estado` ha de ser una variable de tipo entero, y no un puntero a entero.



`rtl_restore_interrupts(rtl_irqstate_t estado)`

Carga los flags del procesador con el valor de `estado`. Se utiliza para reponer el estado de las interrupciones (flag IF), puesto que el resto de flags del procesador o bien no cambian (VM, IOPL, etc.), o bien su valor es indiferente (Carry, Zero, etc).

`rtl_stop_interrupts()`

Macro que se expande a la instrucción ensamblador `cld`, lo cual inhabilita las interrupciones.

`rtl_allow_interrupts()`

Macro que se expande a la instrucción ensamblador `sti` (SeT Interrupt flag), lo cual inhabilita las interrupciones.

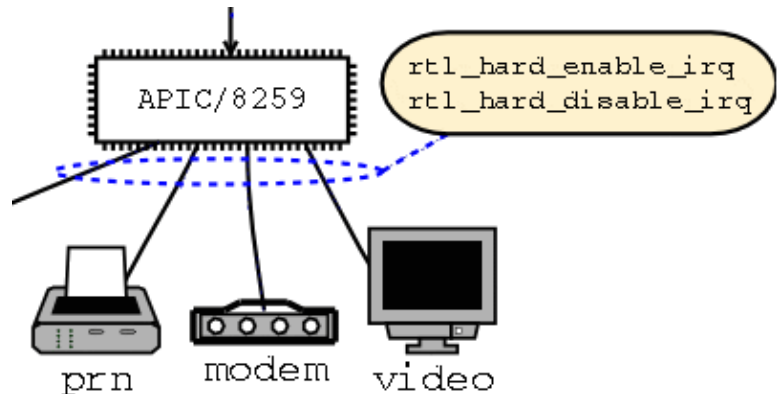
- Nota: todas estas operaciones únicamente tienen efecto sobre el procesador desde el que son ejecutadas, en caso de que el sistema sea multiprocesador.

5.7 Enmascarar interrupciones

- Enmascarar un señal consiste en bloquear temporalmente de forma individualizada su recepción por parte del procesador. El enmascaramiento se realiza en el chip encargado de gestionarlas (APIC o 8259).

`rtl_hard_enable_irq(irq)`

Desenmascara la interrupción `irq`. Lo que implica que se pueden recibir este tipo de interrupciones cuando las interrupciones estén habilitadas.



`rtl_hard_disable_irq(irq)`

Enmascara la interrupción `irq`. No se recibirán nuevas interrupciones de este tipo hasta que se vuelva a desenmascarar.

5.8 Interrupciones hardware

- Estas funciones capturan las interrupciones de la máquina real, y son ejecutadas en el espacio de ejecución de RTLinux sin ningún retraso.

- La interrupción número 0 es la de reloj, 1 la de teclado, etc. Funciones definidas en `include/rtl_core.h`:

`rtl_request_irq(irq, manejador)`

Registra la función `manejador` como el manejador de la interrupción `irq` y la desenmascara. Manejador ha de ser una función del tipo:

`unsigned (*manejador)(unsigned int irq, struct pt_regs *regs)`

El valor devuelto por el manejador no se utiliza.

Si ya hay un manejador para esa interrupción entonces devuelve un valor negativo.

Si se ha podido instalar el manejador, entonces se desenmascara la señal. ¿La información de la hoja de manual es incorrecta?

Cuando se produce la interrupción, la interrupción queda enmascarada hasta que se llame a `rtl_hard_enable_irq`. Esto impide que lleguen más interrupciones mientras el manejador no está

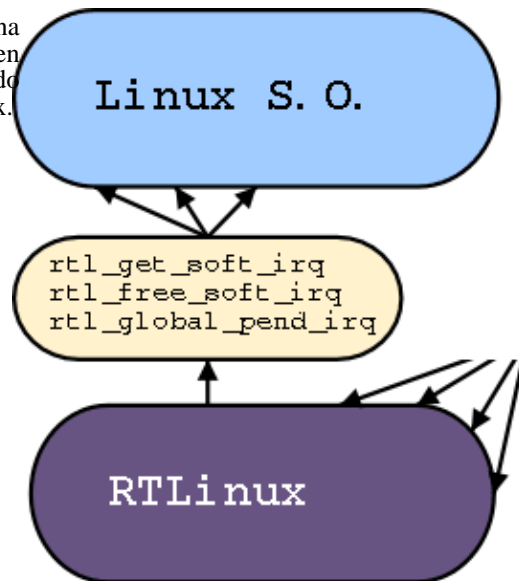
preparado para atenderlas.

```
rtl_free_irq(irq)
```

Libera el manejador de interrupción asociado a la interrupción `irq`.

5.9 Interrupciones software

- RTLinux ha virtualizado el hardware que utiliza Linux, de forma que Linux no tiene acceso directo al hardware real. Una de las partes virtualizadas son las interrupciones. Por defecto Linux recibe todas las interrupciones que RTLinux no tiene capturadas.
- Mediante el envío de señales/interrupciones es posible provocar la llegada de una interrupción al núcleo de Linux. Con este mecanismo es posible ejecutar funciones desde el entorno de ejecución de Linux. Por ejemplo, la función `rtl_printf()` guarda en un buffer la cadena de caracteres a imprimir y luego envía una interrupción al núcleo de Linux que despertará la rutina de servicio de interrupción encargada de llamar a la función `printk()`.
- RTLinux permite definir nuevas interrupciones de la máquina virtual Linux. Interrupciones que Linux "creará" que proceden de algún periférico hardware, pero que en realidad han sido originadas por alguna tarea de RTLinux.



- Funciones declaradas en el fichero de cabecera [include/rtl_core.h](#).

```
int rtl_get_soft_irq(manejador, devname)
```

Instala un manejador de interrupciones Linux. Las interrupciones se atienden en el espacio de ejecución del núcleo de Linux.

`devname` es una cadena de caracteres que sirve para identificar la interrupción. Este nombre aparecerá en el listado de interrupciones del sistema: [/proc/interrupts](#).

El valor devuelto es el número de interrupción libre que se ha asignado a este manejador. Es necesario destacar que en este tipo de interrupciones no es importante el número de interrupción que se utilice sino el hecho de que se ejecute el manejador.

```
rtl_free_soft_irq(irq)
```

Desinstala el manejador de interrupción. El valor de `irq` ha de ser el devuelto por la función `rtl_get_soft_irq` que instaló el manejador.

```
rtl_global_pend_irq(irq)
```

Genera una interrupción software que será entregada a Linux cuando éste se ponga en ejecución. La implementación es muy sencilla, sólo se marca como pendiente de entregar la interrupción `irq` en un mapa de bits.

6 Gestión del tiempo

Ismael Ripoll
iripoll@disca.upv.es
<http://bernia.disca.upv.es/~iripoll>

Permission is granted to copy, distribute and/or modify this document under the terms of the [OpenContent License](#).

6.1 Introducción

- Dos términos similares pero no iguales:

reloj (clock)

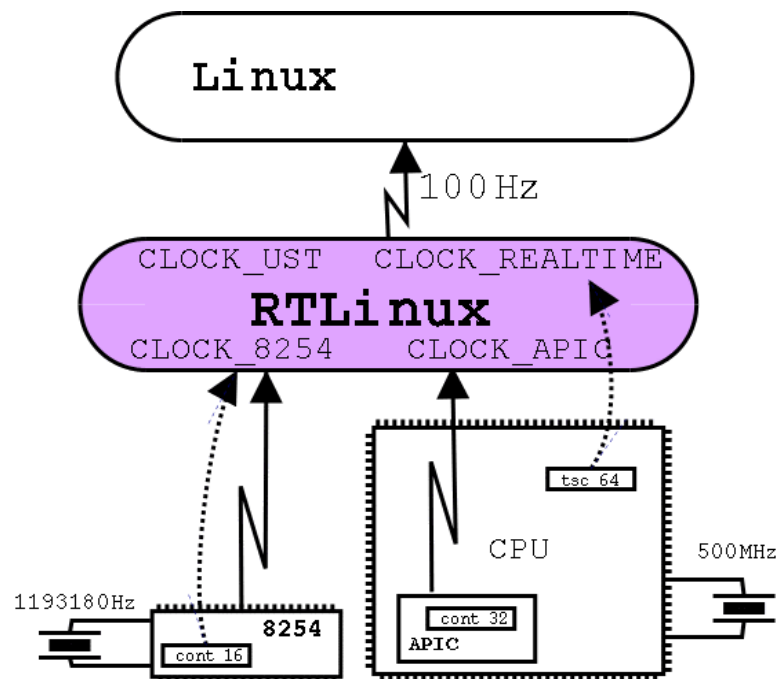
Dispositivo que registra el paso del tiempo. Las operaciones que acepta son: leer/escribir la hora y obtener sus características (resolución, precisión, etc.).

temporizador (timer)

Dispositivo que produce eventos relacionados con el paso del tiempo a los que se les puede asociar acciones. Las operaciones que acepta son: Programar periodo, asociar función, consultar estado del contador, etc.

- POSIX define operaciones para trabajar con relojes y temporizadores. RTLinux sólo implementa los relojes POSIX, pero es posible realizar temporizaciones, pero con funciones no estándar.

6.2 Visión global



6.3 Relojes

- El estándar POSIX especifica la posibilidad de disponer de varios relojes. Cualquier implementación ha de disponer al menos del reloj `CLOCK_REALTIME`.
- RTLinux dispone de tres relojes lógicos: `CLOCK_REALTIME`, `CLOCK_MONOTONIC` y `CLOCK_RTL_SCHED`; y dos físicos: `CLOCK_8254` y `CLOCK_APIC`.
- Durante la configuración previa a la compilación de RTLinux se puede elegir entre dos resoluciones de reloj: ticks del 8254 o nanosegundos.

- La gestión del tiempo está definida en el estándar 1003.1a (Extensiones de tiempo real) y no en el estándar que define los PThreads (1003.1b).

CLOCK_REALTIME

Reloj del sistema. Si nuestro sistema soporta la instrucción ensamblador `rdtsc` (instrucción disponible a partir del procesador Pentium®) entonces se utiliza esta instrucción para obtener la hora (con esta fuente de tiempos se obtiene una resolución próxima al nanosegundo), en caso contrario se obtiene del chip 8254 alimentado a 1193180Hz (0.83 microsegundos). Este reloj puede sufrir ajustes para corregir la fecha.

CLOCK_MONOTONIC

Igual que **CLOCK_REALTIME** pero no se realizan ajustes, por tanto su cuenta es creciente sin saltos bruscos. Útil para medir duraciones de eventos.

CLOCK_8254

Obtenido a partir del chip 8254. Las instrucciones de E/S para leer el valor del reloj (contador) suelen ser muy lentas (del orden de microsegundos) ya que este chip se encuentra conectado al bus ISA. No utilizar si se puede evitar.

CLOCK_APIC

Los micros que soportan SMP disponen de un *Advanced Programmable Interrupt Controller* (APIC) integrado en el mismo procesador, que entre otras funciones dispone de un temporizador programable. La frecuencia de funcionamiento de este reloj es la misma que la del procesador principal.

