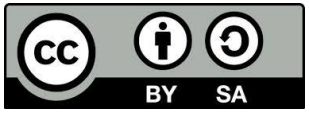
Práctica 4: Trabajos diferidos

DSO - Curso 2013-2014



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

Contenido

1 Introducción

2 Temporizadores

3 Ejercicios

4 Práctica

Contenido

1 Introducción

2 Temporizadores

3 Ejercicios

4 Práctica

Práctica 4: Trabajos diferidos

Objetivos

- Familiarizarse con:
  - Mecanismos para diferir el trabajo en el kernel Linux
  - Uso avanzado de primitivas de sincronización en el kernel
  - Entradas /proc gestionadas vía interfaz `file_operations`
  - Temporizadores

Contenido

1 Introducción

2 **Temporizadores**

3 Ejercicios

4 Práctica

Temporizadores (I)

- Muchas funciones del kernel se invocan periódicamente y no en respuesta a solicitudes expresas del HW o de programas de usuario
  - Ej.: equilibrado de carga, procesamiento de *tick*, ...
- Dos tipos de dispositivo HW para gestión del tiempo:
  1. Real-Time Clock: RTC
     - Dispositivo para almacenamiento no volátil de fecha/hora cuando sistema está apagado
     - El SO lo lee únicamente durante el arranque y, en algunas arquitecturas, se actualiza periódicamente
     - En PC: RTC y CMOS integrados (bateria común)
  2. System Timer
     - Dispositivo que permite generar interrupciones a frecuencia fija (acciones periódicas)
     - En PC: *programmable interrupt timer* (PIT), HPET, local APIC timer

Temporizadores (II)

HZ

- La frecuencia del *system timer* se programa durante el proceso de arranque en base a la macro de preprocesador `HZ`
  - `HZ` indica el número de *ticks* por segundo
  - Valor por defecto 250 (4ms) - Configurable en T. de compilación

```
...
# CONFIG_HZ_100 is not set
CONFIG_HZ_250=y
#CONFIG_HZ_300 is not set
# CONFIG_HZ_1000 is not set
CONFIG_HZ=250
...
```

  - ↓↓ HZ : (+) menor sobrecarga SO (-) Peor tiempo de respuesta
  - ↑↑ HZ : (-) mayor sobrecarga SO (+) Mejor tiempo de respuesta

Temporizadores (III)

jiffies

- La variable global `jiffies` almacena el número de *ticks* transcurridos desde el arranque del sistema
  - segundos transcurridos desde arranque: `jiffies/HZ`
- 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 (I)

- Los temporizadores del kernel constituyen un mecanismo SW para planificar acciones a realizar en un tiempo concreto en el futuro
  - Mecanismo complementario a *bottom halves*: éstas NO permiten especificar cuándo se ejecutará exactamente el trabajo diferido
- 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 */
    ...
};
```

Temporizadores del kernel (II)

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 (III)

- **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. 11 *“Timers and Time Management”*
- Professional Linux Kernel Architecture
  - Cap. 15 *“Time Management”*

Contenido

1 Introducción

2 Temporizadores

3 **Ejercicios**

4 Práctica

Ejercicios (I)

Ejercicio 1

- Analizar el módulo ‘example_timer.c’ que gestiona un temporizador que se activa cada segundo e imprime un mensaje con `printk()`

terminal 1

