Programación en Tiempo Real

RT-Linux, Sistema Operativo en Tiempo Real

http://www.vision.uji.es/~pla/ii75

Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Introducción

- Partió del trabajo de M. Barabanov y V. Yodaiken en la Unv. de Nuevo Mexico.
- Se distribuye bajo “GNU Public License”.
- Funciona sobre arquitecturas:
  - PowerPC
  - i386
  - se está desarrollando para Alpha.
- La versión 1.0 ofrecía una API reducida sin tener en cuenta estándares POSIX.
- A partir de la v2.0 se convierte la API:
  - Compatible POSIX threads.

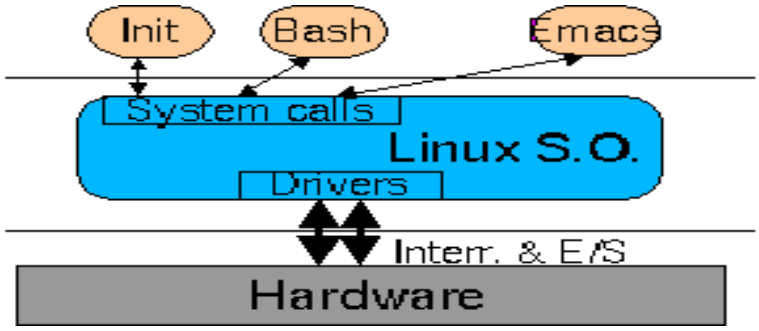
Introducción

- RTLinux NO es código independiente:
  - NO es una nueva versión de Linux.
  - Parte es un “parche” sobre el código del kernel de Linux.
  - La otra parte son módulos cargables.
  - Cada versión RTLinux está diseñada para cada versión del Linux:
    - Ejemplo. RTLinux v3.0 necesita Linux-2.3.48 o superior.
- POSIX
  - Desarrollado IEEE:
    - Estándar ANSI e ISO
  - Extensiones en RT:
    - POSIX.4 (POSIX 1003.b)
  - Basado en UNIX

Introducción

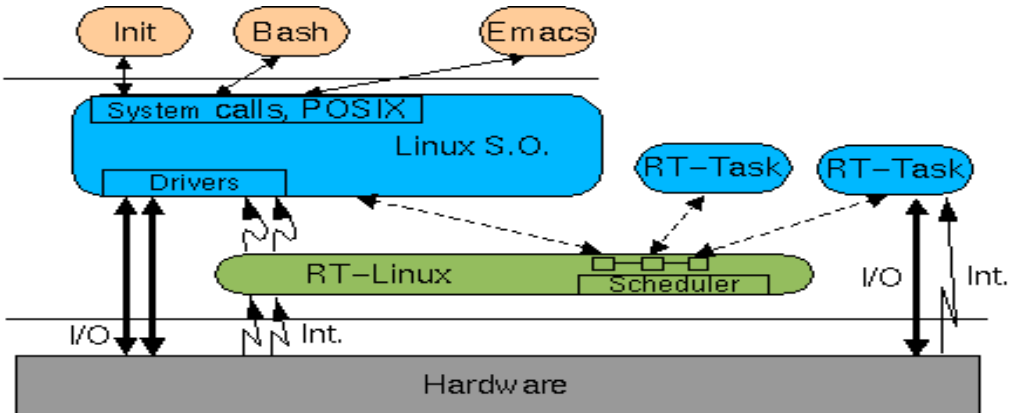
- *OpenGroup* propietario de UNIX:
  - Unificación de estándares UNIX en SUS (*Single UNIX Specification*).
- Objetivo de RTLinux:
  - Conservar servicios característicos de Linux.
  - Permitir funciones RT en entorno de baja latencia.
  - Flexibilidad.
- Estrategia de implementación:
  - Pequeño núcleo RT coexiste con POSIX Linux.

Introducción



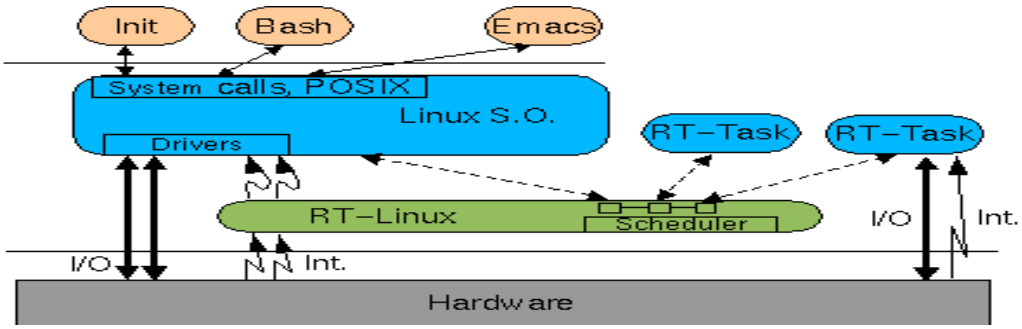
- SO: programa que gestiona los recursos del sistema (modo supervisor).
- Interfaz de acceso de los procesos y aplicaciones a los recursos del sistema:
  - Servicios del SO.
  - Aplicaciones en “modo usuario”.
- Organizar la ejecución de procesos:
  - Planificación.

Introducción



- No añade nuevas llamadas al sistema ni modifica existentes.
- Crea una maquina virtual entre el hardware y el Linux.
- RTLinux toma el control de las interrupciones:
  - Implementa un gestor de interrupciones por software.

Introducción



- Aplicaciones RT:
  - Tareas RT en módulos cargables.
  - Gran flexibilidad: Extensible
  - Modificaciones en tiempo de ejecución
- Linux: una tarea más
- Procesos Linux:
  - funciones no restringidas a RT
    - visualización
    - acceso red, etc.

Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Características básicas

- Características básicas:
  - Tareas con memoria estática:
    - Sin paginación.
    - Sin protección de direcciones de memoria.
  - Planificador de prioridades estáticas:
    - Sin protección a planificaciones imposibles.
    - Otros posibles planificadores (EDF, RMS,...)
  - Acceso directo al hardware:
    - Puertos E/S, memoria, ...
    - Permite deshabilitación de interrupciones hardware.
  - Memoria compartida:
    - Comunicación entre tareas.
    - Comunicación entre tareas RT y procesos Linux.

Características básicas

- Características básicas (continuación):
  - Colas FIFO para comunicación con procesos LINUX.
  - LINUX: tarea de más baja prioridad.
  - No hay protección de sobrecargas del procesador:
    - Tareas con más baja prioridad “se cuelgan” (Linux).
  - No se pueden usar fácilmente *drivers* de Linux.
  - API del tipo POSIX *threads* (a partir v2.0):
    - Señales, Sistema de archivos POSIX, Semáforos, Variables condición.
  - Eficiente gestión de tiempos.
  - Estructura modular.
  - Facilidad para incorporar nuevos componentes:
    - Relojes, dispositivos E/S, planificadores, ..

Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Instalación de RTLinux

- Información actualizada y ficheros de la distribución:
  http://www.rtlinux.org
  http://fsmlabs.com (ftp://ftp.fsmlabs.com)
- Se "transforma" el núcleo del Linux en RTLinux:
  - Se instala como parche del núcleo.
  - Re-compilar núcleo.
- Hay una versión de los fuentes con el parche ya instalado.
- Instalación y compilación de un núcleo cualquiera de Linux.
- Configurar el "lilo" y arrancar con el nuevo núcleo.
- Una vez compilado el núcleo:
  - Compilar módulos adicionales.
  - Compilar ejemplos.

Instalación RTLinux

- Guía de instalación:
  http://fsmlabs.com/developers/man_pages
  /installation_june_2000.htm
- El paquete de instalación lleva también documentación:
  - Páginas *man*.
  - Otros documentos también disponibles en la página web.
- En prácticas:
  - RTLinux v3.1 sobre núcleo 2.4.4
  - Ficheros y documentación disponible en página web:
    http://www.vision.uji.es/~pla/ii75

Instalación RTLinux

- Configuración del lilo (aplicación de arranque):
  - Se puede configurar para arrancar en varios modos (varios núcleos)
  - Editar fichero `/etc/lilo.conf`
  - Ejemplo:

```
...
Image=/vmlinuz.rt
    label=RT-Linux
    ...
Image=/vmlinuz.linux
    label=Linux
...
```

- Instalar el nuevo *kernel*:
  `/sbin/lilo`
- Reiniciar el sistema:
  `/sbin/reboot`

Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Módulos

- Las tareas RT en RTLinux se implementan como módulos del núcleo.
- Los módulos son una característica opcional de Linux cuando se compila el núcleo.
- Definición:

  Partes del SO que se pueden insertar o extraer en tiempo de ejecución
- Características:
  - En forma de ficheros objeto.
  - Se compilan por separado.
  - El SO los enlaza y resuelve referencias.
  - Se pueden crear y cargar sin necesidad de recompilar el núcleo.
  - Pueden utilizar todas las funciones y acceder a todas las variables y estructuras de datos del núcleo.

Módulos

- Características (continuación):
  - El código se ejecuta con el máximo privilegio del procesador:
    - se puede realizar cualquier tipo de E/S.
    - Ejecutar instrucciones privilegiadas.
  - La memoria (programa y datos) esta mapeada directamente con la memoria física y no se puede hacer “paging”.
- Creación de módulos:
  - Programa C con funciones de inicialización y descarga.