6.4 Soporte hardware

- El chip 8254 ([Specs](#)) tiene varios modos de programación. Este chip está alimentado con una señal de 1.193.180Hz.
 - ♦ En el modo periódico (MODE 2), se inicializa el chip con un valor que determina la frecuencia de interrupción y a partir de ese momento, y sin intervención alguna por parte del procesador central, el chip envía interrupciones a esa frecuencia. En los sistemas operativos clásicos se utiliza este modo con una frecuencia de entre 18 y 100 Hz.
 - ♦ En el modo de disparo único (MODE 0, llamado one-shot en RTLinux) se carga el contador del reloj con un valor que al llegar a cero produce un interrupción. Es necesario reprogramar el contador en cada interrupción, *operación que requiere del orden de milisegundos*.
- El APIC es un periférico integrado dentro de propio microprocesador. Apareció en el Pentium® para poder distribuir las interrupciones en sistemas multiprocesador. Los procesadores Celeron® no disponen de APIC y todos los procesadores de AMD no dispuesto de este periférico a excepción Athlon® model 2.
- La descripción del local APIC se puede encontrar en [Intel Architecture Software Developer's Manual, Volume 3, Chapter 7, Multiple Processor Management](#)
- Entre otras características, el APIC dispone de un contador capaz de producir interrupciones temporizadas con muy poca sobrecarga de programación.
- Todos los procesadores Pentium® y compatibles disponen de un registro contador de 64 bits que se incrementa automáticamente cada tick de reloj del procesador.
- La instrucción ensamblador `rdtsc` (*read time stamp counter*) devuelve el número de ciclos de procesador transcurridos desde el arranque de la máquina.
- La precisión depende de la frecuencia del oscilador del procesador. Utilizando como referencia el reloj 8254 es posible determinar con precisión la frecuencia del propio procesador.
- La lectura de este registro es muy rápida y no interfiere con la ejecución normal de los procesos (no accede a buses ni se necesitan muchas líneas de código que llenan la cache).
- A partir del modelo AT, los PC's incorporan el chip MC146818 de Motorola alimentado por una pequeña batería para mantener la fecha y hora mientras el sistema está apagado.
- El chip dispone de un contador con la capacidad de producir interrupciones (interrupción hardware 8) desde 2Hz hasta 8192Hz en incrementos de potencias de 2. (Ver documentación de Linux [rtc.txt](#))
- Este temporizador se le conoce como RTC (Real Time Clock), no porque sea *de* Tiempo Real sino que es *en* tiempo real.
- Este chip también cuenta con 64 bytes de RAM, conocida como la CMOS, donde se guardan parámetros de configuración de equipo.

6.5 Estructuras de datos

- Tipos y constantes de tiempos:

```
struct timespec {
    time_t tv_sec; /* seconds */
    long tv_nsec; /* nanoseconds */
};
typedef struct rtl_clock *clockid_t;
typedef long long hrttime_t;
typedef unsigned useconds_t;
#define HRTIME_INFINITY 0x7fffFfffFfffFfffLL
```

- Las declaraciones de las funciones y estructuras de datos relacionadas con el tiempo están localizadas en los ficheros `/posix/time.h` y `rtl_time.h`. En nuestros programas basta con añadir `#include <time.h>`, ya que desde este fichero de cabecera se incluyen los otros dos.

6.6 Listado de funciones

Leer el tiempo

```
clock_gettime
clock_gettime
gethrtime
```

Resolución

```
clock_getres
gethrtimeres
```

Temporización

```
rtl_delay
usleep
nanosleep
clock_nanosleep
```

Control

```
rtl_getschedclock
rtl_setclockmode
rtl_getbestclock
```

Manjadores de interrupciones

```
rtl_setclockhandler
rtl_unsetclockhandler
```

6.7 Consultar el reloj

```
clock_gettime(clockid_t clock, struct timespec *time)
```

Devuelve el tiempo actual en la estructura `time`.

```
struct timespec ahora;
```

```
clock_gettime(CLOCK_REALTIME, &ahora);
```

```
rtl_printf("%ld, %ld", ahora.tv_sec, ahora.tv_nsec);
```

```
hrttime_t clock_gethrtime(clockid_t clock)
```

Función no estándar equivalente a `clock_gethrtime` pero que devuelve el tiempo como un `long` (64 bits). Devuelve el tiempo medido en nanosegundos:

```
hrttime_t inicio, fin;
```

```
inicio = clock_gethrtime(CLOCK_REALTIME);
```

```
codigo_a_medir();
```

```
fin = clock_gethrtime(CLOCK_REALTIME);
```

```
rtl_printf("Duración = %ld", fin-inicio);
```

```
gethrtime()
```

Igual que la anterior pero aún más espartana. No se le pasa el reloj del que queremos que lea la hora, lo toma del más eficiente que haya disponible.

```
clock_getres(clockid_t clock, struct timespec *time)
```

Función que devuelve en el parámetro `time` la resolución de `clock`. La resolución es la mínima unidad de tiempo que puede diferenciar el reloj. La resolución no tiene nada que ver con la precisión.

```
hrttime_t gethrtimeres()
```

Resolución del tiempo devuelto por `gethrtime()`.

6.8 Esperas

```
rtl_delay(long duracion)
```

Realiza una **espera activa** hasta que hayan transcurrido `duracion` **nanosegundos**. `duracion` es un `long`. Puede ser invocada desde una `rt-task` o un manejador de interrupciones de RTLinux.

```
usleep(unsigned duracion)
```

Realiza una **espera no activa** hasta que hayan transcurrido duracion **microsegundos**. duracion de tipo unsigned. La espera se realiza suspendiendo el thread, por lo que no se puede llamar esta función desde un manejador de interrupciones.

- Las siguientes funciones están declaradas en `rtl_sched.h`:

```
int clock_nanosleep(clockid_t clock_id, int flags, struct timespec *espera,
struct timespec *restante)
```

Espera no activa. `clock_id` es el reloj que se utilizará realizar la espera. El bit `TIMER_ABSTIME` del parámetro `flags` ha de estar activo si los tiempos son absolutos. En `espera` se indica el tiempo que queremos esperar y en `restante` obtenemos el tiempo que nos quedaría por esperar en caso de ser abortada la llamada debido a una señal. En cuyo caso la función finaliza con un valor distinto de cero.

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

Equivale a llamar a `clock_nanosleep(CLOCK_MONOTONIC, 0, espera, restante)`. Esto es, espera relativa sobre el reloj `CLOCK_MONOTONIC`.

6.9 Miscelánea de funciones

```
clockid_t rtl_getschedclock()
```

Función no estándar que devuelve el identificador del reloj utilizado por el planificador.

```
clockid_t rtl_getbestclock(int cpu)
```

Devuelve el identificador del mejor reloj del sistema. Si sólo tenemos un procesador entonces `cpu` ha de valer cero. Si el procesador dispone de APIC entonces se utilizará ese, sino `CLOCK_8254`.

```
clockid_t The_Best;
```

```
struct timespec resolucion;
```

```
The_Best = rtl_getbestclock(0);
```

```
clock_getres(The_Best,
```

```
rtl_printf("Resol: %ld", timespec_to_ns(
```

6.10 Modo de operación

```
rtl_setclockmode(clockid_t clock, int mode, hrtime_t period)
```

Establece el modo de operación del reloj especificado. Los modos soportados son:

`RTL_CLOCK_MODE_ONESHOT` y `RTL_CLOCK_MODE_PERIODIC`. En caso de utilizar el modo periódico es necesario especificar la frecuencia de interrupción en el tercer parámetro.

El modo por defecto en todos los relojes es `RTL_CLOCK_MODE_ONESHOT`.

```
clockid_t cl= rtl_getschedclock();
```

```
hrtime_t period = 1000000; // 1 milisegundo.
```

```
rtl_setclockmode (cl , RTL_CLOCK_MODE_PERIODIC, period);
```

- Para obtener una buena granularidad en modo `RTL_CLOCK_MODE_PERIODIC` se tiene que programar el periodo de interrupción a una alta frecuencia. Lo que produce una alta sobrecarga debido al gran número de interrupciones recibidas.

Sólo se necesita programar el hardware del temporizador una sola vez.

- En modo `RTL_CLOCK_MODE_ONESHOT` el temporizador se tiene que reprogramar en cada interrupción para que produzca la siguiente.

Con este método se puede obtener una granularidad igual a la resolución del hardware, a costa de reprogramar frecuentemente el temporizador (este problema es el que limita la frecuencia de operación cuando se utiliza el 8254).

6.11 Interrupción de reloj

```
rtl_setclockhandler(clockid_t clock, clock_irq_handler_t manejador)
```

Aunque RTLinux dispone de funciones para capturar interrupciones hardware, la interrupción de reloj necesita de un trato especial para asegurar la estabilidad del sistema.

Las interrupciones de reloj son una de las piezas clave para poder trabajar un sistema operativo. Sería desastroso capturar o modificar la frecuencia de las interrupciones de reloj.

RTLinux dispone de funciones para capturar interrupciones, pero este mecanismo NO debe utilizarse con la interrupción de reloj. La solución consiste en utilizar una interrupción de reloj virtual. RTLinux se asegura de que el núcleo de Linux recibe interrupciones de reloj a la frecuencia correcta, y nosotros podamos instalar un

manejador de interrupción de reloj.

manejador es una función que no retorna nada y recibe como parámetros un puntero a struct pt_regs (registros del procesador), que normalmente no se utiliza.

Sólo puede estar instalada una función manejadora por reloj. Y por desgracia el planificador (rtl_sched) necesita instalar un manejador. Por tanto, si usamos el planificador (todas las funciones pthread_create, pthread_make_periodic_np, etc. etc. ...) no podremos usar el manejador (Ver [regression/oneshot_test.c](#)).

```
rtl_unsetclockhandler( clockid_tclock)
```

Elimina el manejador previamente instalado.

6.12 Operar con tiempos

- Hacer operaciones con tiempo expresados como variables de tipo struct timespec es bastante engorroso. RTLinux dispone de una batería de funciones y macros para facilitar la existencia a los programadores:

```
timespec_nz(t)          t != 0
timespec_lt(t1,t2)      t1 < t2
timespec_gt(t1,t2)      t1 > t2
timespec_le(t1,t2)      t1 <= t2
timespec_ge(t1,t2)      t1 >= t2
timespec_eq(t1,t2)      t1 == t2
timespec_add(t1,t2)     t1 = t1+t2
timespec_sub(t1,t2)     t1 = t1-t2
```

```
hrtime_t timespec_to_ns (const struct timespec *ts)
struct timespec timespec_from_ns (hrtime_t t)
```


7 Sincronización: Mutex, Variables condición y Semáforos.

Ismael Ripoll
iripoll@disca.upv.es
<http://bernia.disca.upv.es/~iripoll>

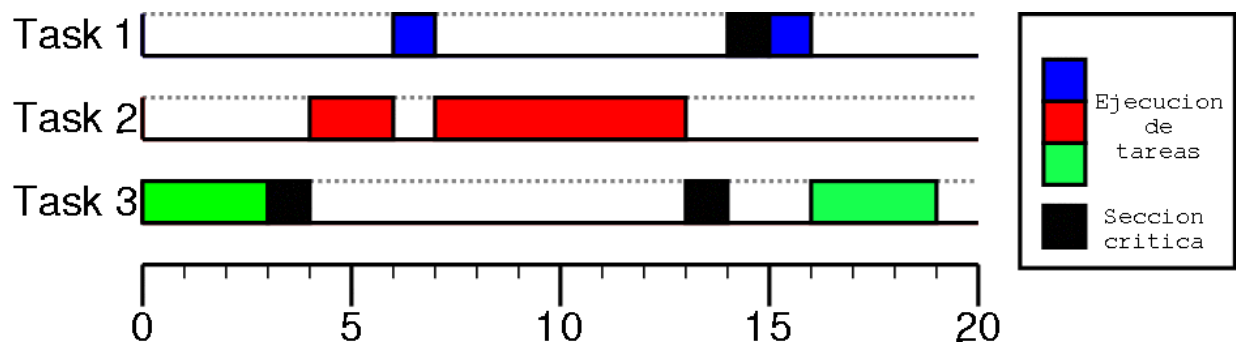
Permission is granted to copy, distribute and/or modify this document under the terms of the [OpenContent License](#).