```
dsouser@debian:~/Ejemplos$ sudo insmod example_timer.ko
```

terminal 2

```
dsouser@debian:~$ sudo tail -f /var/log/kern.log
[sudo] password for dsouser:
...
Jan  4 14:15:22 debian kernel: [233644.504010] Tic
Jan  4 14:15:23 debian kernel: [233645.524021] Tac
Jan  4 14:15:24 debian kernel: [233646.544028] Tic
Jan  4 14:15:25 debian kernel: [233647.564029] Tac
Jan  4 14:15:26 debian kernel: [233648.584021] Tic
Jan  4 14:15:27 debian kernel: [233649.604031] Tac
...
```

Ejercicios (II)

Ejercicio 2

- Estudiar la implementación de los módulos de ejemplo `workqueue1.c`, `workqueue2.c` y `workqueue3.c`, que ilustran el uso de las workqueues

Contenido

1 Introducción

2 Temporizadores

3 Ejercicios

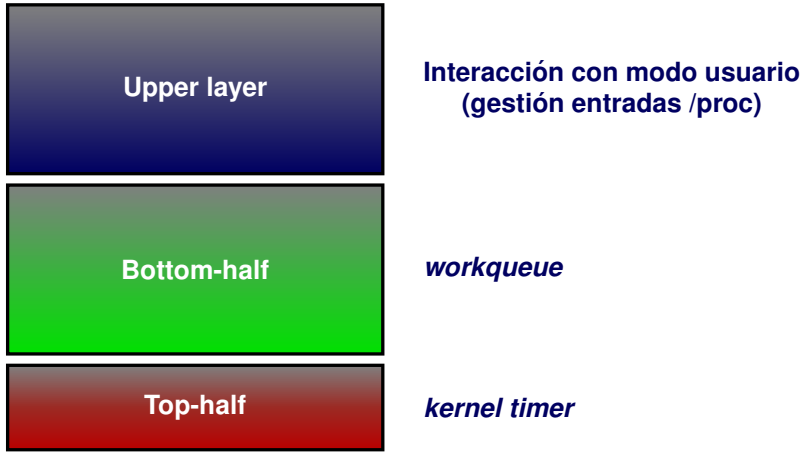
4 Práctica

Especificación de la práctica (I)

- Implementar un módulo `modtimer` que genera una secuencia de números pseudoaleatorios $\in \{0..255\}$
  - Los números se generan periódicamente y se van insertando en una lista enlazada
  - El módulo permitirá que un único programa de usuario "consuma" los números de la lista leyendo de la entrada `/proc/modtimer`
    - El programa se bloqueará si la lista está vacía al hacer una lectura
  - El proceso de generación de números (gestionado mediante un temporizador) se activará sólo cuando un programa de usuario haya abierto entrada /proc
    - Al cerrar la entrada se detendrá el temporizador y se borrarán las estructuras de datos gestionadas por el módulo

Especificación de la práctica (II)

- El módulo consta de tres componentes:
  1. **"Top Half"**: temporizador que genera secuencia de números y los inserta en un buffer circular
     - No es un manejador de interrupción pero se ejecuta en contexto de interrupción
  2. **Bottom Half**: Tarea que transfiere los enteros del buffer circular a la lista enlazada
  3. **Upper Layer**: Implementación de operaciones asociadadas a las entradas /proc exportadas por el módulo



Especificación de la práctica (III)

"Top Half"

- Temporizador que se activa cada `timer_period` *ticks*
- Cada vez que se invoque la función del *timer* se generará un número y se insertará en un buffer circular
  - Generación del número: `int num=jiffies & 0xff;`
  - Necesario implementar buffer circular de enteros (capacidad máxima 10 elementos)
- Cuando el buffer de enteros haya alcanzado un cierto grado de ocupación $\rightarrow$ activar *tarea de vaciado* del buffer (*bottom half*)
  - Parámetro `emergency_threshold` indica el porcentaje de ocupación que provoca la activación de dicha tarea
- La función del *timer* se ejecuta en contexto de interrupción
  - No es posible invocar funciones bloqueantes como `vmalloc()` o `down()`

Especificación de la práctica (IV)

Bottom Half

- Tarea (`struct work_struct`) encolada en workqueue por defecto
  - Se planifica desde la función del timer via `schedule_work()`
- La función asociada a la tarea volcará los datos del buffer a la lista enlazada de enteros
  - Se ha de solicitar memoria dinámica para los nodos de la lista via `vmalloc()`
  - Si el programa de usuario está bloqueado esperando a que haya elementos en la lista, la función le despertará
- Esta función se ejecuta en contexto de proceso en modo kernel
  - Un *kernel thread* se encarga de ejecutarla
  - Es posible invocar funciones bloqueantes siempre y cuando no se haya adquirido un *spin lock*

Especificación de la práctica (V)

Upper Layer

- Código del módulo que implementa las operaciones sobre entradas /proc del módulo
- Dos entradas:
  1. `/proc/modconfig`: permite cambiar/consultar valor de parámetros de configuración `timer_period` y `emergency_threshold`