Ejemplo módulo

```
/* ejemplo.modulo.c */

/* #define MODULE */
/* #define __KERNEL__ */
#include <linux/module.h>
#include <linux/kernel.h>

static int output=1;
MODULE_PARM(output,”i”);

int init_module(void)
{ printk("Output= %d\n",output);
  return 0;
}

void cleanup_module(void)
{ printk("Adeu!\n");
}
```

Compilar
`gcc –DMODULE -D__KERNEL__ -c ejemplo1.c`

Módulos

- **`printk()`**
  - El núcleo no dispone de salida estándar.
  - Funciona como `printf()` pero escribiendo en un buffer circular de mensajes (*kernel ring buffer*).
  - Para ver contenido del buffer, utilizar la orden:

    `dmesg`
  - o consultar el fichero

    `/proc/kmsg`
  - Salida por la consola en pantalla de texto.
  - No sale por pantalla cuando se está en modo gráfico.

Módulos

- En RTLinux es mejor utilizar:
  `rtl_printf()`
  - funciona en rutinas de interrupción y *threads*.

- Carga de módulos
  `insmod ejemplo1.o`
- Descarga de módulos
  `rmmod ejemplo1`
- Listado de módulos instalados en el núcleo:
  `lsmod`

Módulos

- Información sobre un módulo
  `modinfo ejemplo1.o`
- Automatiza/facilita la gestión de módulos
  `modprobe`

- Paso de parámetros
  - Asignación de valores a variables globales a través de parámetros en comando `insmod`

```
insmod ejemplo1.o variable=valor
```

Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Creación de tareas RT

- Las tareas RT en RTLiunx, a partir de la v2.0, se realizan a través de *threads*.
- El concepto es que cada tarea se implementa mediante un *thread*.
- RTLinux adopta el estándar POSIX para los *threads*.
- Además añade ciertas funcionalidades NO POSIX o No Portables (NP):
  - Ejemplo: creación de *threads* periódicos para implementación de tareas RT periódicas
- Los *threads* se pueden crear y destruir dinámicamente.
- El planificador (*scheduler*) gestiona cada *thread* de acuerdo a sus propiedades o atributos.
- Existen mecanismos para sincronización y envío de señales a los *threads* (tareas RT).

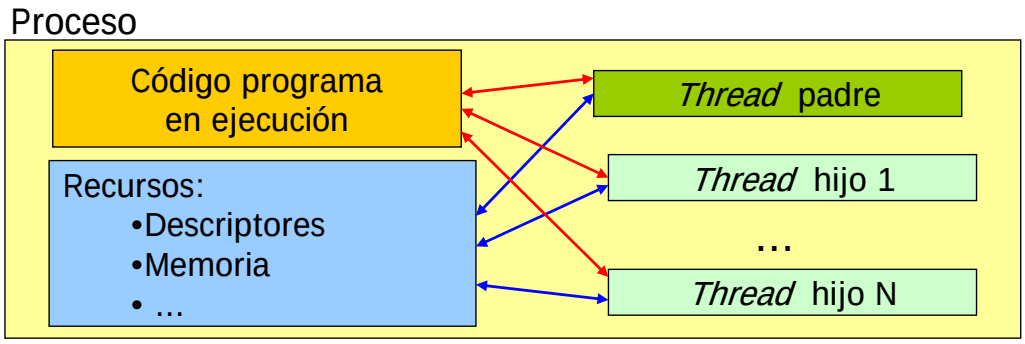
Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Threads en POSIX

- Proceso UNIX:
  - Código de programa en ejecución
  - Conjunto de recursos:
    - Tabla descriptor de ficheros.
    - Espacio de direcciones de memoria.
- *Thread* (hilo):
  - Manejan todas las actividades asociadas con la ejecución.
  - Constan básicamente de:
    - Contador de programa.
    - Pila.
    - Conjunto de registros y estado.
  - Un proceso puede tener varios *threads*.
  - Todos los *threads* dentro de un proceso comparten los mismos recursos.

Threads en POSIX



- Un proceso UNIX normal se puede considerar como un conjunto de recursos y un *thread,* el *thread* padre.
- Los *threads* hijos arrancan a partir del *thread* padre y comparten todos los mismos recursos del proceso.
- Todos los *threads*, padre e hijos, son iguales y se rigen por los mismos principios.

Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Creación de *threads*

- En general, para utilización de funciones relacionadas con *threads*:
  - Fichero cabecera: `pthread.h`
  - Nomenclatura: `pthread_nombre()`
- Inicializa la ejecución de un *thread* que se inicia inmediatamente.

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

- Devuelve el identificador el *thread* desde donde se realiza la llamada a esta función.

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


Creación de *threads*

- Atributos de threads:

```
int pthread_attr_init(pthread_attr_t *attr);
int phtread_attr_destroy(pthread_attr_t *attr);
```

- Los atributos de los threads se definen mediante la estructura `pthread_attr_t`. El acceso a los campos se realiza por medio de funciones como:

```
pthread_attr_setstacksize(pthread_attr_t *attr,
                          size_t size)

pthread_attr_get_schedparam
pthread_attr_set_schedparam(pthread_attr_t *attr,
                      const struct sched_param *param)
```

Establece la prioridad del *thread*. El rango de prioridades (0 a MAXINT) se obtiene mediante las funciones.

```
int sched_get_priority_min(int policy);
int sched_get_priority_max(int policy);
```

Policy: `SCHED_FIFO` (única implementada en RTLinux 3.0)

Ejemplo creación *threads*

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

/* funcion que representa el thread */
void print_message(void *ptr)
{ printf("%s ",(char*)ptr);
}

main()
{ pthread_t thread1,thread2;
  char *message1="Hello";
  char *message2="World";

pthread_create(&thread1,
               pthread_attr_default,
               (void*)&print_message,
               (void*)message1);

pthread_create(&thread2,
               pthread_attr_default,
               (void*)&print_message,
               (void*)message2);
exit(0);
}
```


Ejemplo creación *threads*

- `pthread_attr_default` representa la constante `NULL`, para asignar al *thread* los atributos por defecto.
- Los dos *threads* hijos se ejecutan de forma concurrente:
  - no hay garantía de que el `thread1` acabe antes del `thread2`.
  - Se debería utilizar algún mecanismo de sincronización para asegurar que un *thread* muestra el mensaje antes.
- La función `exit()` termina el proceso, y por tanto el *thread* padre, con lo que a su vez se destruyen todos los *threads* hijos:
  - el thread padre se ejecuta de forma concurrente con los *threads* hijos, por tanto puede terminar con la llamada a `exit()` antes de que terminen los hijos.

Terminación de *threads*

- Para acabar un *thread* sin terminar los demás *threads* en el proceso, se utiliza la función:

```
void pthread_exit(void *retval);
```

  Termina la ejecución de un thread, devolviendo el valor de ejecución `retval`.

- Un *thread* puede invocar la terminación de otro mediante el proceso de cancelación:

```
pthread_cancel(pthread_t thread);
```

  El *thread* puede terminar o no en función de su estado de cancelación. El *thread* que invoca esta llamada no se suspende.

- Función RTLinux no portable:

```
pthread_delete_np(pthread_t thread);
```


Control de ejecución

- Suspender o reanudar la ejecución de un *thread*:

```
pthread_suspend_np(pthread_t thread);
pthread_wakeup_np(pthread_t thread);
```

  Envía una señal `RTL_SIGNAL_SUSPEND/RTL_SIGNAL_WAKEUP` suspendiendo/reanudando la ejecución de un *thread*.

- Suspender la ejecución de un *thread* hasta que termine otra:

```
void pthread_join(pthread_t thread,
                  void **retval);
```

  Suspende la ejecución del *thread* que hace la llamada hasta que el *thread* indicado termina su ejecución.

Tareas (*threads*) periódicas

- Tareas RT en RTLinux se implementan a través de *threads*.
- POSIX no contempla la posibilidad de *threads* periódicas.
- Los mecanismos para implementar y gestionar *threads* periódicas en RTLinux no son portables.

```
pthread_make_periodic_np(pthread_t *thread,
                         hrtime_t start_time,
                         hrtime_t period);
```

Permite “despertar” un *thread* de forma periódica. El propio *thread* ha de suspenderse voluntariamente al final de cada activación y RTLinux lo reanuda en cada periodo. La unidad de tiempo es el nanosegundo.