7.1 Introducción

- Las funciones relacionadas con la gestión de recursos están contenidas en el mismo módulo que el planificador. Por tanto para hacer uso de ellas es necesario tener cargado el módulo `rtl_sched.o`.
- POSIX utiliza un mecanismo similar a los semáforos binarios clásicos para controlar el acceso a las secciones críticas.
- Los semáforos reciben el nombre de "mutex" y las operaciones "P" y "V" pasan a denominarse `pthread_mutex_lock` y `pthread_mutex_unlock`.
- Victor Yodaiken ha sido muy reacio a implementar semáforos en RTLinux. Él cree que la solución más eficiente consiste en hacer las secciones críticas atómicas mediante el bloqueo de las interrupciones.

7.2 Inversión de prioridad

- *Las prioridades están reñidas con las secciones críticas.*



- Una inversión de prioridad no limitada se produce cuando un proceso de prioridad intermedia retrasa la ejecución de otro de prioridad superior porque este último está bloqueado a la espera de un recurso controlado por un tercer proceso de baja prioridad.
- Existen varios métodos para resolver o minimizar el problema de la i.v. no limitada. Los más utilizados se basan en el concepto de **herencia de prioridad**: la tarea que controla un semáforo puede heredar (subir) su prioridad para no ser interrumpida y salir cuanto antes de la sección crítica. A partir de la versión de 3.0 (1-6-00) de RTLinux ya se dispone de semáforos normales y techo inmediato.
- A partir de esta idea de herencia se han desarrollado varios protocolos de semáforos: herencia simple, techo de prioridad, techo inmediato, etc.
- El estándar PThreads ha incorporado dos de estos protocolos: herencia simple y techo inmediato, denominados: `PTHREAD_PRIO_INHERIT` y `PTHREAD_PRIO_PROTECT` respectivamente. Además del semáforo "normal" sin control de inversión de prio.

7.3 Utilización

- Para utilizar las primitivas de sincronización (semáforos y variables condición) es necesario incluir el fichero `pthread.h` e insertar en el núcleo los módulos `rtl.o` y `rtl_sched.o`.
- Declaraciones de tipos y constantes:

```
typedef struct {...} pthread_mutexattr_t;
typedef struct {...} pthread_mutex_t;

typedef struct {...} pthread_condattr_t;
typedef struct {...} pthread_cond_t;
```

7.4 Listado de funciones

Semáforos

```
pthread_mutex_init
pthread_mutex_destroy
pthread_mutex_lock
pthread_mutex_trylock
pthread_mutex_unlock
```

Atributos de semáforos

```
pthread_mutexattr_init
pthread_mutexattr_settype
pthread_mutexattr_destroy
pthread_mutexattr_setpshared
pthread_mutexattr_getpshared
pthread_mutexattr_settype
pthread_mutexattr_gettype
pthread_mutexattr_setprotocol
pthread_mutexattr_getprotocol
pthread_mutexattr_setprioceiling
pthread_mutexattr_getprioceiling
```

Variables condición

```
pthread_cond_init
pthread_cond_destroy
pthread_cond_wait
pthread_cond_broadcast
pthread_cond_signal
pthread_cond_timedwait
```

Atributos var. cond.

```
pthread_condattr_init
pthread_condattr_destroy
pthread_condattr_getpshared
pthread_condattr_setpshared
```

Herencia dinámica

```
pthread_mutex_setprioceiling
pthread_mutex_getprioceiling
```

7.5 Semáforos

*pthread_mutex_lock(pthread_mutex_t *mutex)*

Realiza una operación "P" sobre una variable semáforo. Utilizada para entrar en una sección crítica de código.

- ◊ Si el semáforo está abierto entonces cierra (bloquea) el semáforo y continua la ejecución.
- ◊ Si el semáforo está cerrado (bloqueado) entonces se suspende el thread que invoca esta función.

Si el semáforo es de tipo PTHREAD_PRIO_PROTECT entonces además de bloquear el semáforo, el thread hereda la prioridad del techo del mutex. En este caso si el thread tiene menos prioridad que el techo del semáforo entonces no se cierra el semáforo y devuelve un error. Ver *pthread_mutexattr_setprioceiling*.

*pthread_mutex_unlock(pthread_mutex_t *mutex)*

Realiza una operación "V" sobre el semáforo. Salida de sección crítica.

Esta operación desbloquea el semáforo y luego despierta a todos los threads que estén bloqueados intentando acceder a este semáforo. Todos vuelven a intentar bloquear, pero sólo el thread "más rápido" conseguirá el semáforo, el resto volverá a bloquearse.

Importante: El acceso a una sección crítica NO sigue una política FCFS, sino que seguirá la misma política que el procesador: basada en prioridades fijas. Si se cambia la política de planificación del procesador entonces también cambiará la política de las colas de semáforos.

*pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr)*

Inicializa el semáforo para poder ser utilizado. POSIX define distintos atributos que modifican el comportamiento de los semáforos. Por defecto el mutex no utilizará ningún tipo de herencia de prioridades.

*pthread_mutex_destroy(pthread_mutex_t *mutex)*

Evidente... En la implementación actual esta función no hace nada, sólo existe por compatibilidad.

*pthread_mutex_trylock(pthread_mutex_t *mutex)*

Intenta bloquear un semáforo pero si no lo consigue (otro thread lo mantiene bloqueado) entonces retorna inmediatamente un código de error, pero no bloquea al thread invocante.

7.6 Atributos de semáforos

- Los posibles atributos definidos de los mutex por POSIX son:

- ◈ **tipo:** PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_SPINLOCK_NP, PTHREAD_MUTEX_RECURSIVE, PTHREAD_MUTEX_ERRORCHECK.
Sólo se han implementado los dos primeros.

♦ **compartido**: PTHREAD_PROCESS_SHARED, PTHREAD_PROCESS_PRIVATE.
RTLinux hace caso omiso de este atributo.

♦ **protocolo**: PTHREAD_PRIO_PROTECT, PTHREAD_PRIO_NONE.

En el protocolo PTHREAD_PRIO_PROTECT es necesario especificar el techo de prioridad del mutex.

- Las funciones para manejar los atributos de creación de mutex son:

```
pthread_mutexattr_init(pthread_mutex_t *attr)
pthread_mutexattr_destroy(pthread_mutex_t *attr)
pthread_mutexattr_[set/get]type(*attr, int type)
pthread_mutexattr_[set/get]pshared(*attr, int shared)
pthread_mutexattr_[set/get]protocol(*attr, int proto)
pthread_mutexattr_[set/get]prioceiling(*attr, int pri)
```

! Creo que los *tios* de POSIX confundieron el nombre de las funciones con su descripción 😊 ! ... Son auto-explicativas.

7.7 .._SPINLOCK_NP un caso especial

- A partir de 16-5-00 se añadió un nuevo tipo de mutex para trabajar con multiprocesadores: PTHREAD_MUTEX_SPINLOCK_NP.
- Este tipo de mutex realiza espera activa. Esto es, en lugar de suspender el proceso si el mutex está cerrado, el proceso que invoca la operación de "lock" queda en un bucle infinito intentando constantemente obtener el semáforo.
- Sólo tienen sentido en sistema con varios procesadores ya que de lo contrario se producirían interbloqueos.
- son más rápidos cuando se sincronizan threads situados en CPU's distintas y la sección crítica es muy corta, por ejemplo en los accesos al hardware.

7.8 Techos de prioridad dinámicos

```
pthread_mutex_setprioceiling(pthread_mutex_t *mutex, prio)
```

Únicamente modifica el valor de techo de prioridad de *mutex*, sin modificar la prioridad del thread que lo pudiera tener bloqueado en ese momento.

Esta función no se debería utilizar ya que no está respaldada por la teoría de planificación.

```
pthread_mutex_getprioceiling(pthread_mutex_t *mutex, *prioceiling)
```

Retorna en el entero *prioceiling* el valor del techo de prioridad de *mutex*.

7.9 Variables condición

- Las variables condición son los tipos de datos y funciones necesarias para, junto con los mutex, poder construir "monitores". Entendiendo monitor como un conjunto de funciones que operan sobre un conjunto de datos en exclusión mutua.
- Supongamos que implementamos un buffer circular utilizando por dos tareas. Una de ellas tiene que escribir los datos leídos de un sensor y la otra los lee y procesa. El código para acceder al buffer circular es una sección crítica y ha de estar protegido por semáforos.

El problema surge si la tarea lectora intenta leer del buffer cuando está vacío. La operación de lectura se debe bloquear hasta que se escriba algo en el buffer.

- El nombre de "variable condición" no es especialmente acertado para describir su utilidad 😊.
- Una variable condición es un tipo de datos sobre el que se pueden realizar dos operaciones: wait y signal:

- ♦ La operación de espera (**wait**) produce la suspensión incondicional del thread que la invoca. Esta suspensión lleva aparejada la salida de la sección crítica en la que se está.
- ♦ La operación de despertar (**signal**) reanuda la ejecución de uno de los threads suspendidos. Antes de continuar la ejecución se vuelve a pedir entrar en la S.C. de la que se había salido al suspenderse.
- En el último mes se han incorporado a RTLinux las variables condición.
- Aunque RTLinux implementa las funciones para gestionar los atributos de las variables condición, no se hace ningún uso en la actual implementación.

