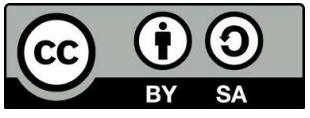
Interrupciones y trabajos diferidos

LIN - Curso 2014-2015



J.C. Sáez
High Performance Computing Division ArTeCs Group
Universidad Complutense de Madrid

Contenido

Contenido

Introducción

- **Interrupción**: señal que un dispositivo HW envía al procesador cuando requiere su atención
  - Motivación interrupciones:
    - Diferencia de velocidad entre CPU y dispositivos de E/S
    - Evitar *polling* (E/S programada)

Procesamiento de interrupciones en computador con SO

- Al producirse una interrupción, el kernel ejecuta una rutina de tratamiento (RTI)
- Procesamiento típico de RTI:
  - Transferir datos entre dispositivo y memoria principal
  - Reseteo de HW para siguiente interrupción
  - ...
- Los *drivers* de dispositivo (módulos del kernel) son los encargados de realizar procesamiento ligado a una interrupción

Gestión RTIs

RTI teclado matricial placa ARM (EC)

```
void KeyboardInt(void) __attribute__ ((interrupt ("IRQ")));

void KeyboardInt(void) {
    int value;
    ...
    /* Identificar la tecla */
    value = key_read();

    /* Si la tecla se ha identificado, visualizarla */
    if(value > -1)
           D8Led_symbol(value);

    /* Esperar a se libere la tecla */
    while ((rPDATG & 0x2)==0);

   /* Esperar trd mediante la funcion Delay() */
    Delay(100);

   /* Borrar interrupciones (solo eint1) */
   rI_ISPC= BIT_EINT1;
}
```

- Formato de la RTI es específico de arquitectura
- Código interactúa con controlador de interrupciones específico
- No es posible usar este código en *driver* de Linux independiente de plataforma

Gestión de RTIs en Linux

Aproximación del kernel Linux

1. Las RTIs se implementan típicamente en el kernel
   - Código específico de arquitectura (`arch/<arquitectura>`)
2. Notificaciones al driver
   - Para ser notificado, un driver ha de registrar un *manejador de interrupción* (función del *driver*) vía `request_irq()`
     - Necesario especificar la línea de interrupción concreta para disp. E/S
     - Manejador se invoca al producirse una interrupción por esa línea
   - El manejador interactúa directamente con el controlador HW del dispositivo de E/S gestionado
3. Controlador de interrupciones se expone al *driver* como una caja negra (Interfaz `irq_chip`)
   - Oculta detalles de implementación

Manejadores de interrupción

```
int request_irq(unsigned int irq, irq_handler_t handler,
        unsigned long flags, const char* name,void *dev);
```

- Descripción parámetros:
  - `irq`: Número de línea de interrupción
  - `irq_handler`: Puntero al manejador de interrupción
    - `typedef irqreturn_t (*irq_handler_t) (int, void*)`
  - `flags`: Máscara de bits con propiedades del manejador
    - `IRQF_DISABLED`: Se ejecuta con todas las interrupciones deshabilitadas en la CPU actual
    - `IRQF_SHARED`: La línea puede compartirse entre dispositivos
    - `IRQF_SAMPLE_RANDOM`: Evento de interrupción usado como semilla para generación de números aleatorios
    - `IRQF_TIMER`: Flag especial para el manejador que gestiona el *system timer*
  - `name`: Nombre del *driver* que gestiona el dispositivo
  - `dev`: Token para identificar qué dispositivo interrumpe

Flujo de procesamiento de interrupciones

Hardware genera interrupción

Controlador de Interrupciones

Procesador

Procesador interrumpe a proceso de usuario o al kernel
Se ejecuta RTI genérica

`do_IRQ()`

¿Hay algún manejador de interrupción instalado en esta línea?