```
pthread_wait_np(void);
```

Suspende la ejecución del *thread* que la invoca. El *thread* tiene que ser periódico.

Ejemplo tarea periódica

```
/* ejemplo.tarea.periodica.c */
#include <rtl.h>
#include <time.h>
#include <pthread.h>

pthread_t tareaRTperiodica;

/*### tarea RT periodica */
void * rutina(void *arg)
{ struct sched_param p;
  int nperiodos=0;

p.sched_priority=1;
pthread_setschedparam (pthread_self(),SCHED_FIFO, &p);

pthread_make_periodic_np (pthread_self(),
                gethrtime(), 500000000); /* periodo 0.5 seg */
while(1)
  { pthread_wait_np ();
    rtl_printf("Periodo %d\n",nperiodos);  nperiodos++;
  }
return 0;
}
```

Ejemplo tarea periódica

```
/*### incializacion del modulo */
int init_module(void)
{
rtl_printf("Cargando modulo ...\n");
return pthread_create (&tareaRTperiodica,
                       NULL, rutina, 0);
}



/*### descarga del modulo */
void cleanup_module(void)
{
pthread_delete_np(tareaRTperiodica);
/* pthread_cancel(tareaRTperiodica); */

rtl_printf("Modulo descargado\n");
}
```


Ejemplo tarea periódica

- Para ejecutar este ejemplo:
  - Cargar el módulo `rtl.o`
  - cargar módulo del planificador (`insmod rtl_sched.o`) para poder utilizar las funciones de creación y gestión de *threads* (tareas RT).
  - `gethrtime()` y las funciones de gestión del tiempo están en el módulo `rtl_time.o`.
  - Todos los módulos están en el directorio:

    /usr/src/rtlinux-3.1/modules
- Consideraciones:
  - Si la tarea `rutina()` tarda más de 0,5 s, LINUX "se colgará".
  - Comprobar como afectan las tareas RT a procesos Linux:
    - variar periodo o tiempo de computo y ejecutar algún proceso Linux al mismo tiempo.

Gestión del tiempo

- POSIX define operaciones para trabajar con relojes y temporizadores.
- RTLinux sólo implementa relojes POSIX.
- Se pueden implementar temporizadores, pero con funciones no estándar.
- Tipos de datos y constantes de tiempos (`time.h`):

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

typedef struct rtl_clock *clockid_t;

typedef long long hrtime_t;

typedef unsigned useconds_t;
```


Gestión del tiempo

- Funciones de consulta de reloj:

`hrtime_t gethrtime()`

Devuelve el tiempo actual en nanosegundos del reloj más eficiente.

`hrttime_t gethrtimeres()`

Devuelve la resolución del tiempo que devuelve `gethrtime()`.

`hrttime_t clock_gethrtime( clockid_t clock)`

Devuelve el tiempo actual del reloj indicado. El reloj puede ser alguno de los tres relojes lógicos o dos físicos disponibles en RTLinux:

```
CLOCK_REALTIME
CLOCK_MONOTONIC
CLOCK_RTL_SCHED
CLOCK_8254
CLOCK_APIC
```

Gestión del tiempo

- Temporización, esperas:

`rtl_delay(long duracion)`

Realiza un espera activa de duración en nanosegundos.

Se puede utilizar desde una tarea RT o un manejador de interrupción.

`usleep(unsigned duracion)`

Realiza un espera NO activa de duración en microsegundos.

La espera se realiza suspendiendo el thread, por lo que no se puede llamar desde un manejador de interrupción.

```
nanosleep(struct timespec *espera,
          struct timespec *restante);
```

Realiza un espera NO activa de duración en nanosegundos. En restante se devuelve el tiempo que queda de la espera en caso de que se abortará por recibir alguna señal.

Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

FIFOs

- Son mecanismos de comunicación basados en las “fifo” de UNIX.
- Se pueden utilizar para comunicar *threads* entre si o *threads* con procesos Linux.
- La comunicación es en forma de *stream* unidireccional:
  - se comporta como un buffer circular donde cada operación de lectura elimina del buffer los datos leídos.
- Son una función opcional de RTLinux y para su utilización hay que cargar el módulo `rtl_fifo.o`
- El módulo `rtl_nfifo.o` ofrece una nueva implementación donde además se implementan colas de mensajes con una estructura definida de información.

FIFOs

- Desde procesos Linux se programan como acceso a ficheros de dispositivo `/dev/rtfx`, donde x es el número de FIFO.
- Ejemplo:

```
#include <fcntl.h>

main()
{ int fd,count;
  int valor;

fd=open(“/dev/rtf0”,O_RDONLY);

while(1)
  { read(fd,&valor,sizeof(valor));
    printf(“valor: %8d\n”,valor);
  }
}
```

- Abrir/Leer/Escribir en las FIFOs a través de las funciones de ficheros `open/read/write`.

FIFOs

- API para el manejo de FIFOs <rtl_fifo.h>

```
int rtf_create(unsigned int fifo,int size)
```
creación de una RT-FIFO

```
int rtf_destroy(unsigned int fifo)
```
destrucción de una RT-FIFO.

```
int rtf_resize(unsigned int fifo,int size)
```
Redimensiona una RT-FIFO

```
int rtf_put(unsigned int fifo,char *buf,int count)
```
Escribir en una RT-FIFO

```
int rtf_get(unsigned int fifo,char *buf,int count)
```
Leer de una RT-FIFO

```
int rtf_flush(unsigned int fifo)
```
Vacia una RT-FIFO.

FIFOs

```
int rtf_create_handler(unsigned int fifo,int
                       (*handler)(unsigned int fifo))
```
Asocia un manejador a una RT-FIFO. Se llama el manejador cuando un proceso Linux lee o escribe en la FIFO asociada.

Para desinstalar el manejador, hay que instalar el manejador por defecto:

```
rtf_create_handler(n_fifo, default_handler);
```

```
int rtf_create_rt_handler(unsigned int fifo,
                      int (*handler)(unsigned int fifo))
```
Asocia un manejador a una RT-FIFO. Se llama el manejador cuando una tarea RT lee o escribe en la FIFO asociada.

La desinstalación se hace de forma similar a la anterior.

Ejemplo FIFO

- Ejemplo manejo de FIFOs:
  - Lee datos de una FIFO que se habrán escrito a través de un proceso LINUX
  - Emite sonido correspondiente en altavoz PC cada periodo de tiempo marcado para la tarea RT.
  - La tarea RT consistirá en leer de la FIFO y emitir sonido en a través de puerto del altavoz cada cierto periodo de tiempo (8192 Hz).
  - El proceso Linux consistirá en escribir sobre el fichero de dispositivo que soporta la FIFO un fichero de sonido Linux

```
cat linux.au > /dev/rtf0
```


Ejemplo FIFO

```
/* ejemplo.fifo.c */

#include <rtl.h>
#include <time.h>
#include <pthread.h>
#include <rtl_fifo.h>
#include <asm/io.h>

pthread_t task;

/*### Filtro de sonido */
static int filter(int x)
{ static int oldx;
  int ret;

if(x & 0x80)
  { x = 382 - x; }

ret = x > oldx;
oldx = x;
return ret;
}
```

Ejemplo FIFO

```
/*### tarea periodica sonido */
void * fun(void *arg)
{ char data;
  char temp;
  struct sched_param p;

p.sched_priority=1;
pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);

pthread_make_periodic_np (pthread_self(), gethrtime(),
   1000000000/8192LL); /* frecuencia 8192 Hz */

while(1)
  { pthread_wait_np ();
    if(rtf_get(0, &data, 1) > 0)
      { data = filter(data);
        temp = inb(0x61);
        temp &= 0xfd;
        temp |= (data & 1) << 1;
        outb(temp,0x61);
      }
  }
}
```


Ejemplo FIFO

```
int init_module(void)
{
rtl_printf("Cargando modulo\n");

rtf_create(0, 4000); /* crear fifo */

/* preparar puerto de sonido */
outb_p(inb_p(0x61)|3, 0x61);
outb_p(0xb0, 0x43);
outb_p(3, 0x42);
outb_p(00, 0x42);

return pthread_create(&task, NULL,fun, 0);
}


void cleanup_module(void)
{
pthread_delete_np(task);
rtf_destroy(0);

rtl_printf("Modulo descargado\n");
}
```

Ejemplo FIFO

- Para ejecutar ejemplo de la FIFO:
  - Compilar módulo con opciones definidas por /usr/src/rtlinux-3.1/rtl.mk
  - Cargar módulos:
    - bàsicos (rtl, rtl_time),
    - del planificador (rtl_sched) y
    - servicios FIFO (rtl_posixio, rtl_fifo).
  - Cargar módulo ejemplo.fifo.o
  - Copiar fichero audio en fichero dispositivo:

```
cat linux.au > /dev/rtf0
```


Ejemplo tarea NO RT

- Ejemplo proceso Linux:
  - Programa C bajo Linux.
  - Misma finalidad que ejemplo FIFO.
  - Diferencias con proceso RT.
  - Ejecutar proceso Linux.
  - Ejecutar proceso que cargue el procesador al mismo tiempo.
- Consideraciones:
  - No hay garantía de que se cumplan las condiciones temporales para un correcto resultado.
  - El proceso RT si que cumple los requisitos temporales exigidos.