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)
    Inicializa la variable condición. El valor de attr puede ser NULL ya que no se utiliza.
int pthread_cond_destroy(pthread_cond_t *cond)
    No hace nada. return 0.
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
    De forma atómica: libera el semáforo mutex y suspende al thread que la invoca en la cola de cond. Cuando el thread sea despertado lo primero que se hace es pedir nuevamente el semáforo mutex. Siempre se bloquea el thread que invoca esta función.
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)
    Igual que pthread_cond_wait() pero devuelve el código de error ETIMEDOUT en caso de no ser despertado antes del valor indicado en abstime. En cualquier caso, al continuar la ejecución se vuelve a pedir el semáforo mutex.
int pthread_cond_broadcast(pthread_cond_t *cond)
    Desbloquea todos los threads que estuvieran esperando en la cola cond.
int pthread_cond_signal(pthread_cond_t *cond)
    Desbloquea todos los threads que estuvieran esperando en la cola cond. De hecho esta función es una macro a pthread_cond_broadcast(). Ver include/rtl\_mutex.h.
```

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

```
int in, out, cont, buffer[10];
pthread_cond_t lleno, vacio;
pthread_mutex_t semafor;

Escribe(int DATO){
    pthread_mutex_lock(&semafor);
    while (cont == 10)
        pthread_cond_wait(&vacio, &semafor);
    cont++; buffer[in]= DATO;
    in = (in+1) % 10;
    pthread_cond_broadcast(&lleno);
    pthread_mutex_unlock(&semafor);
}

int Lee(){
    int dato;
    pthread_mutex_lock(&semafor);
    while (cont == 0)
        pthread_cond_wait(&lleno, &semafor);
    cont--; dato = buffer[out];
    out = (out+1) % 10;
    pthread_cond_broadcast(&vacio);
    pthread_mutex_unlock(&semafor);
    return dato;
}

void *productor(void *arg){
    int i;
    printf("Hijo\n");
    for (i= 0; i< 100; i++)
        Escribe(i);
    pthread_exit(0);
}


main(){
    int i;
    pthread_t hijo;
    in = out = cont = 0;

    pthread_mutex_init(&semafor, NULL);
    pthread_cond_init(&lleno, NULL);
    pthread_cond_init(&vacio, NULL);
    pthread_create(&hijo, NULL, productor, NULL);
    printf("Padre\n");
    for (i= 0; i< 100; i++)
        printf("%d\n ", Lee());
    exit(0);
}
```

7.10 Semáforos generales

- Hace unas semanas se incorporaron a RTLinux los semáforos generales.
- No añaden nuevas funcionalidades. Con los mutex y las variables condición se pueden resolver cualquier tipo problemática de sincronización.
- Las operaciones que se pueden realizar sobre un semáforo general son:
 - wait**
Si el contador es mayor que cero entonces decrementa el contador y continua. En caso contrario se bloquea el thread.
 - post**
Si hay algún thread bloqueado entonces lo despierta, en caso contrario incrementa el contador.

7.11 Funciones

```
int sem_init(sem_t *sem, int pshared, unsigned int value )
    Inicializa sem con el valor inicial del contador a value.
sem_destroy(sem_t *sem) Evidente.
sem_wait(sem_t *sem) Ya comentado.
sem_post(sem_t *sem) 
int sem_timedwait(sem_t *sem, const struct timespec *abstime) Previsible.
int sem_getvalue(sem_t *sem, int *sval)
    Devuelve en sval el valor del contador. El mejor consejo es no usar nunca esta función.
```

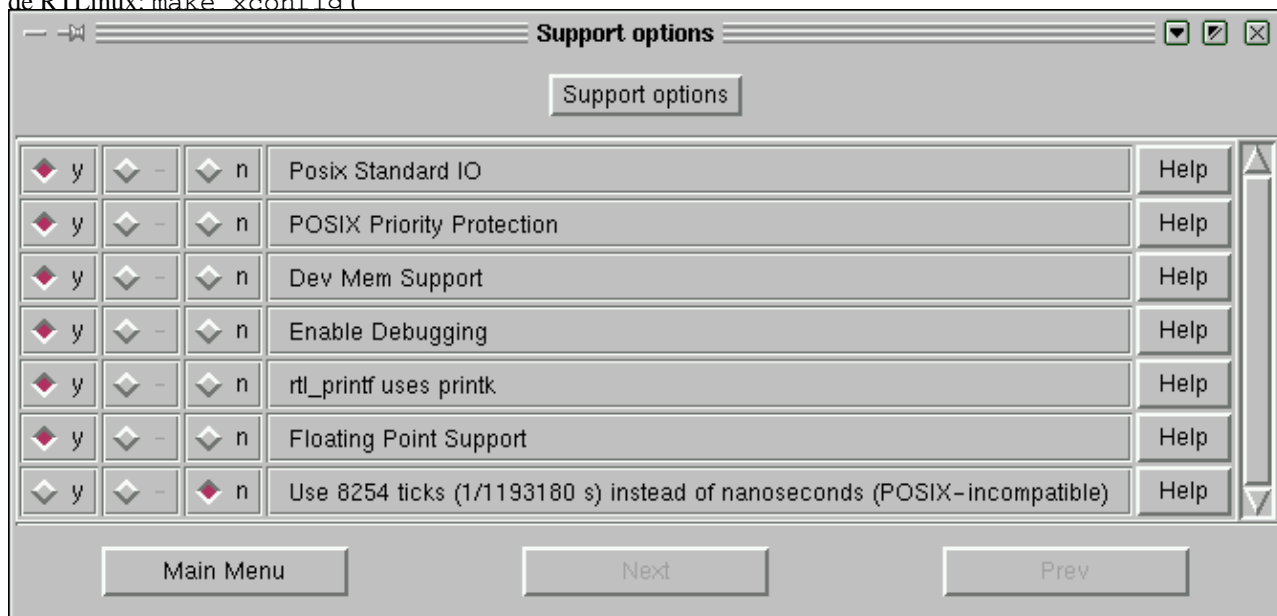

8 Dispositivos POSIX y acceso a la Entrada/Salida

Ismael Ripoll
iripoll@disca.upv.es
<http://bernia.disca.upv.es/~iripoll>

Permission is granted to copy, distribute and/or modify this document under the terms of the [OpenContent License](#).

8.1 Introducción

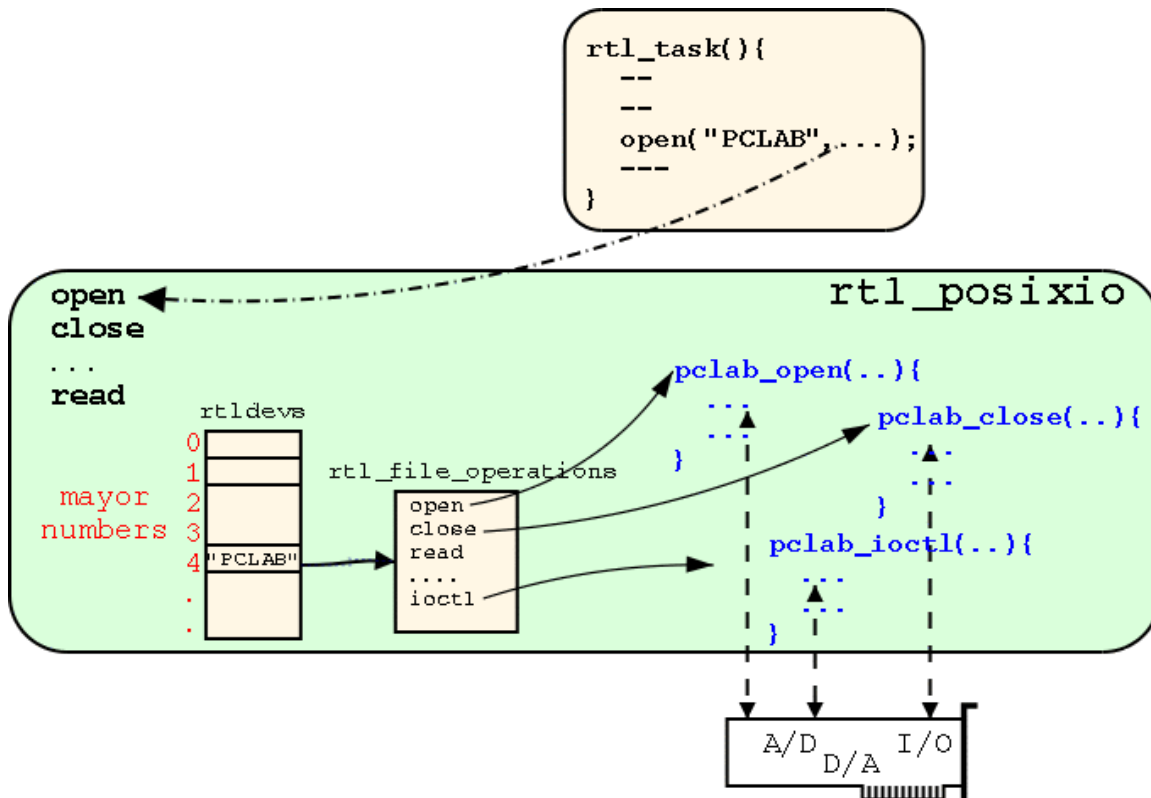
- El sistema de entrada/salida de UNIX está basado en el concepto de fichero. La práctica totalidad de las operaciones que se pueden realizar sobre la E/S se realizan a través de "ficheros".
- POSIX.4 sigue conservando esta filosofía de trabajo, pero de forma más restringida. Se conservan las funciones de acceso a los ficheros, pero no se requiere la existencia de un "sistema de ficheros" completo, únicamente los ficheros especiales (dispositivos) disponibles.
- POSIX obliga a que todos los ficheros especiales residan en el directorio `/dev`. Es importante resaltar que el resto de ficheros y directorios (`/etc`, `/usr`, `/bin`, etc.) no es necesario implementarlos.
- El acceso a los dispositivos al estilo POSIX es una característica opcional que se elige al configurar los fuentes de RTLinux: `make xconfig`



). Marcando la opción de POSIX-IO, RTLinux nos permitirá registrar dispositivos para ser utilizados mediante las operaciones clásicas de ficheros.

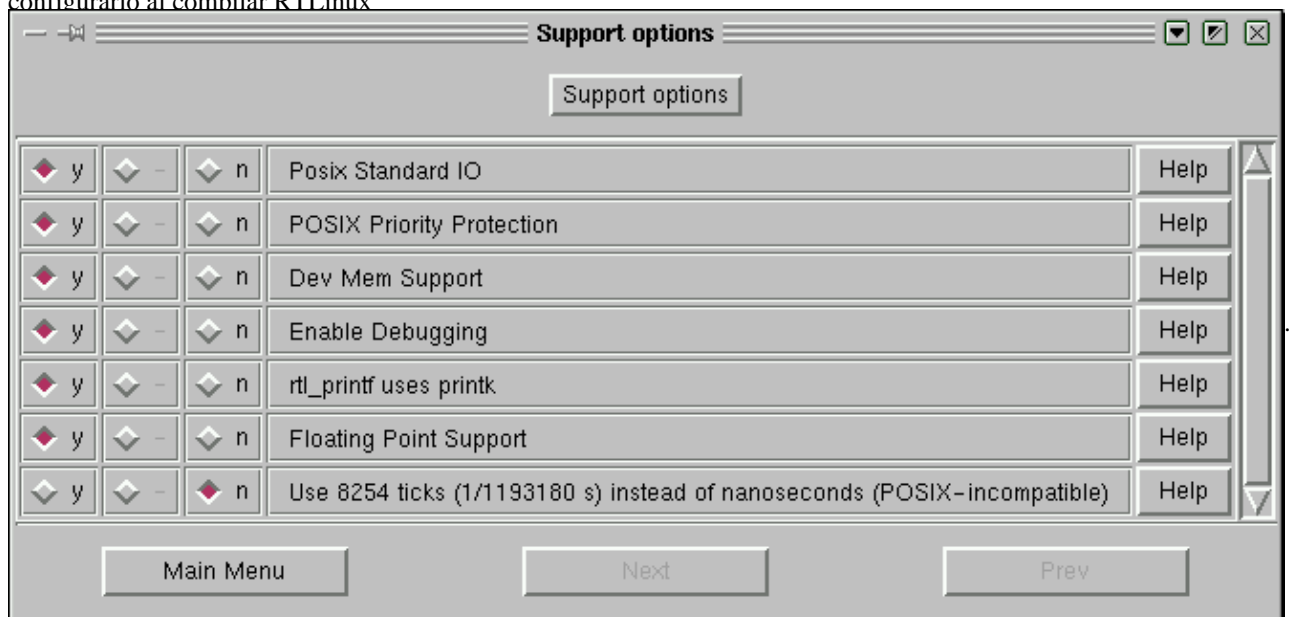
- No tendremos ficheros especiales si no se registran "drivers".
- El directorio `/dev` de RTLinux no tiene ninguna relación con el directorio del mismo nombre de Linux.
- El número máximo de ficheros que pueden abrirse en el sistema es de 128.

8.2 Estructura global



8.3 /dev/mem

- RTlinux dispone de un fichero especial para acceder a la memoria física del sistema: `/dev/mem`. Es necesario configurarlo al compilar RTLinux




- Con este dispositivo especial se puede acceder a la memoria física como si se tratara de un fichero. Mediante este mecanismo se consigue escribir código portable e independiente del hardware.
- De todas formas, si no disponemos de la función `lseek`, este driver tiene poca utilidad, excepto como ejemplo para crear nuevos drivers.

8.4 Registrar nuevos dispositivos

- En procedimiento para registrar nuevos dispositivos es:
 1. Declarar una estructura del tipo `struct rtl_file_operations`.
 2. Implementar las funciones que nuestro "driver" ofrezca de entre las funciones que aparecen en `rtl_file_operations`.
 3. Rellenar la estructura `rtl_file_operations` con esas funciones.
 4. Registrar nuestro "driver" llamando a la función `rtl_register_rtldev`. Pasando como parámetros: el *mayor number*, nombre del dispositivo y el puntero a la estructura `rtl_file_operations`.
- Para eliminar un dispositivo sólo es necesario llamar a `rtl_unregister_rtldev`.
- Al final del propio fichero donde se implementan todas las funciones para manejar el interfaz de ficheros, hay un excelente ejemplo de uso con el dispositivo `/dev/mem`: [system/rtl_posixio.c](#)
- Inicialmente el módulo de POSIX IO estaba diseñado para dispositivos orientados a carácter, esto es, a flujo de datos y no a bloques. Por tanto, algunas de las funciones que aparecen en `rtl_file_operations`, no están completamente operativas: `poll`, `mmap` y `llseek`. Por lo que no es necesario (conveniente) implementarlas.

8.5 Programación de la E/S

- Para acceder a los puertos de entrada salida o a posiciones concretas de memoria podemos utilizar las mismas macros y funciones inline utilizadas por Linux. Funciones que se encuentran en el fichero `#include <asm/io.h>`
- El fichero `io.h` se puede calificar de "ofuscado"  en el mejor de los casos. Pero siempre podemos utilizar el preprocesador de "C" para obtener un resultado más legible:

```
gcc -O2 -D__KERNEL__ -E /usr/include/asm/io.h
```

 En la salida de esta orden encontraremos mucha (demasiada) información sobre todo tipo de estructuras de datos.
- Las funciones de E/S aquí declaradas son muy simples y su funcionamiento es muy intuitivo:
- El acceso a los puertos se puede realizar desde cualquier programa: `rtl-tasks`, manejadores de interrupciones de tiempo real y por supuesto desde el núcleo de Linux.
- ***Siempre que se utilice el fichero `io.h` es necesario utilizar la opción del compilador de optimización `"-O2"` o superior. Si no se genera correctamente el código ensamblador.***

Entrada y salida rápida:

```
void outb(unsigned char value, unsigned short port);
void outw(unsigned short value, unsigned short port);
void outl(unsigned int value, unsigned short port);
unsigned char inb(unsigned short port);
unsigned short inw(unsigned short port);
unsigned int inl(unsigned short port);
```

Después de realizar la operación se fuerza una espera. Necesarias para poder trabajar con hardware lento (bus ISA):

```
void outb_p(unsigned char value, unsigned short port);
void outw_p(unsigned short value, unsigned short port);
void outl_p(unsigned int value, unsigned short port);
unsigned char inb_p(unsigned short port);
unsigned short inw_p(unsigned short port);
unsigned int inl_p(unsigned short port);
```

Entradas y salidas de cadenas:

```
void outsb(unsigned short port, *addr, unsigned long count);
void outsw(unsigned short port, *addr, unsigned long count);
void outsl(unsigned short port, *addr, unsigned long count);
void insb(unsigned short port, *addr, unsigned long count);
void insw(unsigned short port, *addr, unsigned long count);
void insl(unsigned short port, *addr, unsigned long count);
```

8.6 Acceso a la memoria física

Aunque en Linux i386 es posible acceder a todas las posiciones de memoria física directamente puesto que las direcciones físicas están mapeadas directamente en el espacio de direcciones del núcleo, también podemos utilizar las siguientes macros:

```
unsigned char readb(addr); // equivale a: (*addr)
unsigned short readw(addr);
unsigned int readl(addr);
writeb(unsigned char b, addr); // equivale a: (*addr) = b
writew(unsigned short b, addr);
writel(unsigned int b, addr);
```

Para mover bloques de memoria:

```
memset_io(addr, int c, int count);
memcpy_fromio(addr_dest, addr_org, count);
memcpy_toio(addr_org, addr_dest, count);
```

8.7 Programación de puertos desde procesos Linux

- Los procesos de Linux "normales" se ejecutan en "ring level 3". En este nivel el procesador no permite realizar operaciones de E/S (entre otras) a no ser que se habilite un mapa de bits de puertos permitidos.
- La llamada al sistema `ioperm()` permite a los procesos acceder a los puertos creando el mapa de bits correspondiente.
- La llamada al sistema `iopl()` permite a los procesos modificar el nivel de privilegio al que se ejecutan. El nivel 0 es el de máximo privilegio.
- Todas estas protecciones y controles son gestionados automáticamente el procesador (Pentium).

`ioperm(u_long from, u_long num, int turn_on)`

Si `turn_on` es true entonces permite el acceso a rango de puertos [`from`...`from+num-1`]. Si `turn_on` es falso entonces retira el permiso. Solo root puede utilizarla y solo se pueden activar o desactivar los 0x3fff primeros puertos.

`iopl(int level)`

Estable del nivel de privilegio del proceso que invoca la llamada. Este atributo de proceso es heredado por los procesos hijos.

- Ejemplo de proceso Linux que accede a la E/S: [linux_io.c](#)
- Ejemplo módulo que accede a la E/S: [modulo_io.c](#)

9 FIFO

Ismael Ripoll
iripoll@disca.upv.es
<http://bernia.disca.upv.es/~iripoll>

Permission is granted to copy, distribute and/or modify this document under the terms of the [OpenContent License](#).

9.1 Introducción

- Las FIFO son un mecanismo de comunicación basado en las "fifo" de UNIX, pero adaptado a tiempo real. Las fifo implementadas en RTLinux no tienen ninguna relación con las fifo clásicas de UNIX (ver `mkfifo` y `mknod`). Ambas fifos con implementaciones distintas de un mismo método de comunicación.
- Se pueden utilizar para comunicar tanto threads entre sí como threads con procesos Linux.
- Son una funcionalidad de tiempo real opcional, y como tal hay que cargar el módulo correspondiente para poder utilizarlos.
- La comunicación es unidireccional, esto es, se comporta como un buffer circular de forma que cada operación de lectura elimina del buffer los datos leídos.
- Existen dos implementaciones distintas: `rtl_fifo` y `rtl_nfifo`. El API que ofrecen a los procesos Linux es el mismo.

`rtl_fifo`

Primera implementación. Para trabajar con estas FIFO se pueden utilizar funciones propias de RTLinux, como por ejemplo `rtf_put`, `rtf_get`, etc. Y por otra parte utilizando el interfaz de dispositivos de POSIX: `open("/dev/rtf0",...)`, `read`, `write`, etc

`rtl_nfifo`

La nueva implementación ofrece además de los dos interfaces anteriores un interfaz similar al de las colas de mensajes de POSIX: `mq_fast_send`, `mq_fast_send`, etc., etc. Este API permite pasar de una comunicación por *stream* (flujo de datos) a otra por mensajes, en la que se puede definir claramente la unidad de información.

9.2 Programación desde procesos Linux

- Las fifo se "ven" desde los procesos Linux como dispositivos especiales de caracteres (con *mayor number* 150): `/dev/rtfx`.
- Si bien estos dispositivos especiales no suelen estar en las distribuciones normales de Linux, son creados por los scripts de instalación de RTLinux. De todas formas es posible crearlos utilizando la orden: `mknod /dev/rtf0 c 150 0`.
- Se pueden utilizar hasta 64 fifos distintas, si bien se puede incrementar este número hasta 248 modificando la macro `RTF_NO` en el fichero [include/rtl_fifo.h](#). No es posible disponer de 256 fifos ya que las últimas 8 están reservadas.
- Para poder abrir un dispositivo fifo es necesario que una tarea de RTLinux lo haya creado mediante la llamada `rtf_create`.
- Programa ejemplo:

```
#include <fcntl.h>

main () {
    int fd, count;
    int valor;;
    fd=open("/dev/rtf0", O_RDONLY);
    while (1) {
        read(fd, sizeof(valor));
        printf("valor: %8d\n", valor);
    }
}
```
- Evidentemente también es posible leer o escribir sobre las fifo's utilizando ordenes UNIX convencionales: `cat /dev/rtf0`.

9.3 Copia de variables por FIFO

- Cuando se compila con la opción de optimización el código que se ejecuta puede ser bastante distinto del que creemos.

Supongamos tenemos declarada una estructura que contiene un `char`, un `int` y otro `char` y la enviamos por una fifo. No es igual leer de la fifo una variable del mismo tipo que la estructura, que los tres componentes por separado ya que es posible que el compilador haya rellenado la estructura para que todos los elementos estén alineados a inicio de palabra.

Si necesitamos conocer la posición real de cada miembro de la estructura, la utilidad de Perl `pstruct` puede ser útil.

9.4 Listado de funciones