Sí

`handle_irq_event()`

Ejecuta todos los manejadores de esta línea

No

`ret_from_intr()`

Volver al código interrumpido

API kernel

Función	Descripción
`local_irq_disable()`	Desactivar interrupciones localmente en CPU actual
`local_irq_enable()`	Habilitar interrupciones CPU local
`local_irq_save()`	Guardar estado interrupciones y desactivarlas
`local_irq_restore()`	Restaurar estado de las interrupciones a estado antiguo
`disable_irq()`	Desactivar una línea de interrupción específica y esperar a que manejadores activos terminen
`disable_irq_nosync()`	Desactivar una línea de interrupción específica sin esperar a que manejadores activos terminen
`enable_irq()`	Activa una línea de interrupción concreta
`irqs_disabled()`	Devuelve valor distinto de 0 si interrupciones están desactivadas en la CPU local y 0 en caso contrario
`in_interrupt()`	Devuelve valor distinto de 0 si el flujo de ejecución está ejecutando en contexto de interrupción y 0 en caso contrario
`in_irq()`	Devuelve valor distinto de 0 si el flujo de ejecución está ejecutandosé dentro de un manejador de interrupción y 0 en caso contrario

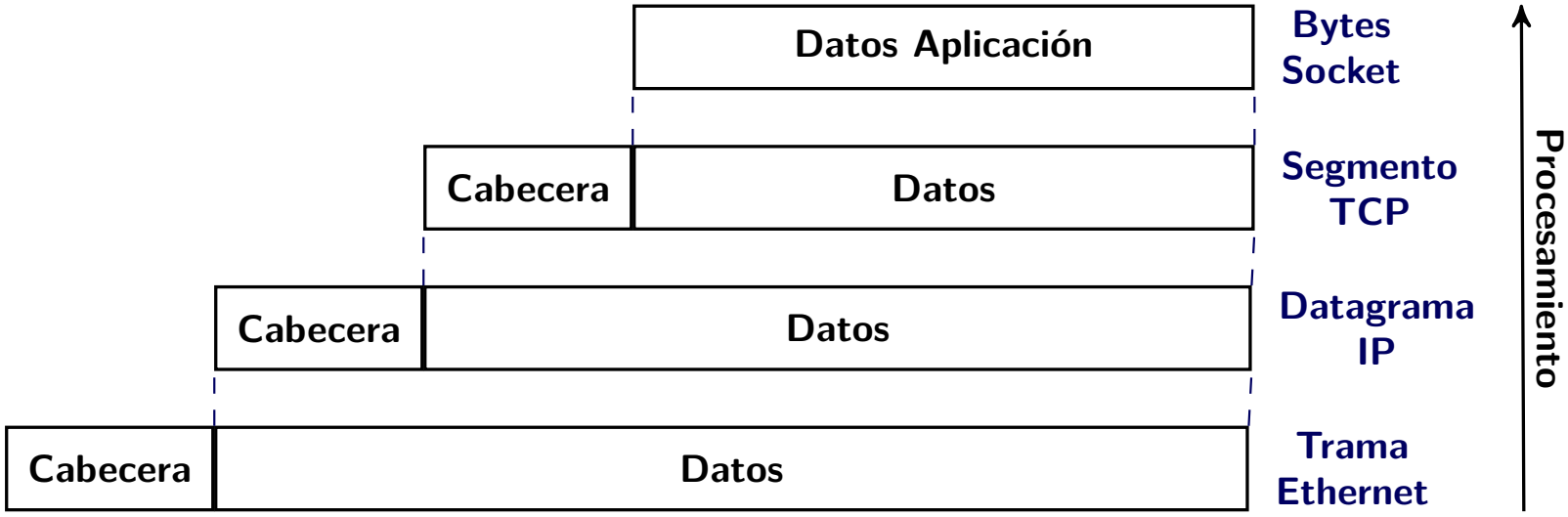
Restricciones manejadores de interrupción

Restricciones

1. **No es posible ejecutar funciones bloqueantes** en un manejador de interrupción
   - Ejecución en Contexto de Interrupción
2. Un manejador ha de **ejecutarse lo más rápido posible**
   - Un manejador puede interrumpir a otro manejador
   - Cada manejador **se ejecuta con una o todas las líneas de interrupción locales deshabilitadas**
     - El HW no puede comunicarse con el SO mientras interrupciones deshabilitadas
     - Crítico para interfaces de red, system timer, ...