Ejemplo tarea NO RT

```
/* ejemplo.linux.c */

#include <unistd.h>
#include <asm/io.h>
#include <time.h>

static int filter(int x)
{ static int oldx;
  int ret;

if(x & 0x80) x = 382 - x;
ret = x > oldx;
oldx = x;
return ret;
}

void espera(int x)
{ int v;

for (v=0; v<x; v++);
}
```


Ejemplo tarea NO RT

```
void fun()
{ char data;
  char temp;

while (1)
  { if (read(0, &data, 1) >  0)
      { data = filter(data);
        temp = inb(0x61);
        temp &= 0xfd;
        temp |= (data & 1) << 1;
        outb(temp,0x61);
      }
     espera(3000);
   }
}
```

```
int main(void)
{ unsigned char dummy,x;

ioperm(0x42, 0x3,1);
ioperm(0x61, 0x1,1);
dummy= inb(0x61);espera(10);
outb(dummy|3, 0x61);
outb(0xb0, 0x43);espera(10);
outb(3, 0x42);espera(10);
outb(00, 0x42);

fun();
}
```

Ejemplo tarea NO RT

- Particularidades:
  - El programa no puede acceder directamente a puertos E/S:
  - Petición permiso al núcleo (`ioperm()`).
  - Ajustar bucle de espera según frecuencia de reloj del procesador para obtener frecuencia de 100Mhz.
- Compilación ejemplo4

```
gcc -O2 ejemplo4.c -o ejemplo4
```

- Ejecutar con un fichero de sonido

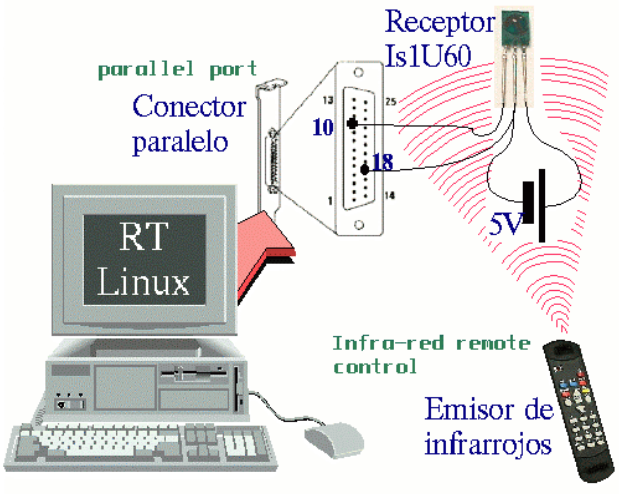
```
cat linux.au | ejemplo
```

- Si se ejecuta otro programa que cargue el procesador:
  - No hay garantías de cumplimiento de tiempos.
  - Sonidos entrecortados.

Ejecución de tarea RT

- En prácticas:

  manejador de receptor de infrarrojos, basado en artículo I.Ripoll y E. Acosta en LinuxFocus (http://www.linuxfocus.org):



Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Sincronización de *threads*

```
void print_message(void *ptr)
{ printf("%s ",(char*)ptr);
  pthread_exit(0);
}

main()
{ pthread_t thread1,thread2;
  char *message1="Hello",
  char *message2="World";

pthread_create(&thread1,
               pthread_attr_default,
               (void*)&print_message,
               (void*)message1);
usleep(2000000);

pthread_create(&thread2,
               pthread_attr_default,
               (void*)&print_message,
               (void*)message2);
usleep(2000000);
exit(0);
}
```

Sincronización de *threads*

- No se puede determinar a priori que *thread* terminará antes.
- La función `usleep()` introduce un retardo en el *thread* que lo llama, en este caso el *thread* padre.
- Si se hubiera utilizado `sleep()` en su lugar
  - hubieran parado todos las *threads* del proceso
  - `sleep()` es una función relativa al proceso.
- La utilización de retardos no es fiable:
  - hay que utilizar otros mecanismos de sincronización que ofrezcan garantías.
- POSIX ofrece dos mecanismos de sincronización:
  - *mutexes* y
  - variables condición.

Mutex

- *Mutex*
  - primitiva de bloqueo simple para controlar el acceso a recursos compartidos.
  - Funciones relacionadas:

    todas las funciones devuelven el valor cero si se ha realizado con éxito.

- Creación/inicialización de un *mutex*:

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

para atributos por defecto utilizar la constante `thread_mutexattr_default` que representa `NULL`

Mutex

- Bloqueo de un *mutex*:

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

  Bloquea el *mutex* siendo propietario del bloqueo el *thread* que realiza la llamada.

  Si el *mutex* ya estaba bloqueado por otro *thread*, el *thread* que intenta bloquearlo se queda suspendido hasta que el *mutex* quede desbloqueado.

- Desbloqueo de un *mutex*:

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

  desbloquea el *mutex*, si el *mutex* estaba bloqueado por el mismo thread.

- Destrucción de un *mutex*:

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

  destruye el *mutex* si está desbloqueado.

Ejemplo *mutex*

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

char buffer; /* buffer de 1 caracter */
int buffer_has_item=0; /*estado buffer*/
pthread_mutex_t mutex;
void writer(void);
void reader(void);

main()
{ pthread_t th_reader;

pthread_mutex_init(&mutex,
             thread_mutexattr_default);

pthread_create(&th_reader,
               pthread_attr_default,
               (void*)&reader,NULL);
writer();
}
```

Ejemplo *mutex*

```
void writer(void)
{ while(1)
    { pthread_mutex_lock(&mutex);
      if(!buffer_has_item)
        { buffer=make_new_item();
          buffer_has_item=1;
        }
      phthread_mutex_unlock(&mutex);
      usleep(2000000);
    }
}

void reader(void)
{ while(1)
    { pthread_mutex_lock(&mutex);
      if(buffer_has_item)
        { consume_item(buffer);
          buffer_has_item=0;
        }
      phthread_mutex_unlock(&mutex);
      usleep(2000000);
    }
}
```


Variables condición

- Variables condición
  - primitiva de bloqueo combinada con envío y recepción de señales.
  - Funciones relacionadas:
    todas las funciones devuelven el valor cero si se ha realizado con éxito.
- Creación/inicialización de una variable condición:

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *condattr);
```

  para atributos por defecto utilizar la constante `thread_condattr_default` que representa `NULL`

- Destrucción de una variable condición:

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

  destruye la variable condición si no hay ningún *thread* esperandola.

Variables condición

- Esperar señal sobre una variable condición:

```
int pthread_cond_wait(pthread_cond_t *cond);
```

  el *thread* que llama a la función suspende su ejecución (no consume CPU) hasta que otro *thread* envía una señal a la variable condición a la que está esperando, o en general, hasta que se cumple un predicado.

- Enviar señal a una variable condición:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

  Re-arranca uno de los *threads* que están esperando sobre la variable condición, sin especificar cual. Si no hay ningún *thread* esperando, no ocurre nada.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

  Re-arranca todos los *threads* que están esperando sobre la variable condición. Si no hay ningún *thread* esperando, no ocurre nada.

Ejemplo variable condición

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

char buffer; /* buffer de 1 caracter */
int buffer_has_item=0; /*estado buffer*/
pthread_cond_t cond;

void writer(void);
void reader(void);

main()
{ pthread_t th_reader;

pthread_cond_init(&cond,
             thread_condattr_default);
pthread_create(&th_reader,
               pthread_attr_default,
               (void*)&reader,NULL);
usleep(2000000);
writer();
}
```

Ejemplo variable condición

```
void writer(void)
{ while(1)
    { if(!buffer_has_item)
        { buffer=make_new_item();
          buffer_has_item=1;
          phthread_cond_signal(&cond);
        }

      usleep(2000000);
    }
}

void reader(void)
{ do
    { phthread_cond_wait(&cond);

        if(buffer_has_item)
        { consume_item(buffer);
          buffer_has_item=0;
        }
    }
  while(1);
}
```


Semáforos

- Semáforos
  - Variable sobre la que se pueden realizar operaciones de inicialización, incremento y reducción del valor.
  - Entraron a formar parte del estándar POSIX.1b, por lo que es posible que en algunos sistemas no estén disponibles.
  - Funciones relacionadas:
    - Fichero cabecera: `semaphore.h`
    - Nomenclatura: `sem_nombre()`
  - Se utilizan, en general, para sincronización de acceso a recursos compartidos en procesos (tareas RT) y también en hilos.

Semáforos

- Inicialización y destrucción de semáforos:

```
int sem_init(sem_t *sem,
             int pshared,
             unsigned int value);
```

Inicializa el semáforo con el valor `value` e indicando en `pshared` si el semáforo se puede compartir con otros procesos.

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

Destruye el contenido de un semáforo previamente inicializado con `sem_init()`;

Semáforos

- Operaciones sobre semáforos:

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

Espera (se bloquea) si el valor del semáforo es cero o menor que cero. Si es mayor que cero, se reduce el valor del semáforo y se desbloquea.

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

Misma que anterior pero no se bloquea si el valor del semáforo es cero o menor que cero. Si es mayor que cero, se reduce el valor del semáforo y se desbloquea.

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

incrementa el valor del semáforo.

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

obtiene el valor actual del semáforo.

Semáforos

- Semáforos para *threads*:
  - Se pueden implementar semáforos para *threads* a partir de *mutexes* y variables condición:
  - En el artículo de Wagner (1995) viene viene un ejemplo de código.
    - Funciones del ejemplo:

```
int semaphore_init(
   Semaphore *s);

int semaphore_destroy(
   Semaphore *s);

int semaphore_down(
   Semaphore *s);
   análoga a sem_wait().

int semaphore_up(
   Semaphore *s);
   análoga a sem_post().
```

```
int semaphore_decrement
   (Semaphore *s);
      decrementa el valor del
      semáforo sin bloqueo.

int semaphore_value
   (Semaphore *s);
    devuelve valor del semáforo.

typedef struct Semaphore
     { int v;
       phtread_mutex_t mutex;
       phthread_cond_t_ cond;
     }
```


Ejemplo semáforos

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

void writer(void);
void reader(void);

char buffer; /* buffer de 1 caracter */
sem_t writers_turn;
sem_t readers_turn;

main()
{ pthread_t th_reader;

sem_init(&readers_turn,0,1);
sem_init(&writers_turn,0,1);

sem_wait(&readers_turn); /* el writer debe ir primero */

pthread_create(&th_reader,pthread_attr_default,
               (void*)&reader,NULL);
writer();
}
```

Ejemplo semáforos

```
void writer(void)
{ while(1)
    { sem_wait(&writers_turn);
      buffer=make_new_item();
      sem_post(&readers_turn);
    }
}


void reader(void)
{ do
    { sem_wait(&readers_turn);
      consume_item(buffer);
      sem_post(&writers_turn);
    }
  while(1);
}
```


Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Señales en *threads*

- En UNIX se utilizan como mecanismo asíncrono de comunicación:
  - Se puede entender como el papel que hacen las interrupciones del procesador.
- En RTLinux se utilizan para:
  - Comunicar tareas RT entre si.
  - Modificar estado de ejecución de tareas RT.
  - Atender interrupciones hardware.
  - Disparar eventos en el núcleo de Linux.
- Actualmente la gestión de interrupciones es no estándar:
  - Está previsto gestionarlas a través de señales.