```
dsouser@debian:~$ cat /proc/modconfig
timer_period=125
emergency_threshold=80
dsouser@debian:~$ echo timer_period 250 > /proc/modconfig
```

  2. `/proc/modtimer`: permite *consumir* elementos de la lista enlazada de enteros
     - El módulo ha de implementar operaciones `open()`, `close()` y `read()`
     - Se usará el campo `struct file_operations* fops` de `struct proc_dir_entry`. Usar este campo inhibe *callbacks* de L/E

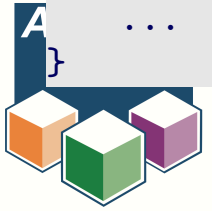
“Interfaz” file_operations

Interfaz de Operaciones

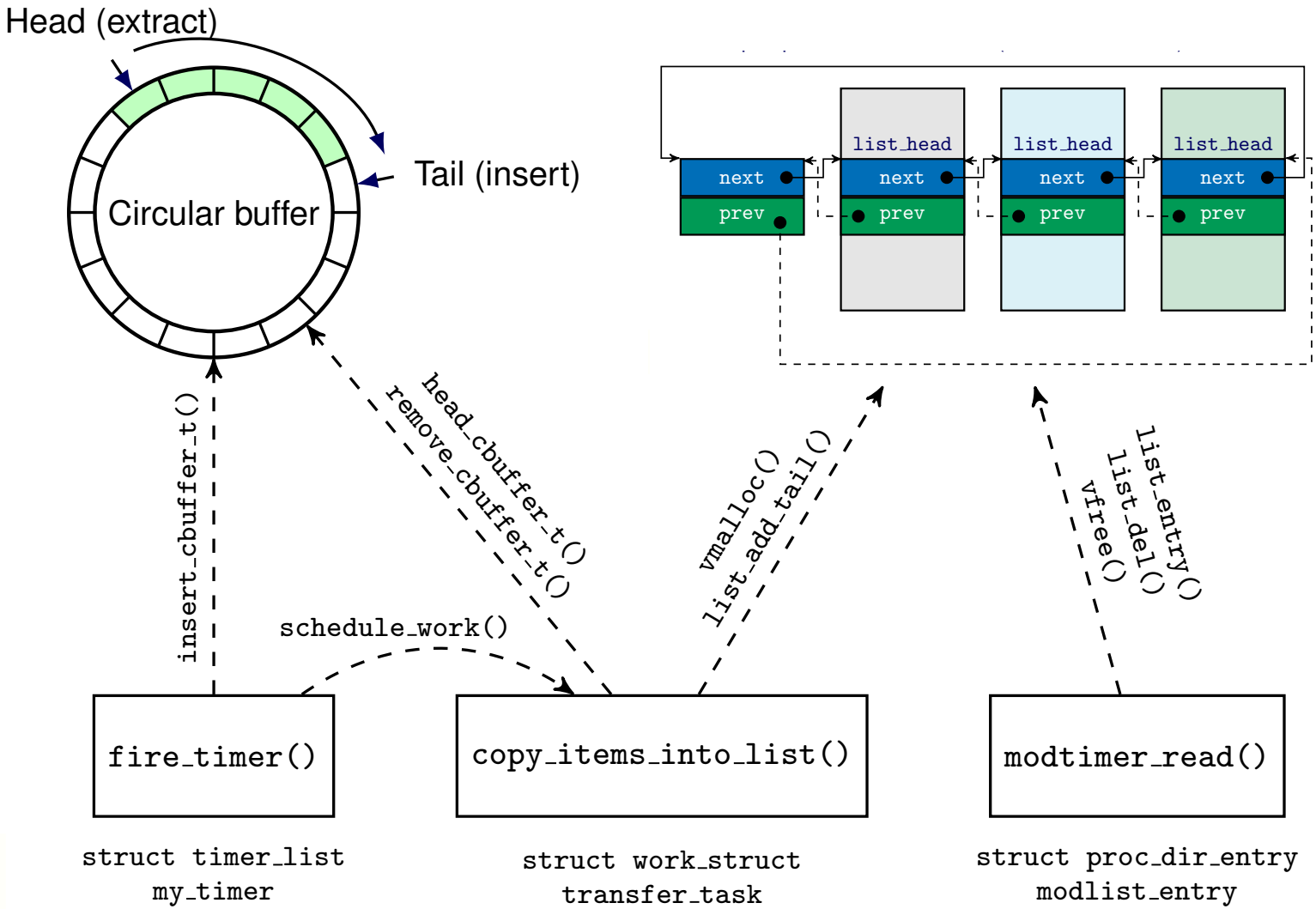
```
struct file_operations {
    struct module *owner;
    loff_t(*llseek) (struct file *, loff_t, int);
    ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t(*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t(*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
....
};
```

/proc: file_operations