Ejemplo: Recepción de datos interfaz de red

- Interfaz de red genera interrupción al recibir un paquete
  - Se ejecuta manejador de interrupción del *driver*

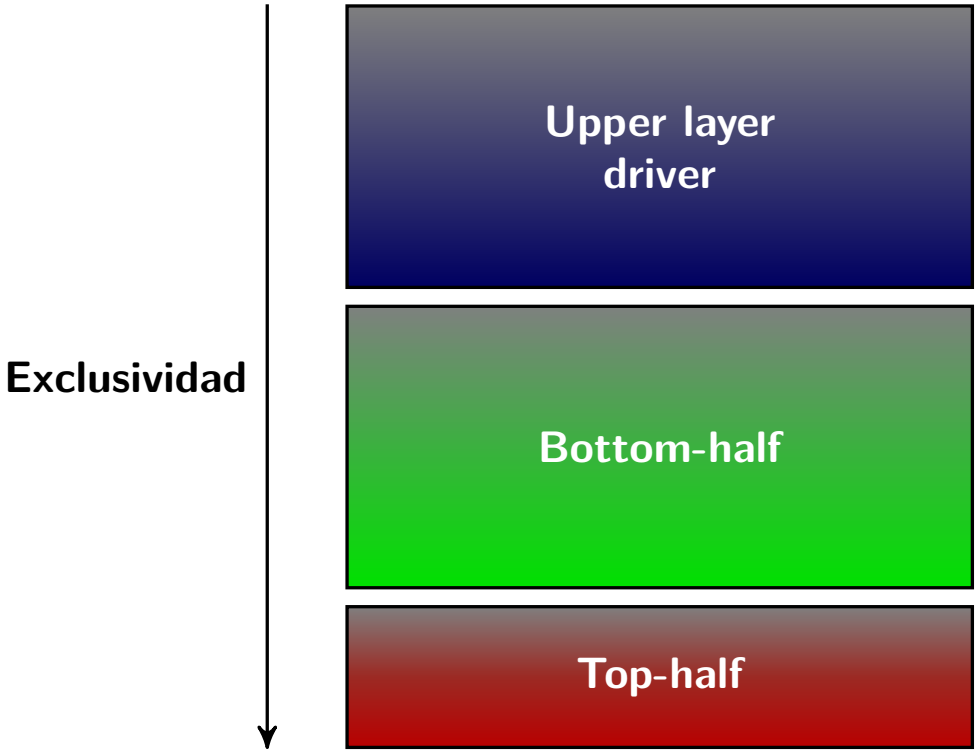


- No es posible realizar todo este trabajo en un manejador de interrupción
  - Interrupciones deshabilitadas → posible pérdida de paquetes

Acciones en manejador de interrupciones

- **Regla general**: realizar sólo lo *indispensable* en el manejador
  1. Interacción directa con el HW gestionado (borrar interrupciones pendientes, leer/escribir registros de datos/control,..)
  2. Aquello que sea crítico en tiempo y no pueda posponerse
- El resto debe posponerse para ser ejecutado en *un mejor momento*
  - Cuando sistema esté menos ocupado procesando interrupciones
  - ... y en un punto de ejecución con interrupciones habilitadas de nuevo
- Exige disponer de mecanismos para diferir el trabajo
  - Mayor parte de SSOO de propósito general los implementan

Estructura *driver*: Top-half y Bottom-half



Interacción con modo usuario
(entradas /proc,
disp. caracteres, ...)

Trabajo planificado por Top-half
(softirq, workqueue, tasklet)

Manejador(es) de Interrupción

Contenido

Mecanismos para diferir el trabajo

- La version 3.14.1 de Linux implementa **3 mecanismos para diferir el trabajo**: softirqs, tasklets y worqueues

Aspectos comunes

- El trabajo diferido (tarea) se modela como una estructura
  - Uno de los campos de la estructura es un puntero a función
  - Almacena dirección de la función que hace el trabajo diferido
- La tarea **se planifica** típicamente **desde manejador de interrupción**
  - Distintas funciones para planificar tareas
- La tarea **se ejecuta** en otro momento **con interrupciones habilitadas**
  - Ejecución tarea → invocación de función correspondiente

Mecanismos para diferir el trabajo

Mecanismo	Ventajas/Inconvenientes	Descriptor tarea	Ejecución
Softirqs	Alto rendimiento Exige modificar el kernel Gestión concurrencia	`softirq_action` `<linux/interrupt.h>`	Contexto de Interrupción[1]
Tasklets	Softirq desde módulos del kernel Gestión concurrencia Escalabilidad limitada	`tasklet_struct` `<linux/interrupt.h>`	Contexto de Interrupción[1]
Workqueues	Ejecución en contexto de proceso Uso sencillo Menor rendimiento	`work_struct` `<linux/workqueue.h>`	Contexto de Proceso

[1] Cuando sistema muy cargado con softirqs, se ejecutan mediante kernel thread `ksoftirqd` (contexto de proceso).