Señales en *threads*

- Un proceso puede establecer la forma de atender señales, dependiendo del estado de la señal en el proceso:
  - *Ignorada:* el proceso no recibe estas señales.
  - *Capturada:* se ejecuta una función manejadora
  - *Opción por defecto:* depende del tipo de señal.
  - *Bloqueada:* no se entregan hasta que se desbloquean.
- Las señales no llegan al proceso destino de forma inmediata.
- No reanudan procesos.
  - Esperan hasta que se reanude el proceso destino.
- Las señales físicas, interrupciones, si que son atendidas inmediatamente.
- En RTLinux, el número de señales es de 1024, no como en SO clásicos (16 a 64).

Señales en *threads*

- Hasta el momento, está implementado como realizar el envío de una señal a un *thread*:

```
pthread_kill(pthread_t thread, int signal);
```

  Envía la señal `signal` a `thread`. La señal no llegará hasta que el `thread` esté en ejecución. La función retorna inmediatamente.

  Las posibles señales son:

  - `RTL_SIGNAL_NULL` no causa ninguna acción. Se utiliza para comprobar que el *thread* destino está en ejecución.
  - `RTL_SIGNAL_SUSPEND` suspende la ejecución del *thread*.
  - `RTL_SIGNAL_WAKEUP` despierta el *thread*.
  - `RTL_SIGNAL_CANCEL` cancela la ejecución del *thread*.
  - `RTL_LINUX_MIN_SIGNAL` equivale a enviar una interrupción software al núcleo de Linux. Representa la interrupción 0. El *thread* destinatario es el núcleo de Linux `rtl_get_linux_thread(rtl_getcpuid())`
  - `RTL_SIGNAL_KILL` termina la ejecución del *thread*.

Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Paralelismo y concurrencia

- RTLinux implementa tareas RT a través de *threads*.
- Se ejecutan de forma concurrente, normalmente en un mismo procesador.
- RTLinux ofrece funciones no POSIX para poder asignar *threads* a procesadores en sistemas SMP con varios procesadores:
  - Se pueden implementar procesos (tareas RT) que se ejecuten en paralelo.
- API:

```
int pthread_attr_getcpu_np(phtread_attr_t *attr,
                           int *cpu);

int pthread_attr_setcpu_np(phtread_attr_t *attr,
                           int cpu);
```

Si no se especifica en los atributos del *thread*, se asigna por defecto el procesador actual.

Ejemplo *threads* en SMP

```
/* Ejemplo: Asignacion de hilos a CPUs */

#include <rtl.h>
#include <time.h>
#include <pthread.h>
#include <rtl_core.h>
#include <rtl_sched.h>
#include <errno.h>

pthread_t thread1,thread2;

void * start_routine(void *arg)
{ int cpu;

cpu=rtl_getcpuid();
rtl_printf("Thread \"%s\" en la CPU %d\n",(char*)arg,cpu);
return NULL;
}

void cleanup_module(void)
{
pthread_delete_np (thread1); pthread_delete_np (thread2);
}
```

Ejemplo *threads* en SMP

```
int init_module(void)
{ pthread_attr_t attr;
  struct sched_param sched_param;
  int thread_status;

rtl_printf("El modulo de arranque esta situado en la CPU %d\n",
                                rtl_getcpuid());
pthread_attr_init(&attr);
pthread_attr_setcpu_np(&attr, 0);
sched_param.sched_priority = 1;

pthread_attr_setschedparam(&attr,&sched_param);
if(pthread_create(&thread1,&attr,start_routine,"Hello "))
  { rtl_printf("Hilo NO creado.\n");
    return -1;
  }

pthread_attr_setcpu_np(&attr, 1);
if(pthread_create(&thread2,&attr,start_routine,"World!"))
  { rtl_printf("Hilo NO creado.\n");
    return -1;
  }
return 0;
}
```


Paralelismo y concurrencia

- Ejercicio:
  - a) Crear un módulo RT con 4 *threads*, 3 más el *thread* padre, que realice la siguiente operación:

    `op1+op2+op3 = resultado`

    - El *thread* 1 realizará la `op1=x*y`
    - El *thread* 2 realizará la $op3=x^3*y$
    - El *thread* 3 realizará la $op1=x^2*y^3$
    - El *thread* padre calculará el resultado final a partir del los cálculos realizados por los demás *threads*.
    - Los valores de x e y deben poder asignarse en el momento de carga del módulo.
  - b) Aplicar paralelismo asignando cada *thread* a un procesador en un sistema tetraprocesador.

Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Gestión de interrupciones

- El núcleo de RTLinux captura todas las interrupciones y se las reenvía al Linux.
- Está proyectado que las señales sean el interfaz para manejo y captura de interrupciones en RTLinux.
  - Ejemplo: el rango de señales de `RTL_SIGRQMIN` a `RT_SIGIRQMIN+NR_IRQS` son señales de interrupciones hardware.
- El núcleo del RTLinux ofrece servicios no estándar para instalar y desinstalar interrupciones.
- Includes:

```
#include <rtl_core.h>
#include <asm/rt_irq.h>
```

Gestión de interrupciones

- Habilitar/deshabilitar interrupciones:

  `rtl_no_interrupts(rtl_irqstate_t estado);`
  macro que salva los *flags* del procesador en `estado` e inhabilita las interrupciones (`cli`).

  `rtl_restore_interrupts(rtl_irqstate_t estado);`
  carga los *flags* del procesador con el valor `estado`.

  `rtl_stop_interrupts();`
  macro que deshabilita las interrupciones (`cli`).

  `rtl_allow_interrupts();`
  habilita las interrupciones (`sti`).

  Todas estas funciones únicamente tienen efecto sobre el procesador que se ejecutan.

Gestión de interrupciones

- Enmascarar interrupciones:
  - Bloquear temporalmente de forma individualizada la recepción de una interrupción por el procesador.

  `int rtl_hard_disable_irq(unsigned int irq);`
  enmascara la interrupción `irq`.

  `int rtl_hard_enable_irq(unsigned int irq);`
  desenmascara la interrupción `irq`.

- Instalar interrupción:

```
int rtl_request_irq(unsigned int irq,
      unsigned int (*handler)(unsigned int irq,
                              struct pt_regs *reg));
```

  Devuelve 0 si se ha instalado correctamente, sino, un valor negativo.
  El valor devuelto por el manejador no se utiliza.
  Cuando se instala la interrupción, ésta queda enmascarada hasta que se llame a `rtl_hard_enable_irq()`.

Gestión de interrupciones

- Desinstalar interrupción:

```
int rtl_free_irq(unsigned int irq);
```

  Libera el manejador asociado a la interrupción.

- Interrupciones software:
  - Pasar una interrupción al núcleo de Linux.
  - Ejemplo: `rtl_printf()` guarda en un buffer la cadena de caracertes a imprimir y luego llama a la interrupción de Linux encargada de llamar a `printk()`.
  - Permite definir interrupciones para la máquina virtual Linux.
  - Linux creerá que proceden de algún periférico.

Gestión de interrupciones

- Interrupciones software:

```
int rtl_get_soft_irq(handler,char *devname);
```

  Instala un manejador de interrupciones de Linux. Se atienden en el espacio de ejecución de Linux.
  `devname` identifica la interrupción. Aparece en el listado de interrupciones del sistema `/proc/interrupts`.
  El valor que devuelve es el número de interrupción libre que se ha asignado.

```
int rtl_free_soft_irq(int irq);
```

  Desinstala el manejador de interrupción. Se devuelve el valor de `irq` que asigno la función `rtl_get_soft_irq()`.

```
int rtl_global_pend_irq(int irq);
```

  Genera una interrupción software que se pasará a Linux cuando éste se ponga en ejecución.