```

Creación / Destrucción
rtf_create
rtf_destroy

Trabajar con fifos
rtf_put
rtf_get
rtf_flush
rtf_isempty

open, close, read, write

Manejadores de eventos
rtf_create_handler
rtf_create_rt_handler

```

9.5 Crear FIFOS

- El fichero de cabecera que hay que incluir para usar estas llamadas es [include/rtl_fifo.h](#).

`rtf_create(fifo_nr, size)`

Crea la fifo número `fifo_nr` con un tamaño de buffer de `size` bytes. Esta función **NO se puede invocar desde tareas de tiempo real** (sólo desde la inicialización de los módulos) ya que se invocan funciones del núcleo de Linux para reservar memoria (`vmalloc`). Esta función devuelve un valor negativo en caso de error; cero si la llamada se completó con éxito.

`rtf_destroy(fifo_nr)`

Marca como libre la fifo y libera la memoria utilizada. Puesto que utiliza la función `vfree` de Linux para liberar la memoria de la fifo, **tampoco se puede llamar desde `rtl-tasks`**. Normalmente se llama desde la función de terminación del módulo `cleanup_module()`.

9.6 Leer y escribir en fifos

`rtf_get(fifo_nr, *buf, n_bytes)`

Intenta leer `n_bytes` bytes de la fifo y los deposita en `buf`. El valor de retorno indica el número de bytes que realmente se han podido leer.

Si no hay datos en la fifo entonces retorna inmediatamente indicando que se han leído cero bytes. Es una operación no bloqueante.

`rtf_put(fifo_nr, *buf, n_bytes)`

Escribe `n_bytes` en la fifo. Si no hay espacio en la fifo escribir todos entonces la operación y finaliza con error `-ENOSPC` y no se escribe nada.

9.7 Control de estado

`rtf_isempty(fifo_nr)`

Está claro. ¿No? 😊. Devuelve `TRUE` si la FIFO está vacía y `FALSE` en caso contrario.

`rtf_flush(fifo_nr)`

- Vacía `fifo_nr`. Pone a cero el número de bytes contenidos. Los datos que pudiera haber en la fifo se pierden.
- Ejemplo de programación con fifos: [rt_fifo2.c](#) ([Makefile](#), [rtl.mk](#)). Nota: ajusta la variable `RTL_DIR` del fichero `rtl.mk` para que contenga el directorio donde están instalados los fuentes de RTLinux.

9.8 Manejadores de FIFOS

`rtf_create_handler(fifo_nr, function)`

Instala una función manejadora que se invoca cada vez que un proceso **normal de Linux** lee o escribe sobre la fifo indicada. Sólo se llama la función cuando se ha llevado a cabo la operación con éxito, p.e. una operación de lectura sobre una fifo vacía no causa la llamada del manejador hasta que se escribe y luego se completa la llamada de lectura.

Para desinstalar un manejador nuestro tenemos que instalar la función manejadora por defecto (`default_handler`): `rtf_create_handler(n_fifo, default_handler)`.

`rtf_create_rt_handler(fifo_nr, function)`

Instala una función manejadora que se invoca cada vez que alguna **tarea de tiempo real** lee o escribe sobre la fifo indicada. El manejador se desinstala de igual forma que el manejador anterior.

- En el directorio de la distribución [examples/frank/](#) de la distribución de RTLinux se puede encontrar un excelente ejemplo del uso de los manejadores de fifo para controlar tareas de tiempo real desde un proceso Linux ([frank_module.c](#)).

9.9 API POSIX.4

- RTLinux ha incorporado la capa POSIX.4 de abstracción de dispositivos, que se compone de los nombres de dispositivo, en este caso `/dev/rtfx`, y las clásicas funciones de trabajo con ficheros.
- Es importante tener claro que aunque el nombre de los ficheros y las operaciones que se utilizan tienen los mismos nombres que los utilizados en UNIX (POSIX) clásico, son dos implementaciones distintas.
- Para poder utilizar este API es necesario haber compilado RTLinux con la opción `CONFIG_RTL_POSIX_IO` (desde el menú de configuración: `make xconfig`).
- Utilizando esta interfaz se pueden escribir programas que compilen tanto como procesos Linux como tareas de RTLinux.

9.10 open y close

`fd=open("/dev/rtfx", flags)`

Antes de realizar esta operación es necesario haber creado la fifo llamando a `rtf_create()`. A efectos prácticos la función `open` no hace nada útil excepto devolver el descriptor de fichero y comprobar que hemos abierto el fichero en modo `O_NONBLOCK`, esto es, sólo podemos realizar operaciones no bloqueantes. En caso de error devuelve un valor negativo. Si se pudo abrir la FIFO entonces devuelve el descriptor de fichero correspondiente.

`close(fd)`

Existe por compatibilidad. Realmente no hace nada, sólo liberar las estructuras de datos relacionadas con el descriptor de fichero.

9.11 Read y write

`read(fd, *buf, size)`

Esta función es un *front end* para la función `rtf_get`. Retorna el número de bytes efectivamente leídos.


`write(fd, *buf, size)`

Front end a `rtf_put`.

9.12 Colas de mensajes

- Las colas de mensajes son un mecanismo de comunicación muy similar a las fifos.
- La principal diferencia entre una cola de mensajes y una fifo es que en la fifo no están delimitados los datos de cada operación de escritura, mientras que en una cola de mensajes en cada operación de escritura se define el tamaño de los datos y éste se conserva hasta la entrega (lectura).
- Otra diferencia es que podemos asignar atributos a los mensajes, por ejemplo el sistema puede tener los mensajes ordenados por prioridad en lugar de por orden de llegada.
- Si queremos utilizar las fifo como si fueran colas de mensajes tenemos que utilizar el módulo `rtl_nfifo` en lugar de `rtl_fifo`. Con este nuevo módulo podremos seguir utilizando los api's anteriores además del de

C.M..

- La implementación actual se hizo en 1999, no está completa y al API es poco consistente 
- Las funciones implementadas son:

```
mq_fast_send
mq_fast_send
mq_fast_create
mq_isempty
```

9.13

```
// Programa: rt_fifo.c
#include <linux/module.h>
#include <linux/kernel.h>
#include <rtl_fifo.h>
#include <rtl_core.h>

char *frase="Hola desde una rt-task\n";

int manejador(unsigned int fifo){
    printk("Puedo llamar a printk!!\n");
    return 0;
}

void tarea(char *arg){
    rtf_put(0, frase, strlen(frase));
    pthread_exit(0);
}

int init_module(void){
    if (rtf_create(0, 4096)) return -1;
    if (rtf_create_handler(0, &manejador))
        return -1;
    return 0;
}

void cleanup_module(void){
    rtf_destroy(0);
}
```

9.14

```
#include <linux/errno.h>
#include <rtl.h>
#include <time.h>

#include <rtl_sched.h>
#include <rtl_fifo.h>
#include "control.h"

pthread_t tasks[2];

static char *data[] = {"Frank ", "Zappa "};

#define TASK_CONTROL_FIFO_OFFSET 4

void *thread_code(void *t)
{
    int fifo = (int) t;
    int taskno = fifo - 1;
    struct my_msg_struct msg;
    while (1) {
        int ret;
        int err;
        ret = pthread_wait_np();
        if ((err = rtf_get (taskno + TASK_CONTROL_FIFO_OFFSET, &msg, sizeof(msg))) == sizeof(msg)) {
            rtl_printf("Task %d: executing the \"%d\" command to task %d; period %d\n", fifo - 1, msg.command, msg.taskno, msg.period);
            switch (msg.command) {
                case START_TASK:
                    pthread_make_periodic_np(pthread_self(), gethrtime(), msg.period * 1000);
                    break;
            }
        }
    }
}
```

```

        case STOP_TASK:
            pthread_suspend_np(pthread_self());
            break;
        default:
            rtl_printf("RTL task: bad command\n");
            return 0;
    }
}
rtf_put(fifo, data[fifo - 1], 6);
}
return 0;
}

int my_handler(unsigned int fifo)
{
    struct my_msg_struct msg;
    int err;

    while ((err = rtf_get(COMMAND_FIFO, &msg, sizeof(msg))) == sizeof(msg)) {
        rtf_put(msg.task + TASK_CONTROL_FIFO_OFFSET, &msg, sizeof(msg));
        rtl_printf("FIFO handler: sending the \"%d\" command to task %d; period %d\n", msg.command,
            msg.task, msg.period);
        pthread_wakeup_np(tasks[msg.task]);
    }
    if (err != 0) {
        return -EINVAL;
    }
    return 0;
}

/* #define DEBUG */
int init_module(void)
{
    int c[5];
    pthread_attr_t attr;
    struct sched_param sched_param;
    int ret;

    rtf_destroy(1);
    rtf_destroy(2);
    rtf_destroy(3);
    rtf_destroy(4);
    rtf_destroy(5);
    c[0] = rtf_create(1, 4000);
    c[1] = rtf_create(2, 4000);
    c[2] = rtf_create(3, 200); /* input control channel */
    c[3] = rtf_create(4, 100); /* input control channel */
    c[4] = rtf_create(5, 100); /* input control channel */
    printk("Fifo return 1=%d 2=%d 3=%d\n", c[0], c[1], c[2]);

    pthread_attr_init(&attr);
    sched_param.sched_priority = 4;
    pthread_attr_setschedparam(&attr, &sched_param);
    ret = pthread_create(&tasks[0], &attr, thread_code, (void *)1);

    pthread_attr_init(&attr);
    sched_param.sched_priority = 5;
    pthread_attr_setschedparam(&attr, &sched_param);
    ret = pthread_create(&tasks[1], &attr, thread_code, (void *)2);

    rtf_create_handler(3, &my_handler);
    return 0;
}

void cleanup_module(void)
{
#ifdef DEBUG
    printk("%d\n", rtf_destroy(1));
    printk("%d\n", rtf_destroy(2));
    printk("%d\n", rtf_destroy(3));
    printk("%d\n", rtf_destroy(4));
    printk("%d\n", rtf_destroy(5));
#else
    rtf_destroy(1);
    rtf_destroy(2);
    rtf_destroy(3);
    rtf_destroy(4);
    rtf_destroy(5);
#endif
}

```

```
pthread_cancel (tasks[0]);  
pthread_join (tasks[0], NULL);  
pthread_cancel (tasks[1]);  
pthread_join (tasks[1], NULL);  
}
```

10 Memoria compartida

Ismael Ripoll
iripoll@disca.upv.es
<http://bernia.disca.upv.es/~iripoll>

Permission is granted to copy, distribute and/or modify this document under the terms of the [OpenContent License](#).

10.1 Introducción

- Las *rt-tasks* se ejecutan en el mismo espacio de memoria que el núcleo de Linux. Todas las *rt-tasks* comparten el mismo espacio de memoria. Las funciones *mmap* sirven para poder compartir zonas de memoria entre procesos de Linux y tareas de *RTL*.
- *mmap* permite compartir zonas de memoria arbitrariamente grandes, siempre y cuando el sistema disponga de suficiente RAM.
- El API utilizado por *mmap* no sigue ningún estándar. Si bien el mecanismo de memoria compartida de POSIX.4 (*shm_open*, *shm_close* y *mmap*) es bastante similar.
- La zona de memoria compartida mediante *mmap* siempre estará en memoria principal, no puede sufrir *swap-out*.
- *mmap* utiliza *mmap* para obtener una zona de memoria, lo cual implica:
 1. No se pueden invocar las funciones de *mmap* desde una tarea de tiempo real (o manejador de interrupción de *RTL*).
 2. La zona de memoria obtenida puede que **no sea contigua en memoria física**. Hay que recordar que Linux utiliza la paginación para gestionar la memoria. Las direcciones "lineales" con las que trabajan los procesos pasan por el sistema de paginación antes de salir del procesador y llegar a memoria principal.

De hecho uno de los principales requisitos para poder implementar un sistema UNIX es disponer del hardware para poder implementar memoria virtual.

10.2 Memoria física contigua

- El núcleo de Linux dispone de la función *mmap* para reservar zonas de memoria físicamente contiguas, pero con la restricción de que el bloque más grande que permite reservar es de 128Kb.
- Linux utiliza un sistema de gestión de memoria total. La idea que motiva este tipo de gestión es que tener páginas de memoria sin usar es equivalente a desaprovecharlas. Sólo se libera memoria cuando se necesita. El resultado es que siempre se tiene en memoria lo último que se ha hecho. Por ejemplo, la segunda vez que se ejecuta un programa no es necesario acceder a disco para leer el ejecutable.
- Este tipo de gestión provoca un elevado nivel de fragmentación de la memoria física. Incluso realizando una fuerte liberación de memoria (enviando procesos a *swap* y liberando páginas de la cache de disco) no se suelen conseguir zonas de memoria contigua mucho mayores. El problema reside en que el algoritmo de liberación de memoria no utiliza criterios de "situación" de las páginas sino de "utilidad".
- Tenemos dos opciones para conseguir zonas de memoria contiguas:
 1. Forzar que Linux reconozca menos memoria RAM de la que realmente hay instalada. Esta solución se denomina "memoria alta no utilizada".
 2. Utilizar el parche [bighphysarea](#), que consiste en reservar durante el arranque tanta memoria física consecutiva como se quiera y luego gestionarla al margen de Linux.

10.3 Listado de funciones

Para procesos Linux
mmap_alloc
mmap_free

```
mbuff_attach
mbuff_detach
```

Para threads de RTLinux

```
mbuff_alloc
mbuff_free
```

10.4 Uso de mbuff desde Linux

- El driver de mbuff se ha implementado como un fichero especial: `/dev/mbuff` (mayor=10, minor=254). Y las operaciones se realizan mediante las llamadas al sistema `mmap`, `open` y `ioctl`. De todas formas se dispone de un fichero de cabecera que simplifica enormemente su uso.
- En principio mbuff se desarrolló para comunicar procesos Linux con threads RTLinux, pero también se puede utilizar para establecer zonas de memoria común entre procesos Linux.
- Las funciones están declaradas en el fichero `drivers/mbuff/mbuff.h`. Es curioso observar como en este mismo fichero se definen estas funciones tanto para procesos Linux como para RTLinux mediante directivas del preprocesador.

10.5 Crear y liberar memoria

```
void *mbuff_alloc(name, size)
```

Si no existe ninguna zona con el nombre *name* entonces crea una zona de tamaño *size* y devuelve la dirección de inicio de la zona; si ya existe esa zona entonces devuelve el puntero a la misma, en este caso es necesario que *size* sea menor o igual que el tamaño de la zona ya existente.