Softirqs

- Las *softirqs* se definen estáticamente en el código del kernel
  - Kernel mantiene array de descriptores `softirq_vec` (`kernel/softirq.c`)
  - Cada softirq tiene asociado un ID (índice) y una función
    - A menor índice, mayor prioridad

Softirq	Índice	Descripción
`HI_SOFTIRQ`	0	Tasklets de alta prioridad
`TIMER_SOFTIRQ`	1	*Kernel timers*
`NET_TX_SOFTIRQ`	2	Envío de paquetes de red
`NET_RX_SOFTIRQ`	3	Recepción de paquetes de red
`BLOCK_SOFTIRQ`	4	Dispositivos de bloque
`BLOCK_IOPOLL_SOFTIRQ`	5	Dispositivos de bloque
`TASKLET_SOFTIRQ`	6	Tasklets de baja prioridad
`SCHED_SOFTIRQ`	7	Equilibrador de carga del planificador
`HRTIMER_SOFTIRQ`	8	Temporizadores de alta resolución
`RCU_SOFTIRQ`	9	Mecanismo de sincronización RCU

Softirqs (II)

Planificar la ejecución de una *softirq*

```
raise_softirq(HI_SOFTIRQ);
```

Posibles puntos de ejecución de una *softirq* (`do_softirq()`)

1. Al volver de un flujo de código que finalizó el procesamiento de una interrupción
2. En cualquier fragmento de código que compruebe explícitamente si hay *softirqs* pendientes e invoque `do_softirq()`
3. Al despertar al kernel thread *ksoftirqd* (baja prioridad)
   - Existe una instancia de este thread en cada CPU
   - En este caso las softirqs se ejecutan en Contexto de Proceso

Tasklets

Tasklets

- Descriptor de tasklet: `tasklet_struct`
  - Función asociada (trabajo diferido)
- El kernel mantiene 2 colas de tasklets: alta y baja prioridad
  - Un tasklet se puede encolar en cualquier cola
    - `void tasklet_hi_schedule(struct tasklet_struct *t);`
    - `void tasklet_schedule(struct tasklet_struct *t);`
- La ejecución de los tasklets gestiona mediante *softirqs*
  - `HI_SOFTIRQ` (alta prioridad) y `TASKLET_SOFTIRQ` (baja prioridad)
  - Al encolar un *tasklet* se activa la softirq correspondiente
  - Softirq ejecuta secuencialmente las funciones de tasklets en la cola

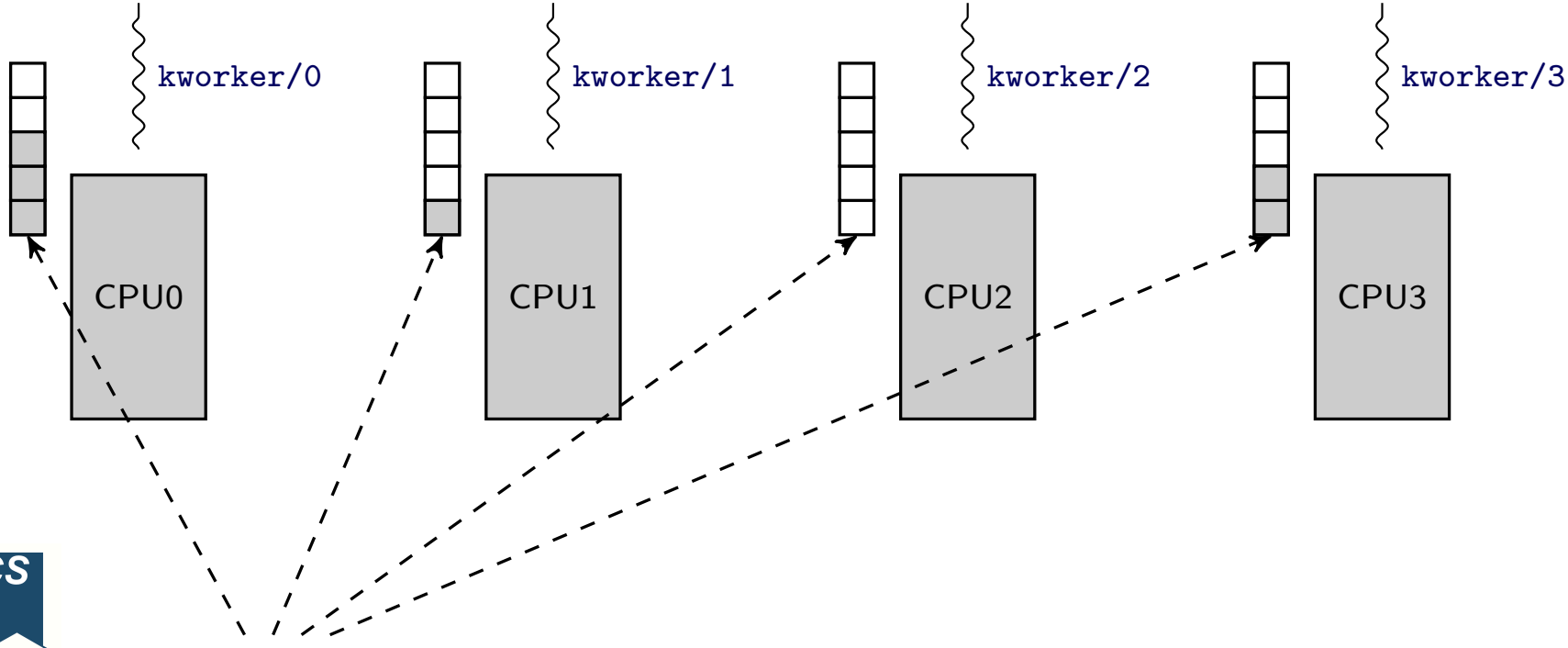
Tasklets (II)