```
ssize_t modtimer_read (struct file *filp, char __user *buf, size_t len, loff_t *off){ ... }

int modtimer_open(struct inode *inode, struct file *filp) { ... }

int modtimer_close(struct inode *inode, struct file *filp) { ... }

static const struct file_operations my_fops = {
    .open = modtimer_open,
    .read = modtimer_read,
    .release = modtimer_close,
};


int init_modtimer_module( void ) {
    struct proc_dir_entry *modtimer_entry= create_proc_entry( "modtimer", 0666, NULL );

    if ((modtimer_entry == NULL)) {
            printk(KERN_INFO "modtimer: Can't create /proc entry\n");
            return -ENOMEM;
     }

    timermod_entry->proc_fops=&my_fops;
    ...
}
```



Implementación



Implementación

- Se precisa de **mecanismos de sincronización** para proteger acceso al buffer circular y a la lista enlazada
- El buffer *debe* protegerse mediante un *spin lock*
  - Accedido desde *timer* (cont. interrupción) y otras funciones que se ejecutan en contexto de proceso
  - Necesario deshabilitar interrupciones antes de adquirir el *spin lock*
    - Emplear `spin_lock_irqsave()` y `spin_unlock_irqrestore()`
- La lista enlazada puede protegerse usando *spin lock* o *semáforo*
  - Siempre se accede a la lista desde funciones que se ejecutan en contexto de proceso
  - Por simplicidad/flexibilidad, se recomienda usar un semáforo
- Para bloquear al programa de usuario mientras la lista esté vacía se hará uso de un semáforo (cola de espera)

Implementación

Comentarios adicionales

- El módulo no debe poder descargarse mientras programa de usuario esté usando sus funciones
  - Incrementar el contador de referencias del módulo cuando programa haga `open()`y decrementarlo al hacer `close()`
    - `try_module_get(THIS_MODULE)`
    - `module_put(THIS_MODULE)`
- Realizar las siguientes acciones cuando el programa de usuario cierre `/proc/modtimer`
  1. Eliminar el temporizador con `del_timer_sync()`
  2. Esperar a que termine todo el trabajo planificado hasta el momento en la workqueue por defecto
  3. Vaciar el buffer circular
  4. Vaciar la lista enlazada (liberar memoria)
  5. Decrementar contador de referencias del módulo

Ejemplo de ejecución

terminal1

```
dsouser@debian:~$ sudo insmod modtimer.ko
dsouser@debian:~$ cat /proc/modconfig
timer_period=125
emergency_threshold=80
dsouser@debian:~$ cat /proc/modtimer
61
195
74
208
87
221
100
235
114
249
128
7
141
20
154
33
^C
```

Ejemplo de ejecución (cont.)

terminal2