Ejemplo interrupciones

```
/* instalar.interrupcion.c */
#include <rtl_core.h>
#include <asm/rt_irq.h>

rtl_irqstate_t f; /* registro de flag del microprocesador */

/*### funcion de tratamiento de la interrupcion */
unsigned manejador_interrupcion(unsigned int irq,
                                struct pt_regs *regs);

/*### Instalacion de la rutina de interrupcion */
int instalar_interrupcion(unsigned int irq)
{ int res;

  rtl_no_interrupts(f); /* salvar flags del microprocesador */
  res=rtl_request_irq(irq,manejador_interrupcion);

  if(!res) /* res=0, interrupcion instalada correctamente */
    { rtl_printf("Rutina de interrupcion instalada\n");
      rtl_hard_enable_irq(irq); /* desenmascararla */
    }
return(!res); /* res==0, no se pudo instalar */
}
```


Ejemplo interrupciones

```
/*### Desinstala la rutina de interrupcion de irq  */
void desinstalar_interrupcion(unsigned int irq)
{
rtl_free_irq(irq); /* liberar rutina de interrupcion */
rtl_restore_interrupts(f); /* restaurar flags del micro */
}

/*### Incializacion del modulo */
int init_module(void)
{
rtl_printf("Instalacion de la rutina de interrupcion ...\n");
if(!instalar_interrupcion(IRQ_NUMBER))
  {rtl_printf("Rutina de interrupcion no instalada\n");
  }
return(0);
}

/*### Descarga del modulo */
void cleanup_module(void)
{
Desinstalar_interrupcion(IRQ_NUMBER);
rtl_printf("Rutina de interrupcion desinstalada.\n");
}
```

Ejemplo 2 interrupciones

```
/* interrupcion.linux.c
   capturar la interrupcion de teclado y reenviarla a Linux */

#include <rtl_core.h>
#include <asm/rt_irq.h>

#define KEYBOARD_INTERRUPT  1

/*### funcion de tratamiento de la interrupcion */
unsigned my_keyboard_interrupt_handler(unsigned int irq,
                                       struct pt_regs *regs)
{
/* imprimir algo para decir que hemos pasado por aqui */
rtl_printf("+"); /* aquí no funciona printk() */

/* enviar la misma interrupcion al nucleo de Linux, para que
   pueda recoger la tecla apretada.
   Si no se reenvia, Linux no recibira esta interrupcion. */
rtl_global_pend_irq(KEYBOARD_INTERRUPT);

return 0;
}
```


Ejemplo 2 interrupciones

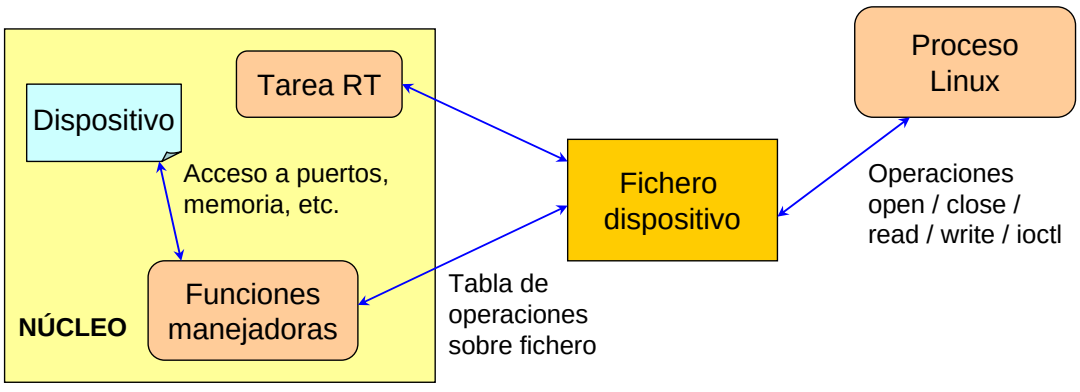
```
/*### funcion de inicializacion del modulo */
int init_module(void)
{
printk("Cargando modulo %s\n", __FILE__);
return  rtl_request_irq( KEYBOARD_INTERRUPT,
                         my_keyboard_interrupt_handler);
}


/*### funcion de descarga del modulo */
void cleanup_module(void)
{
rtl_free_irq(KEYBOARD_INTERRUPT);
printk("Modulo %s descargado.\n", __FILE__);
}
```

Contenido

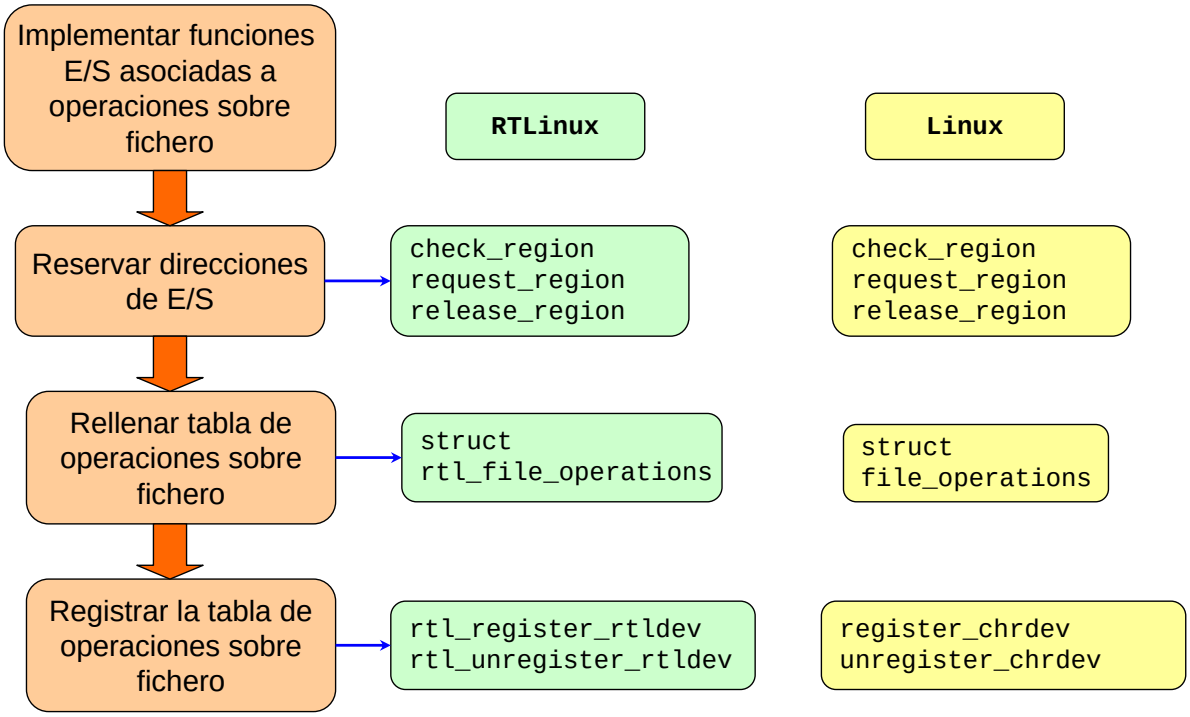
- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Gestión E/S

- Dispositivos POSIX y E/S:
  - Sistema de E/S en UNIX se basa en el concepto de fichero.
  - POSIX obliga que los ficheros especiales estén en `/dev`.
  - RTLinux ofrece servicios para implementar dispositivos POSIX.



Drivers POSIX



Drivers POSIX

- Verificación, reserva y liberación de dirección E/S:

```
int check_region(unsigned int port,
                 unsigned int range);
```

verifica si el rango de puertos indicados está (devuelve valor menor que cero) o no en uso.

```
void request_region(unsigned int port,
                    unsigned int range, const char *name);
```

reserva el rango de puertos indicado, asignándoles el identificador `name`.

```
void release_region(unsigned int port,
                    unsigned int range);
```

libera el rango de puertos indicado.

Drivers POSIX

- Registro y baja de dispositivos y sus tablas de operaciones:

```
int rtl_register_rtldev(unsigned int major,
                        const char *name,
                        struct file_operations *fops);
```

si se registra correctamente, devuelve 0 o valor positivo.
`file_operations` es una estructura donde se asignan las funciones manejadoras de cada acción sobre el fichero:

```
lseek, read, write, readdir, select,
ioctl, mmap, open, close, fsync, fasync
```

```
int rtl_unregister_rtldev(unsigned int major,
                        const char *name);
```

siendo name el nombre registrado y su correspondiente `major`.

- Para crear un fichero especial de dispositivo, hay que utilizar el comando `mknod`.

Entrada/Salida

- Funciones de acceso a puertos de E/S:

```
#include <asm/io.h>
```

- El acceso a puertos se puede realizar también desde:
  - tareas RT (*threads*).
  - manejadores de interrupción.
- Al compilar con el fichero de cabecera, hay que utilizar siempre la opción de optimización `-O2` o superior.
- Funciones E/S rápidas:

```
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);
```

Acceso a memoria

- Funciones E/S con espera posterior (hardware lento):

```
outb_p();  outw_p()  outl_p();
inb_p();   inw_p();  inl_p();
```

- Acceso a la memoria física:
  - Memoria física mapeada en el núcleo: acceso directo.
  - También se pueden utilizar las macros:

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


Acceso a memoria

- Funciones de acceso a bloques de memoria:

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

- Programación E/S desde procesos Linux:
  - Procesos normales no tienen acceso a puertos (nivel de privilegio 3).
  - Se ha de mapear el acceso solicitando al sistema a través de la función:

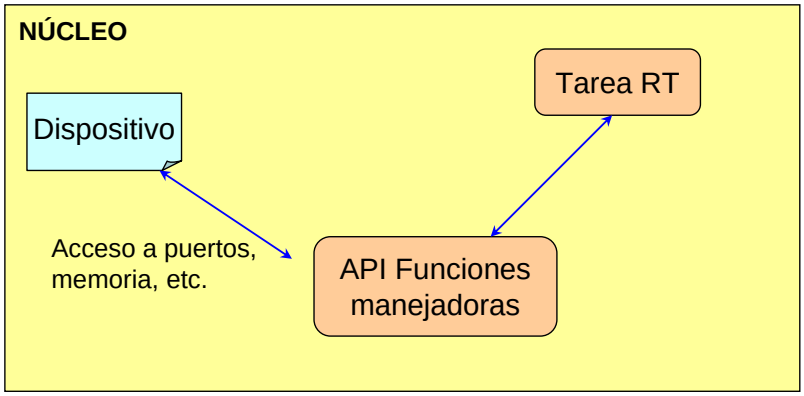
```
ioperm(u_long from, u_long num, int turn_on);
```

    - Solicita `turn_on` el rango `[from,form+num-1]`.
    - Solo puede utilizarla los procesos de *root*.
  - Un proceso puede solicitar al sistema poder ejecutarse en otro nivel (nivel 0 es el mayor privilegio):

```
iopl(int level);
```

Drivers RT

- *Drivers*: en RTLinux, acceso directo a recursos.
- Módulos con un API de funciones de interfaz.