Características tasklets

- A diferencia de las softirqs, es posible crear/destruir *tasklets* dinámicamente
  - No es preciso modificar/recompilar el código del kernel
- Usados frecuentemente para implementación de *drivers*
- No es posible invocar funciones bloqueantes en un *tasklet*
  - Se ejecutan típicamente en contexto de interrupción

Workqueues

- **Workqueue:** Conjunto de colas de tareas (una por CPU)
  - El trabajo que se difiere (tarea) se inserta en una de estas colas
  - Cada tarea se ejecuta en contexto de un proceso (kworker/X)
    - Un *kernel thread* por CPU
    - kworker se *despierta* si hay tareas en cola correspondiente



Workqueues

Worqueues vs. *tasklets* y *softirqs*

- El trabajo diferido puede invocar funciones bloqueantes
  - El `kworker` correspondiente se bloqueará si es necesario
- ... aunque mayor latencia
  - Es preciso despertar *kernel thread* y darle la CPU

Workqueues: `work_struct`

- Cada tarea encolable está descrita mediante `struct work_struct`
  - Definida en `<linux/workqueue.h>`

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
};
```

  - `func` es la función asociada al trabajo diferido

```
typedef void (*work_func_t)(struct work_struct *work);
```

- Una `work_struct` puede encolarse en dos tipos de *worqueues*
  1. Por defecto (`system_wq`)
  2. Worqueue privada del *driver* (creadas explícitamente)
     - Aconsejable si se desea llevar más control sobre tareas planificadas

Workqueue API (I)

Inicializar trabajo

- `INIT_WORK(struct work_struct *work, work_func_t func );`
  - func hace referencia al manejador que se difiere

Planificar Trabajo en *worqueue* por defecto

- `int schedule_work(struct work_struct *work);`
- `int schedule_work_on(int cpu, struct work_struct *work);`
  - Se puede seleccionar en qué CPU

Workqueue API (II)

Crear/Eliminar una *workqueue*

- `struct workqueue_struct *create_workqueue(const char* name );`
- `void destroy_workqueue( struct workqueue_struct * );`

Planificar Trabajo en cualquier *worqueue*

- `int queue_work( struct workqueue_struct *wq, struct work_struct *work );`
- `int queue_work_on( int cpu, struct workqueue_struct *wq, struct work_struct *work );`
  - Se puede seleccionar en qué CPU

Workqueue API (III)

Esperar finalización trabajo pendiente

- `int flush_work( struct work_struct *work );`
  - Esperar finalización de trabajo concreto
- `int flush_workqueue( struct workqueue_struct *wq );`
  - Esperar finalización de todos los trabajos en *workqueue* dada
- `void flush_scheduled_work( void );`
  - Esperar finalización de todos los trabajos en *workqueue* por defecto

Workqueue API (IV)

Cancelar trabajos

- `int cancel_work_sync( struct work_struct *work );`
  - Cancelar trabajo/esperar a que acabe si está en ejecución

Consultar si un trabajo está pendiente

- `work_pending( struct work_struct *work );`

Workqueues: Ejemplo


```c
#include <linux/workqueue.h>

/* Work descriptor */
struct work_struct my_work;