```
dsouser@debian:~$ sudo tail -f /var/log/kern.log
....
Jan  3 18:13:30 dsouser kernel: [161532.116030] Generated number: 61
Jan  3 18:13:30 dsouser kernel: [161532.652020] Generated number: 195
Jan  3 18:13:31 dsouser kernel: [161533.192020] Generated number: 74
Jan  3 18:13:31 dsouser kernel: [161533.728018] Generated number: 208
Jan  3 18:13:32 dsouser kernel: [161534.268018] Generated number: 87
Jan  3 18:13:32 dsouser kernel: [161534.804023] Generated number: 221
Jan  3 18:13:33 dsouser kernel: [161535.344022] Generated number: 100
Jan  3 18:13:33 dsouser kernel: [161535.884017] Generated number: 235
Jan  3 18:13:33 dsouser kernel: [161535.924681] 8 elements moved from the buffer to
Jan  3 18:13:34 dsouser kernel: [161536.424019] Generated number: 114
Jan  3 18:13:34 dsouser kernel: [161536.964019] Generated number: 249
Jan  3 18:13:35 dsouser kernel: [161537.504026] Generated number: 128
Jan  3 18:13:36 dsouser kernel: [161538.044024] Generated number: 7
...
```

Partes opcionales (I)

- **(Opcional 1)** Reimplementar la práctica con las siguientes variantes:
  1. Hacer uso de workqueues privadas para el módulo en lugar de recurrir a las workqueues por defecto (`schedule_work()`)
  2. Al leer de la entrada `/proc/modtimer`, el módulo devolverá un vector de enteros al programa de usuario en lugar de una secuencia de caracteres
     - Necesario escribir un programa de usuario *ad-hoc* que lea de la entrada /proc, interprete la secuencia de enteros y la imprima por pantalla

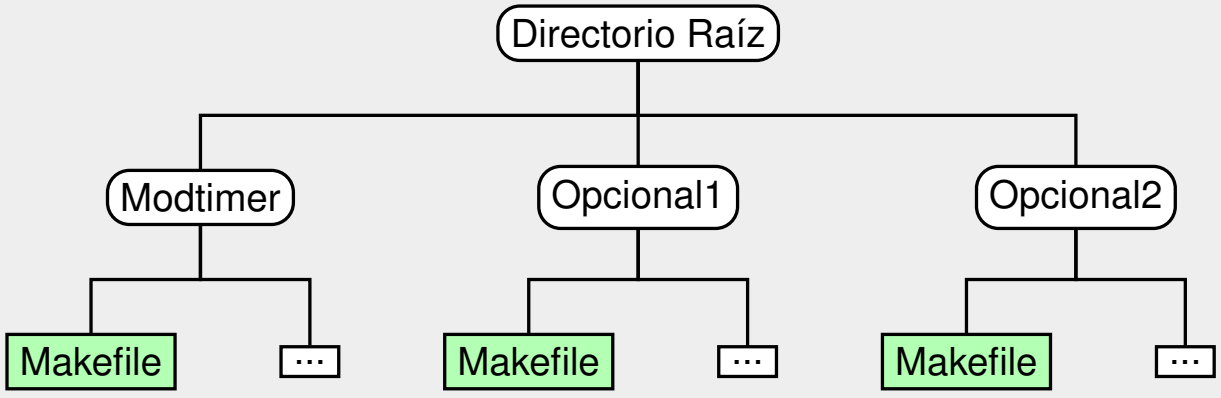
Partes opcionales (II)

- **(Opcional 2)** Implementar una versión alternativa de la práctica en la cual los números pares generados se inserten en una lista enlazada y los pares en otra
  - El módulo permitirá que dos programas de usuario abran la entrada `/proc/modtimer`
  - El primer proceso en abrir la entrada procesará los números impares y el segundo los pares
    - Pista: Para poder distinguir entre ambos procesos puede modificarse el campo `private_data` de `struct file` al abrir el fichero
  - La secuencia de números comenzará a generarse cuando ambos procesos hayan abierto la entrada /proc

Entrega de la práctica

- A través del Campus Virtual
  - Hasta el 31 de Enero
  - No se permiten entregas tardías de esta práctica
- Es aconsejable mostrar el funcionamiento antes de hacer la entrega

Estructura entrega (en un fichero comprimido .tar.gz o .zip)



Licencia

DSO - Práctica 4: Trabajos diferidos
Versión 0.1

Este documento (o uno muy similar) está disponible en
`https://cv2.sim.ucm.es/moodle/course/view.php?id=37161`