Drivers RT

- Ejemplo `rt_com`: *driver* para el puerto serie.
  - Include: `rt_com.h`
  - API:
    - Configurar puerto serie

```
void rt_com_setup(
    unsigned int com,
    unsigned baud,
    unsigned parity,
    unsigned stopbits,
    unsigned wordlength);

parity:   RT_COM_PARITY_EVEN
          RT_COM_PARITY_ODD
          RT_COM_PARITY_NONE
```

Drivers: rt_com

- Escribir y leer en puerto serie

```
void rt_com_write(unsigned int com,
                  char *ptr,int cnt);

void rt_com_read(unsigned int com,
                 char *ptr,int cnt);
```

  En modo POSIX, estas funciones se re-emplazan por escribir y leer en fichero dispositivo.

- Información sobre especificaciones del puerto serie

```
http://www3.uji.es/~vmarti/info/serial.pdf
```


Driver CAN. Ejemplo código

```
/* Mensajes de los buffers de emision y recepcion */
typedef struct dat_buffer_CAN{
   dword id;       // campo de control
   byte datos[8];  // datos
   byte num_bytes; // numero de bytes de datos en el mensaje
   byte tipo;      // tipo de mensaje CAN
} MsgCAN;

/*------- funciones de la API del driver -------*/

/*### Inicializar controlador CAN. Devuelve el numero de
   identificador CAN del controlador */
int InicializarCAN(void);

/*### Funcion para leer mensajes recibidos. Devuelve 0 si no
   hay mensajes en la cola de recepcion */
int LeerMensajeCAN(MsgCAN *msg);

/*### Enviar un mensaje a la red CAN */
int EnviarMensajeCAN(MsgCAN *msg);
```

Driver CAN. Ejemplo código

```
/*### Incializacion del modulo */
int init_module(void)
{ int card_id;

/* comprobar y reservar rango de direciones io */
if(-EBUSY == check_region(ioCANbase,33))
  { rtl_printf("ERROR! Direcciones de E/S ocupadas\n");
    error_instalacion=1;
 }
request_region(ioCANbase,33,"rt_pccan3");

/* incializacion del controlador */
card_id=InicializarCAN();

rtl_printf("Instalacion de la rutina de interrupcion ...\n");
if(!InstalaInterrupcion())
  { error_instalacion=2;
    rtl_printf("ERROR! al instalar la rutina de interrupcion;
  }

rtl_printf("ID tarjeta CAN: %d\n",(int)card_id);
return(0);
}
```


Driver CAN. Ejemplo código

```
/*### Descarga del modulo */
void cleanup_module(void)
{
/* liberar direcciones io */
if(error_instalacion!=1)
  release_region(ioCANbase,33);

/* desinstalar interrupcion */
if(error_instalacion!=2)
  DesinstalarInterrupcion();

rtl_printf("Modulo driver CAN descargado\n");
}
```

Driver CAN. Ejemplo código

```
/*### Instalacion de la rutina de interrupcion */
int InstalaInterrupcion(void)
{ int res;

  rtl_no_interrupts(f);
  res=rtl_request_irq(irqCAN,can_interrupt);

  /* res=0, interrupcion instalada correctamente */
  if(!res)
    { rtl_printf("Rutina correctamente instalada\n");
      rtl_hard_enable_irq(irqCAN);
    }

  return(!res);
}

/*### Desinstala la rutina de interrupcion CAN */
void DesinstalarInterrupcion(void)
{
  rtl_free_irq(irqCAN);
  rtl_restore_interrupts(f);
}
```


Driver CAN. Ejemplo código

```
/*### Manejador de interrupciones CAN */
static unsigned int can_interrupt(unsigned int num,struct
   pt_regs *reg)
{
// ...
// Acceder a direcciones E/S del controlador CAN
// Recoger dato que ha generado la interrupcion
// Poner dato en la cola local de recepcion
// Enviar posibles datos que queden en la cola de transmision
}

/*### Enviar un mensaje a la red CAN */
int EnviarMensajeCAN(MsgCAN *msg)
{
rtl_hard_disable_irq(irqCAN);
// ...
// ver si controlador esta libre para emitir
// si no esta libre, poner mensaje en la cola
rtl_hard_enable_irq(irqCAN);
}
```

Driver CAN. Ejemplo código

```
/*### Funcion para leer mensajes recibidos */
int LeerMensajeCAN(MsgCAN *msg)
{
/* Extrae mensaje de la cola de recepcion */
rtl_hard_disable_irq(irqCAN);
// ...
// sacar mensaje de la cola de recepcion
// devolver 0 si la cola esta vacia
rtl_hard_enable_irq(irqCAN);
}

/* registro de flag del microprocesador */
rtl_irqstate_t f;

/*### Inicializar controlador CAN e interrupciones */
int InicializarCAN(void)
{
// acceder a registros E/S del controlador CAN y
// configurar controlador.
}
```


Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Gestión de memoria compartida

FIFOS vs Memoria Compartida

- FIFOS:
  - Cola de datos. No se necesita un protocolo adicional para prevenir sobre-escritura de los datos.
  - Limites de la cola no están mantenidos internamente, lo tiene que hacer la aplicación.
  - Proporcionan bloqueo para sincronización. Los procesos no necesitan consultar si ha llegado un dato.
  - Es un canal de comunicación punto a punto de tipo serie, análogo a dispositivos UNIX
  - No soportan un proceso que escribe y varios lectores a la vez.

Gestión de memoria compartida

- Memoria compartida:
  - Los clientes necesitan definir algún tipo de protocolo para prevenir sobre-escritura.
  - Las datos pueden ser actualizados rápidamente, incluso de estructuras de gran tamaño.
  - Pueden ser escritas o leídas por varios procesos a la vez.
  - El número de canales independientes de comunicación dependen del tamaño de la memoria disponible.
  - Bloqueo para sincronización no esta directamente soportado. Se tiene que gestionar de forma adicional.
  - Exclusión mutua entre procesos no esta garantizada. Se pueden detectar lecturas y escrituras interrumpidas.

Memoria compartida

- Las tareas RT se ejecutan en el mismo espacio de memoria del núcleo.

- Existen funciones para reservar memoria y poder compartirla con procesos Linux (`mbuff`):
  - No son estándar. Hay funciones POSIX similares.
  - Siempre estará en memoria principal.
  - No hay garantía de que sea físicamente contigua (utiliza `vmalloc`).
  - No se puede llamar desde tareas RT o rutinas de interrupción.

Memoria compartida

`void *mbuff_alloc(name,size);`
si no existe ninguna zona con el nombre `name`, la crea de tamaño `size` y devuelve su dirección del inicio del bloque. Si ya existe, `size` tiene que ser menor o igual que el tamaño existente.

`void *mbuff_free(name,*buff);`
libera la zona de memoria indicada cuando ya no la utilice ningún otro proceso o tarea RT.

- Para su utilización:
  - Fichero cabecera: `mbuff.h`
  - Para procesos Linux, se realiza a través del *driver* `/dev/mbuff`
  - Para RTLinux tiene que cargarse el módulo `mbuff.o`
  - Para RTLinux y procesos Linux son funciones diferentes.

Memoria compartida

- Memoria física contigua:
  - `kmalloc()`, pero está limitada a 128Kb.
  - Forzar en el arranque a utilizar menos memoria RAM de la existente. La parte que no se utilizará siempre es de la parte alta.
  - Utilizar el parche `bigphysarea`, para reservar en el arranque la memoria que se precise y luego gestionarla al margen de Linux.

Bigphysarea

- `bigphysarea` es un *driver* que que reserva una cantidad de memoria en el arranque. Se instala como un parche de Linux.
- Son bloques de memoria físicamente contigua.
- Funciones para obtener y liberar memoria:

  `caddr_t bigphysarea_alloc_pages(count,align,prio);`
  devuelve la dirección de memoria compuesta por `count` páginas (4Kb) alineadas a un múltiplo de `align` páginas. El parámetro `prio` (usar el valor `GFP_KERNEL`) se utiliza sólo la primera vez que se llama a esta función.

  `bigphysarea_free_pages(base);`
  libera memoria reservada con `bigphysarea_alloc_pages()`.

  - Sólo se pueden utilizar desde dentro del núcleo.
- Ventaja frente a “memoria alta no usada”:
  - Se podrán realizar sobre ella operaciones de DMA.

Memoria alta

- Se configura como un bloque de memoria física aparte en el momento del arranque del sistema.
- Ejemplo:
  - Sistema con 32 MB.
  - Se quiere utilizar 1 MB de memoria compartida.
  - La dirección base de la memoria compartida será la del último MB.
  - Configuración del arranque (`/etc/lilo.conf`):

```
image=/boot/zImage
label=rtlinux
root=/dev/hda2
read-only
append="mem=31m"
```


Memoria alta

- Ejemplo 2
  - Sistema con 16MB
  - Se quiere utilizar 512KB de memoria compartida.
  - La dirección base de la memoria compartida será

    16384KB - 512KB = 15872KB
  - "`/etc/lilo.conf`":

```
...
append="mem=15872k"
```

- El tamaño del área de memoria compartida debe de ser menor que el tamaño de página declarado en "`/usr/include/asm/param.h`" que normalmente es 4MB.