/* Work's handler function */
static void my_wq_function( struct work_struct *work ) {
    printk(KERN_INFO "HELLO WORLD!!\n");
}

int init_module( void ) {
  /* Initialize work structure (with function) */
  INIT_WORK(&my_work, my_wq_function );

  /* Enqueue work */
  schedule_work(&my_work);

  return 0;
}

void cleanup_module( void ) {
  /* Wait until all jobs scheduled so far have finished */
  flush_scheduled_work();
}
```

Contenido

Temporizadores del kernel (I)

- Mecanismo SW para planificar acciones a realizar en un tiempo concreto en el futuro
  - Mecanismo complementario a *bottom halves*

HZ y jiffies

- La macro `HZ` indica el número de *ticks* por segundo
  - Valor por defecto 250 (4ms) - Configurable en T. de compilación
  - Frecuencia del *system timer*
- La variable global `jiffies` almacena el número de *ticks* transcurridos desde el arranque del sistema
  - segundos transcurridos desde arranque: `jiffies/HZ`

Temporizadores del kernel (II)

HZ y jiffies (cont.)

- Los "tiempos de activación" de los temporizadores del kernel se configuran en términos de `jiffies`

```
unsigned long time_stamp = jiffies;          /* now */
unsigned long next_tick = jiffies + 1;       /* one tick from now */
unsigned long later = jiffies + 5*HZ;        /* five seconds from now */
unsigned long fraction = jiffies + HZ / 10; /* a tenth of a second from now */
```

Temporizadores del kernel (III)

- Cada temporizador se describe mediante estructura `timer_list`
  - Declarada en `<linux/timer.h>`

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;  /* Tiempo de 'activación' del timer */
    struct tvec_base *base;
    void (*function)(unsigned long); /* Función que ejecuta el kernel
                                        al activarse el timer */
    unsigned long data; /* Parámetro pasado a la función */
    ...
};
```

- Cuando `jiffies==timer.expires` (temporizador expira) se ejecuta la función correspondiente
  - Ejecución gestionada mediante una *softirq* (`TIMER_SOFTIRQ`)
  - Se ejecutan típicamente en contexto de interrupción
    - ¡¡NO es posible ejecutar funciones bloqueantes!!

Temporizadores del kernel (IV)

Pasos para configurar *timer*

1. Implementar la función asociada al *timer*

```
void my_timer_function(unsigned long data) {...}
```

2. Definir *timer* globalmente:

```
struct timer_list my_timer;
```

3. Inicializar valores internos del *timer*

```
my_timer.expires = jiffies + delay; /* temporizador expira en 'delay' ticks */
my_timer.data = 0;                  /* Ej: 0 (no usado aquí) */
my_timer.function = my_timer_function; /* Función que se ejecutará cuando
                                          temporizador expire */
```

4. Activar el *timer*

```
add_timer(&my_timer);
```

Temporizadores del kernel (V)

- **Otras operaciones sobre *timers***

Función	Descripción
`mod_timer(timer,expiration)`	Modifica la marca de tiempo de expiración de un *timer* activo. Permite reactivar un timer dentro de su propia función de activación (acciones periódicas).
`del_timer(timer)`	Desactiva un *timer* antes de que expire. (No es necesario llamar a esta función para timers que ya han expirado)
`del_timer_sync(timer)`	Desactiva un *timer* y espera a que termine la función asociada. Esta función es más segura que `del_timer()` en entornos multiprocesador y debe usarse siempre para *timers* que se reactiven a sí mismos.

Referencias

- Linux Kernel Development
  - Cap. 7 *"Interrupts and Interrupt Handlers"*
  - Cap. 8 *"Bottom Halves and Deferring Work"*
  - Cap. 11 *"Timers and Time Management"*
- Professional Linux Kernel Architecture
  - Cap. 14 *"Kernel Activities"*
  - Cap. 15 *"Time Management"*

Licencia

LIN - Interrupciones y trabajos diferidos
Versión 0.1

©J.C. Sáez

This work is licensed under the Creative Commons Attribution-Share Alike 3.0 Spain License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/es/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105,USA.

Esta obra está bajo una licencia Reconocimiento-Compartir Bajo La Misma Licencia 3.0 España de Creative Commons. Para ver una copia de esta licencia, visite http://creativecommons.org/licenses/by-sa/3.0/es/ o envíe una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

Este documento (o uno muy similar) está disponible en
`https://cv4.ucm.es/moodle/course/view.php?id=49276`