```
mbuff_free(name, *buff)
```

Indica que no se va a seguir usando la zona *name* (*buff* ha de ser el puntero a devuelto por la llamada `mbuff_alloc` correspondiente, aunque actualmente no se utiliza). Cuando no hay procesos o tareas usando una zona entonces la zona se libera.

```
void *mbuff_attach(name, size)
```

Equivalente a `mbuff_alloc` se libera automáticamente al finalizar la ejecución del programa.

```
mbuff_detach(name, *buff)
```


El equivalente a `mbuff_free` cuando la reserva de ha realizado con `mbuff_attach`.

- En los propio directorio de mbuff se puede encontrar un sencillo ejemplo de uso: [demo.c](#).

10.6 Utilización de mbuff desde RLinux

- Para poder utilizar mbuff desde RTLinux es necesario tener insertado el módulo mbuff, e incluir el fichero de cabecera `drivers/mbuff/mbuff.h` en los fuentes que lo utilicen.
- El API de mbuff en RTLinux es el mismo que para procesos Linux: `mbuff_alloc` y `mbuff_free`. El nombre de las funciones, así como los parámetros y su uso es el mismo que el ya explicado, pero las funciones son distintas.
- No se pueden utilizar las funciones `mbuff_attach` y `mbuff_detach`.
- Sólo hay que recordar que *no se puede llamar a las funciones `mbuff_alloc` y `mbuff_free` desde tareas o interrupciones de tiempo real*.

10.7 Memoria alta no usada

- Es un método un tanto "truculento"... 
- Por defecto, al arrancar Linux determina la cantidad de memoria instalada consultando la BIOS.
- Pero mediante un parámetro de arranque se le puede indicar a Linux la cantidad exacta de memoria que queremos que utilice.
- Algunas BIOS no son capaces de determinar correctamente la cantidad de RAM instalada, informando que hay menos de la real. Con este parámetro podemos corregir este problema.

- Si forzamos Linux a utilizar menos memoria de la real, entonces la zona alta queda libre para cualquier uso.
- El procesador Pentium (y posteriores) permite definir páginas de dos tamaños: 4Mb y 4Kb. Linux utiliza páginas de 4Mb para mapear toda la memoria física de forma que las direcciones de memoria lineal coincidan con la física. Luego esas direcciones pasarán a formar parte también de la memoria virtual de otros procesos o del propio núcleo.
- Suponiendo que tenemos 64Mb, podemos indicar a Linux que sólo tenemos 61Mb, y entonces los tres últimos megas de la última página están contenidos en la página número 16 (páginas de 4Mb), pero no utilizados por Linux. Esto se puede hacer añadiendo la siguiente línea en el fichero [/etc/lilo.conf](#):

```
append="mem=61M"
```

10.8 Memoria alta desde RTLinux

- Acceder a esta zona de memoria es inmediato desde RTLinux. Sólo tenemos que declarar un puntero y hacer que apunte a la dirección 61Mb:

```
char *ptr;
ptr = (61 * 0x100000);
*ptr = 'x';
```

Este ejemplo escribe una "x" en el primer byte del mega 61.

10.9 Memoria alta desde Linux

- Desde un proceso Linux es necesario utilizar el fichero especial `/dev/mem` y mapear 3Mb a partir del 61 de ese fichero:

```
char *ptr;
int fd = open("/dev/mem", O_RDWR);
ptr = mmap(NULL, (3 * 0x100000),
            PROT_READ|PROT_WRITE,
            MAP_SHARED, fd, (61 * 0x100000));
printf("%c\n", *ptr);
```

- El prototipo de `mmap` es:
`void * mmap(void *start, length, prot, flags, fd, offset);`
- Este método no se puede utilizar con procesadores i486 y anteriores pues sus páginas siempre son de 4Kb.

10.10 Bigphysarea

- Al arrancar Linux se ejecutan las funciones de inicialización de todos los "drivers". Cada driver reserva los recursos que precisa: rangos de puertos, interrupciones, zonas de memoria, etc. La forma de reservar memoria durante esta fase es mucho más fácil que durante el funcionamiento normal y sobre todo la memoria reservada es contigua y puede ser tan grande como el driver requiera.
- `bigphysarea` es un driver que reserva inicialmente una cantidad de memoria y luego la gestiona sin utilizar ningún mecanismo de memoria paginación. Los bloques de memoria que devuelve siempre son contiguos en memoria física. Las funciones para obtener y liberar memoria son: `bigphysarea_alloc_pages` y `bigphysarea_free_pages`.
- `Bigphysarea` es una ampliación al núcleo de Linux que hay que instalar como un parche. Las modificaciones sobre el fuente de Linux son mínimas. Como todo parche, se necesita recompilar el núcleo y luego arrancar con el nuevo núcleo.
- Al igual que sucedía con la memoria alta no usada, se le tiene que pasar el núcleo de Linux un parámetro indicando el tamaño de la memoria a reservar y éste la reserva durante el arranque.
- Una ventaja de `bigphysarea` frente a la memoria alta no asignada es que por regla general sobre la primera se podrán realizar operaciones de DMA (ya que la memoria quedará dentro de las direcciones de trabajo de la DMA, 16Mb) mientras que con la memoria alta no se podrá al quedar fuera el rango de direcciones alcanzable por el sistemas de DMA.

- Estas dos funciones sólo se pueden utilizar dentro del núcleo, por lo que si queremos compartir con procesos de usuario esta zona de memoria será necesario habilitar algún método para
- Es posible monitorizar la cantidad de memoria utilizada de `bigphysarea` a través del fichero especial `/proc/bigphysarea`.

`caddr_t bigphysarea_alloc_pages(count, align, prio)`

Devuelve la dirección de una zona de memoria compuesta por `count` páginas (4Kb) alineadas a un múltiplo de `align` páginas. El parámetro `prio` únicamente tiene utilidad la primera vez que se llama a esta función.... utilizar siempre `GFP_KERNEL`.

La primera vez que se llama a esta función puede tardar algo de tiempo ya que puede necesitarse liberar memoria "normal" del núcleo.

`bigphysarea_free_pages(base)`

Libera la memoria reservada previamente mediante una llamada a `bigphysarea_alloc_pages`.

10.11

```
#include <stdio.h>

#include "mbuff.h"

/* the contents of shared memory may change at any time, thus volatile */
volatile char * shm1, *shm2;

main (int argc, char *argv[]){

    shm1 = (volatile char*) mbuff_alloc("demo1", 1024*1024);
    shm2 = (volatile char*) mbuff_alloc("demo1", 1024*1024);
    if( shm1 == NULL || shm2 == NULL ) {
        printf("mbuff_alloc failed\n");
        exit(2);
    }
    sprintf((char*)shm1, "example data\n");
    sleep(5); /* you may change it from the kernel or other program here */
    printf("shm1=%p shm2=%p shm2->%s", shm1, shm2, shm2);
    mbuff_free("demo1", (void*)shm1);
    sleep(3);
    /* you may still access shm2 here, it is still the same memory area */
    mbuff_free("demo1", (void*)shm2);
    return(0);
}
```

11 Herramientas de depuración

Ismael Ripoll
iripoll@disca.upv.es
<http://bernia.disca.upv.es/~iripoll>

Permission is granted to copy, distribute and/or modify this document under the terms of the [OpenContent License](#).

11.1 Introducción

- El método más utilizado para depurar programas de bajo nivel consiste en incluir líneas de traza, esto es, sentencias que impriman o que causen algún efecto reconocible por el programador como sonidos o estado de puertos E/S.
- RTLinux dispone de tres funciones para imprimir por pantalla: `conpr`, `conprn` y `rtl_printf`.
- Evidentemente también se pueden utilizar todos los mecanismos de comunicación estudiados (mbuff, fifo) para conocer cómo se ejecutan las tareas de tiempo real.

11.2 Imprimir directo a consola

- Las funciones para imprimir a consola están implementadas en el fichero `rtl_core.c` y declaradas en el fichero de cabecera `include/rtl_core.h`. Las funciones son:

```
conpr
conprn
```

- Estas funciones imprimen directamente sobre la memoria de vídeo, pero sólo cuando la máquina está en modo texto. No se realiza ningún tipo de buffering. Sólo se imprime sobre la consola de texto que en ese momento esté activa. Por tanto, si cuando se invocan estas funciones se está en modo gráfico entonces no se imprimirá nada.

- La salida por pantalla de estas ordenes se realiza en el mismo momento en que se realiza la llamada.

`conpr(cadena)`

Imprime la cadena de caracteres `cadena`. No realiza ningún tipo de formateado de la cadena.

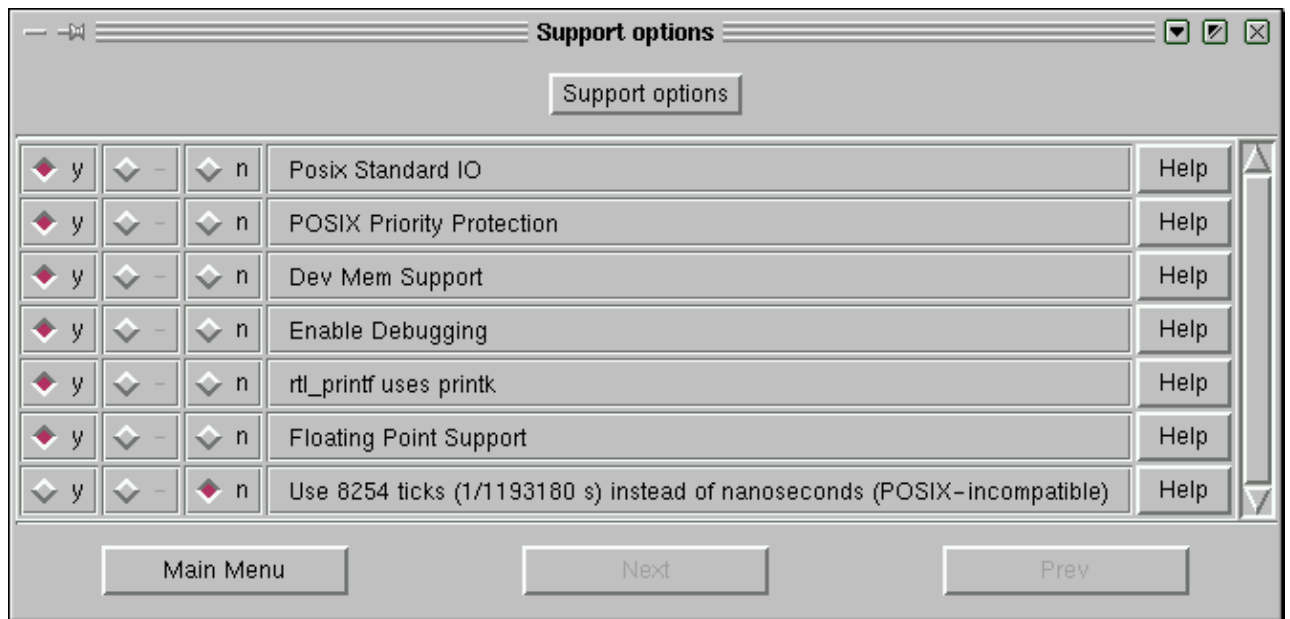
Si cuando se invoca a esta función no está activa ninguna las consolas, entonces no se imprime nada. Estando en modo gráfico no produce ninguna salida.

`conprn(número)`

Imprime el `número` (de tipo `unsigned int`) en formato hexadecimal. Esta función se sirve de `conpr` para imprimir, por lo que se le aplican las mismas restricciones.

11.3 Imprimir mediante printf

- La función `rtl_printf` está declarada en el fichero de cabecera `include/rtl_printf.h`, aunque no es necesario incluirla en nuestros programas ya que se incluye automáticamente desde `include/rtl_core.h`.
- `rtl_printf` es una función muy similar a la función `printf` de la biblioteca de "C", con la posibilidad de formatear muchos tipos de datos (punteros, enteros, caracteres, etc.).
- Durante la configuración (

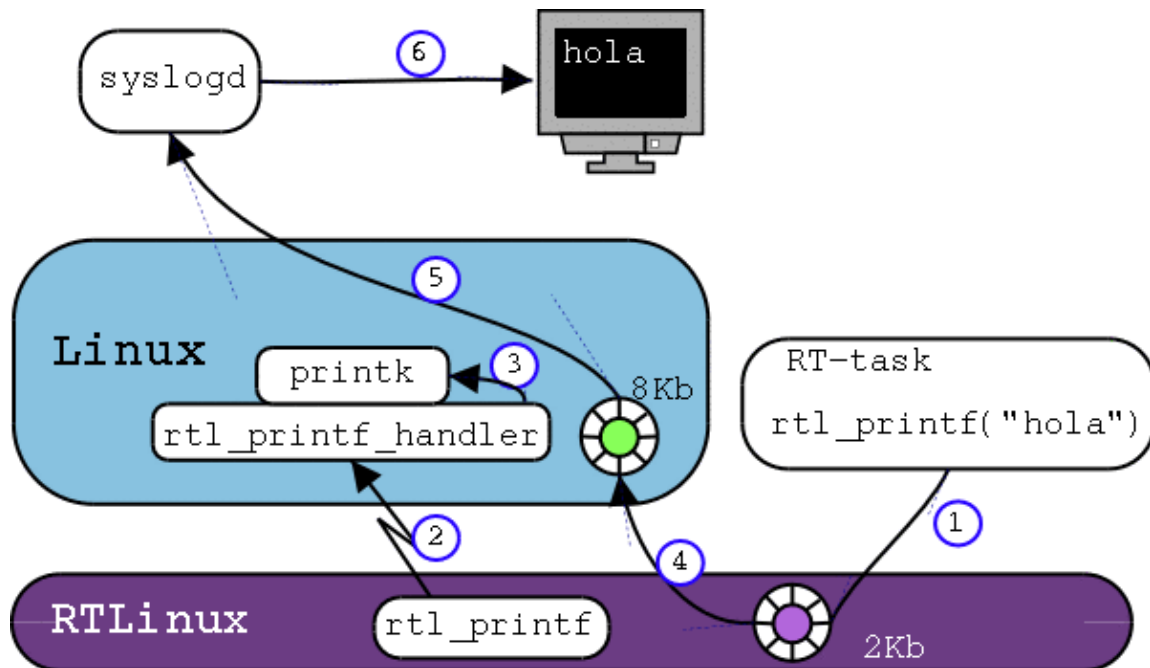


) de los fuentes de RTLinux se puede elegir entre dos formas de funcionamiento de `rtl_printf`:

- ♦ a través de `printk` (Linux)
 - ♦ o directamente a consola.
- Operando a través de `printk`, `rtl_printf` se comporta como un *front end* a `printk`. Recordemos que `printk` es la función de Linux que permite imprimir mensajes desde el núcleo de Linux. Estos mensajes son enviados al demonio de "log" (`syslogd`) para registrarlos. Por otra parte el núcleo mantiene un buffer con los últimos mensajes enviados por el núcleo, los cuales se pueden visualizar con la orden `dmesg`.

El importante resaltar que en este método el núcleo de Linux es el que realmente lleva a cabo la impresión, por tanto, si Linux no la tiene posibilidad de ejecutarse entonces no se imprimirán los mensajes de RTLinux.

- Si elegimos el funcionamiento directo a consola entonces `rtl_printf` formatea los parámetros que se le pasan y llamará a la función `conpr`.
- La cadena más larga que podemos imprimir es de 1970 caracteres. Éste es el espacio máximo de la cadena que se imprima, incluidas todas las expansiones de los formatos. **No se comprueba el desbordamiento de este límite**.



`rtl_printf(cadena, ...)`

Su funcionamiento es el mismo que la función `printf` de la biblioteca de "C" con la excepción de que no trabaja con números en coma flotante (no se acepta `%f` ni `%lf`). En las versiones del núcleo anteriores a la 2.3.x tampoco se podía imprimir números de tipo `long long`.

Si la impresión se ha podido realizar con éxito entonces devuelve el número de caracteres que se han imprimido. Si no hay espacio en el buffer intermedio para almacenar la cadena hasta que pueda ser imprimida desde Linux, entonces no se imprime nada y retorna con el valor cero.

Si necesitamos imprimir números en coma flotante podemos enviarlos por una FIFO a un proceso Linux y que éste los formatee y los imprima.

12 FAQ

Esta página contiene es un listado (no ordenador ni organizado) de detalles "poco agradables" que suelen surgir al programar con RTLinux.

- Recuerda que hay disponible hoja de manual en el propio Linux de la mayoría de las funciones del estándar PThreads. Estas hojas de manual describen el funcionamiento de los PThreads a nivel de proceso Linux, ya que fueron creadas por Xavier Leroy (autor de la biblioteca LinuxThreads) y no por los desarrolladores de RTLinux. A pesar de ello, la mayoría de la información de estas hojas sigue siendo válida para RTLinux, ya que ambos casos son una implementación del estándar POSIX (1003.1b).
- No se pueden crear nuevas tareas de tiempo real (pthread) desde una tarea de tiempo real. Para crear una tarea hay que "pedir" memoria al núcleo de Linux, lo cual no es recomendable desde una rtl-task.
- En todo este tutorial he utilizado la palabra "rtl-task" para denominar los threads creados por RTLinux mediante la llamada `pthread_create`.
- La llamada a `pthread_cleanup_pop()` se tiene que realizar en el mismo nivel sintáctico en el que se hizo la llamada `pthread_cleanup_push()`.
- Las rtl-tasks se ejecutan en el mismo espacio de memoria que el núcleo de Linux. Por ello, un bug de programación como puede ser un puntero no inicializado o salirse de un vector puede dejar todo el sistema inestable.
- Los ficheros [GettingStarted.txt](#) y [FAQ](#) que acompañan la distribución son una excelente fuente de información.
- [RT-Linux functions and their possible context](#) (fichero `local`).
- El directorio [semaphores](#) de los fuentes de RTLinux contiene una implementación realizada por Jerry Epplin de varios mecanismos de comunicación entre procesos. Esta implementación no sigue ningún estándar si bien la semántica es similar a POSIX.4. En la versión 3.0pre6d ya se han dispone de semáforos que siguen el estándar POSIX.
- El fichero de cabecera [asm/io.h](#) contiene la declaración de todas las macros y funciones inline para acceder a los puertos de E/S. Para que estas macros se expandan correctamente es necesario compilar los programas con la opción de optimización "-O2". De hecho es recomendable utilizar siempre esta opción de compilación, aunque se luego se quiera depurar pues no es incompatible con la de depuración "-g".
- La opción de compilación "-fomit-frame-pointer" fuerza a generar código que no utilice el registro EBP (Enhanced Base Pointer), lo que agiliza las llamadas a funciones. En cada llamada a función se realizan tres instrucciones máquina menos. Por contra, el depurador gdb depende de este registro, por lo que programas compilados con esta opción so se podrán depurar.
- Es posible enlazar "linkar" varios ficheros objeto en uno solo utilizando la opción "-i" con el linker (ld). También se puede utilizar el linkado incremental para resolver símbolos de bibliotecas. Por ejemplo:
`ld -i -o final.o no_loadable.o /usr/lib/libc.a`

Genera el fichero `final.o` con todas las refencias a la biblioteca `libc.a` resueltas.

© Jose Ismael Ripoll Ripoll (disca), Abril 2000 [Opencontent License](#)

[Ismael Ripoll](#) Last modified: Wed May 16 13:45:15 CEST 2001

13 Licencia

OPENCONTENT

[Home](#) | [Open Content License v1.0](#) | [Open Publication License v1.0](#)
[Content Database](#) | [Open Source Content Development](#) | [Questions and Feedback](#)

OpenContent License (OPL)

Version 1.0, July 14, 1998.

This document outlines the principles underlying the OpenContent (OC) movement and may be redistributed provided it remains unaltered. For legal purposes, this document is the license under which OpenContent is made available for use.

The original version of this document may be found at <http://opencontent.org/opl.shtml>

LICENSE

Terms and Conditions for Copying, Distributing, and Modifying

Items other than copying, distributing, and modifying the Content with which this license was distributed (such as using, etc.) are outside the scope of this license.

1. You may copy and distribute exact replicas of the OpenContent (OC) as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the OC a copy of this License along with the OC. You may at your option charge a fee for the media and/or handling involved in creating a unique copy of the OC for use offline, you may at your option offer instructional support for the OC in exchange for a fee, or you may at your option offer warranty in exchange for a fee. You may not charge a fee for the OC itself. You may not charge a fee for the sole service of providing access to and/or use of the OC via a network (e.g. the Internet), whether it be via the world wide web, FTP, or any other method.

2. You may modify your copy or copies of the OpenContent or any portion of it, thus forming works based on the Content, and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified content to carry prominent notices stating that you changed it, the exact nature and content of the changes, and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the OC or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License, unless otherwise permitted under applicable Fair Use law.

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the OC, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the OC, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. Exceptions are made to this requirement to release modified works free of charge under this license only in compliance with Fair Use law where applicable.

3. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to copy, distribute or modify the OC. These actions are prohibited by law if you do not accept this License. Therefore, by distributing or translating the OC, or by deriving works herefrom, you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or translating the OC.

NO WARRANTY

4. BECAUSE THE OPENCONTENT (OC) IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE OC, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE OC "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK OF USE OF THE OC IS WITH YOU. SHOULD THE OC PROVE FAULTY, INACCURATE, OR OTHERWISE UNACCEPTABLE YOU ASSUME THE COST OF ALL NECESSARY REPAIR OR CORRECTION.

5. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MIRROR AND/OR REDISTRIBUTE THE OC AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE OC, EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

[Home](#) | [Open Content License v1.0](#) | [Open Publication License v1.0](#)
[Content Database](#) | [Open Source Content Development](#) | [Questions and Feedback](#)

OPENCONTENT