Acceso a la memoria alta

- Cálculo de la dirección base:

```
#define BASE_ADDRESS (32 * 0x100000)
#define BASE_ADDRESS (15872 * 0x400)
```

- Desde un proceso Linux:
  - La memoria física del ordenador está mapeada en el dispositivo `/dev/mem`.
  - Un proceso Linux tiene que acceder a ella a través de este dispositivo, mapeando la memoria de este fichero de dispositivo en el proceso Linux.
  - El fichero de dispositivo solo tiene permisos de acceso para el usuario *root*. Para que cualquier proceso de usuario pueda acceder a este dispositivo se deberían cambiar los permisos.

Ejemplo acceso a memoria en *Linux*

```
#include <stdlib.h>   /* sizeof() */
#include <unistd.h>   /* open() */
#include <fcntl.h>    /* O_RDWR */
#include <sys/mman.h> /* mmap(), PROT_READ, MAP_FILE */

#define MAP_FAILED ((void *) -1)


...

int fd;
MY_STRUCT *ptr; /* estructura de datos propia */

if((fd = open("/dev/mem", O_RDWR)) < 0)
  { printf("No se pudo abrir el dispositivo
            mem\n");
    return(MAP_FAILED);
  }
```

Ejemplo acceso a memoria en *Linux*

```
/* mapear memoria fisica en memoria del proceso */
ptr = (MY_STRUCT *) mmap(0, sizeof(MY_STRUCT),
             PROT_READ | PROT_WRITE,
             MAP_FILE | MAP_SHARED,
             fd, BASE_ADDRESS);

if (MAP_FAILED == ptr)
  { printf("No se pudo realizar el mapeado de
           memoria\n");
    return(MAP_FAILED);
  }

close(fd);   /* fd ya no se necesita*/

...
```


Acceso a memoria alta

- `mmap()` devuelve la dirección de la memoria mapeada en el proceso Linux:

```
void *mmap(void *start,length,prot,flags,fd,offset);
```

- Para desmapear la memoria cuando ya no se necesite:

```
munmap(ptr,sizeof(MY_STRUCT));
```

- Desde una tarea RT-Linux
  - se puede direccionar directamente:

```
MY_STRUCT *ptr;
ptr=(MY_STRUCT*) BASE_ADDRESS;
```

  - Para versiones 2.1.XX, se tiene que mapear mediante la llamada a la macro `__va()`, definida en `/usr/include/asm/page.h`

```
ptr=(MY_STRUCT*) __va(BASE_ADDRESS);
```

Exclusión mutua

- Memoria compartida entre proceso Linux y RTLinux:
  - El proceso RT puede interrumpir una lectura/escritura del proceso Lunix, pero no al contrario.
  - Se puede utilizar la técnica del “flag” (bandera) para que el proceso Linux informe al RT del estado de uso de la memoria:
    - Se puede definir una variable *flag* al principio de la estructura de memoria compartida.

```
typedef struct {
        unsigned char en_uso;
        ... } MY_STRUCT;
```

    - El proceso Linux puede levantar el *flag* cuando quiera escribir o leer y bajarla cuando termine.

Exclusión mutua

    - La tarea RTLinux comprueba el estado de acceso del proceso Linux y si la bandera esta levantada, la tarea RT desiste en el acceso a la memoria hasta que el proceso Linux termine.

- Cuando proceso Linux escribe en memoria compartida:

```
MY_STRUCT *ptr, mi_estructura;
/* se asume que el puntero ptr ya apunta al inicio de la
   memoria compartida */

ptr->en_uso=1; /* levantar la bandera */

/* copiar información local en memoria compartida */
memcpy(ptr,&mi_estructura,sizeof(MY_STRUCT));

ptr->en_uso=0; /* bajar la bandera */
```

Contenido

- Introducción
- Características básicas
- Instalación RTLinux
- Módulos
- Creación de tareas RT.
- *Threads* en POSIX
- Gestión de *threads*.
- FIFOs
- Sincronización de *threads*
- Señales en *threads*.
- Paralelismo y concurrencia.
- Gestión de interrupciones.
- Gestión de E/S.
- *Drivers* en RTLinux.
- Gestión de memoria compartida
- RTLinux como sistema empotrado
- Bibliografía

Exclusión mutua

- Cuando proceso Linux lee en memoria compartida:

```
ptr->en_uso=1; /* levantar la bandera */

/* copiar información de memoria compartida en memoria local */
memcpy(&mi_estructura,ptr,sizeof(MY_STRUCT));

ptr->en_uso=0; /* bajar la bandera */
```

- Cuando la tarea en tiempo real quiere acceder a la memoria compartida:

```
if(ptr->en_uso)
  { /* el proceso linux esta accediendo a ella */
    /* intentarlo un poco mas tarde */
  }
else
  { /* acceso permitido a la memoria */
  }
```

Sistemas empotrados

- Sistemas Empotrados:
  - Aplicación de sistemas basados en tiempo real.
  - Sistemas informáticos en tiempo real integrados en otros sistemas y que están dedicados exclusivamente a una tarea de control o gestión del sistema donde están integrados.
  - Pueden llevar un entorno de usuario (dispositivos E/S para gestión del usuario), incluso gráfico.
  - Usualmente encapsulados en algún tipo de hardware y de tamaño reducido.
- Ejemplos:
  - Aplicaciones bio-medicas.
  - Sistemas de control industrial.
  - Electrodomésticos.

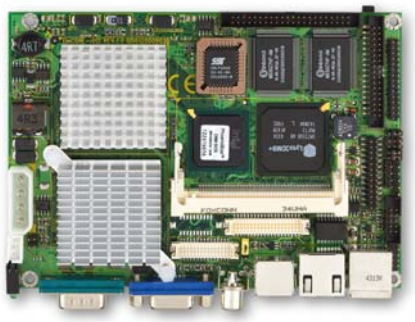
Sistemas empotrados

- RTLinux se puede utilizar para sistemas empotrados:
  - posee las características para implementar un sistema multitarea en tiempo real (periódicas y no periódicas).
  - posee toda la funcionalidad del Linux para la parte del proceso que no precise tiempo real
    - acceso a red,
    - entorno gráfico (*XWindow*)
    - dispositivos de entrada/salida estándar (teclado, ratón, *joystick*, sonido, dispositivos gráficos, etc.).
- No siempre es la mejor opción para un sistema empotrado basado en PC:
  - Gran tamaño del software del sistema.
  - Tamaño del hardware estándar puede resultar demasiado voluminoso.

Sistemas empotrados

- Posibles soluciones:
  - Arrancar RTLinux a través de la red.
  - Arrancar desde una EPROM [Dave, 1997].
  - Existen placas base de PC con microprocesador incluido en tamaños reducidos, entorno a 10x10 cm.
  - miniRTL (www.rtlinux.org/minirtl):
    - Implementación reducida de RTLinux.
    - Cabe en un disco de 1,44 MB.

145x102 mm
Celeron 400MHz



Bibliografía

- "A Linux-based Real-Time Operating System", Michael Barabanov. Master of Science, New Mexico Institute of Mining and Technology.
- "Linux as an Embedded Operating System",Jerry Epplin, http://www.espmag.com/97/fe39710.htm
- "Implementing Loadable Kernel Modules for Linux", Matt Welsh http://www.ddj.com/ddj/1995/1995.05/welsh.html
- "Linux Kernel Internals", M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner. Ed: Addison-Wesley.
- "Linux Kernel Module Programming Guide", Ori Pomerantz http://www.linuxdoc.org/LDP/lkmpg/mpg.html
- Linux Kernel Programming Guide: http://www.linuxhq.com/guides/LKMPG
- "Using Shared Memory in Real-Time Linux", F. M. Proctor, http://www.isd.cme.nist.gov/projects/emc/shmem.html.
- "Booting Linux from EPROM", B. Dave, Linux Journal (http://www.linuxjournal.com), January 1997.
- Embedded Systems Programming in C and Assembly", J. R. Brown, 1994.

Bibliografía

- "Programación Linux 2.0, API de sistema y funcionamiento del núcleo", Rémy Card, Eric Dumas y Franck Mével. Ed: Eyrolles, Ediciones Gestión 2000.
- "RT-Linux Manual Project", RT-Linux Documentation Group, http:///www.rtlinux.org
- "Real-Time Linux (RT-Linux)" I. Ripoll, http://www.linuxfocus.org, mayo 98.
- "Real-Time Linux II" I. Ripoll, http://www.linuxfocus.org, Julio 98.
- "Receptor de Mando a Distancia con RTLinux" I. Ripoll, E. Acosta http://www.linuxfocus.org, Marzo 2000.
- Pthreads Information. http://www.cs.ucr.edu/~sshah/pthreads/
- "Getting Started with POSIX Threads", T. Wagner and D. Towsley, 1995, http://centaurus.cs.umass.edu/~wagner/threads_html
- "UNIX, Programación Práctica. Guía para la Concurrencia, la Comunicación y las Multihilos", Robbins, K.A. and Robins, S.; Prentice Hall, 1997.