Práctica 4: Sincronización en el kernel

Juan Carlos Sáez Alcaide

Índice

1 Objetivos

El objetivo de esta práctica es familiarizarse con las siguientes abstracciones de Linux:

- Spinlocks
- Semáforos
- Wait queues
- Kernel threads
- Buffers circulares del kernel (`struct kfifo`)

2 Ejercicios

Ejercicio 1

El módulo de ejemplo `refmod.c` ilustra la importancia del contador de referencia asociado a un módulo del kernel. Este módulo, al cargarse, expone al usuario un fichero especial de caracteres `/dev/refmod`. Al leer de dicho fichero especial con `cat`, el proceso lector se bloquea durante unos segundos, y a continuación retorna devolviendo 0 bytes en la operación de lectura. El periodo de bloqueo durante la lectura –especificado en milisegundos– es configurable, gracias al parámetro `stime_ms` que exporta el módulo, y que puede establecerse en tiempo de carga en el kernel.

Consultar la implementación de la operación de lectura del fichero especial de caracteres (función `refmod_read()` del módulo). Como puede observarse, el bloqueo se lleva a cabo llamando a la función `msleep()` vista en clase. Responder de forma razonada a las siguientes preguntas:

1. ¿Cuánto dura el periodo de bloqueo si el módulo del kernel se carga sin establecer el valor del parámetro configurable `stime_ms`?
2. Ejecutar el comando `cat /proc/refmod` en un terminal. ¿Qué sucede al intentar interrumpir la ejecución de `cat` con `CTRL + C`? ¿Termina la ejecución de `cat` si se envía una señal SIGTERM o SIGKILL con `kill` a este proceso (usar otra terminal para ello, e incrementar el periodo de bloqueo para tener más tiempo para comprobarlo)? ¿En qué estado se bloquea un proceso cuando la llamada al sistema invocada hace uso de `msleep()`?
3. ¿Cuál es el valor del contador de referencia del módulo del kernel mientras NO se esté accediendo al fichero de dispositivo? Usar el comando `lsmod` para consultar el valor de este contador. ¿Sigue siendo el valor del contador de referencias el mismo mientras ejecutamos `cat /proc/refmod` en otra terminal (y antes de que ese comando termine)?
4. ¿Es posible descargar satisfactoriamente el módulo del kernel mientras éste está en uso? Para comprobarlo pueden abrirse 2 terminales, usando el primero de ellos para ejecutar `cat /proc/refmod`, y el segundo para descargar el módulo (`sudo rmmod refmod`). En caso de que el comando de descarga funcione, comprobar si la ejecución del comando `cat /proc/refmod` termina correctamente, y si se muestran o no mensajes de error en el fichero de *log* del kernel a raíz de haber tratado de descargar el módulo cuando está en uso. ¿Cuál es la causa de este comportamiento?

Modificar el código de `refmod.c` para que el contador de referencia del módulo del kernel se incremente al hacer `open()` en el fichero especial de dispositivo, y se decremente al hacer `close()` del mismo. Recordad que las funciones que permiten incrementar y decrementar el contador de referencia son `try_module_get()` y `module_put()`, respectivamente. Para más información se aconseja consultar cualquiera de las variantes del módulo `chardev` proporcionadas en la práctica anterior, donde se usan estas funciones. Tras llevar a cabo las modificaciones propuestas, comprobar ahora el valor del contador de referencia del módulo del kernel mientras haya un "cat" en curso sobre el fichero de dispositivo. ¿Es posible descargar ahora el módulo mientras un proceso esté leyendo del fichero especial de dispositivo?

Ejercicio 2

Probar el funcionamiento del módulo de ejemplo `kthread-mod.c` , que al cargarse con `insmod` crea un *kernel thread*. Este kernel thread pasa casi todo el tiempo bloqueado, y se despierta cada segundo para imprimir un mensaje en el fichero de *log* del kernel.

Para poder percibir esta impresión periódica del mensaje (tras cargar el módulo) se recomienda abrir una ventana de terminal adicional y ejecutar el siguiente comando: `sudo tail -f /var/log/kern.log`. Este comando permite mostrar en tiempo real los mensajes que se van imprimiendo en el fichero de log del kernel:

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

Ejercicio 3

Analizar el código del módulo de ejemplo `Clipboard-update` que hace uso de *wait queues* para dotar de semántica bloqueante al dispositivo especial de caracteres que el módulo gestiona. Este módulo de ejemplo es una variante del módulo `Clipboard-dev` , que se estudió en la práctica anterior.

Al cargar el módulo `Clibpoard-update` en el núcleo, se crea automáticamente el fichero de dispositivo `/dev/clipboard_update`. A pesar de que la lectura y escritura en este fichero especial de caracteres tiene un comportamiento similar a la entrada `/dev/clipboard` del módulo en el que se basa (consulta y modificación, respectivamente del "clipboard") el fichero especial implementado en este caso tiene semántica bloqueante. Todo proceso que lee de la entrada `/dev/clipboard_update` se queda bloqueado en la operacion `clipboard_read()` hasta que se produce una actualización (escritura) del contenido del "clipboard". El proceso escritor, que invoca `clipboard_write()` es el encargado de despertar a los procesos lectores tras realizar una operación de actualización.

Para probar la funcionalidad de este ejemplo se han de abrir tantas ventanas de terminal como procesos acceden al "clipboard". A continuación se muestra un ejemplo con cuatro terminales. Para observar los bloqueos se aconseja hacer primero tres lecturas con `cat` y a continuación la escritura con `echo`:

Terminal 1

```
pi@raspberrypi:~ $ cat /dev/clipboard_update
new clipboard
```

Terminal 2

```
pi@raspberrypi:~ $ cat /dev/clipboard_update
new clipboard
```

Terminal 3

```
pi@raspberrypi:~ $ cat /dev/clipboard_update
new clipboard
```

Terminal 4

```
pi@raspberrypi:~ $ echo "new clipboard" > /dev/clipboard_update
```

Tras estudiar el código fuente, se ha de proporcionar una respuesta a las siguientes preguntas:

1. ¿Qué sucede si, mientras un proceso está bloqueado en `clipboard_read()`, tecleamos `CTRL + C` en la misma terminal (enviando la señal SIGINT al proceso que ejecuta `cat`)? ¿Podría conseguirse el mismo comportamiento si se hubiera usado `wait_event()` en lugar de `wait_event_interruptible()` en la implementación? Justifica la respuesta.
2. ¿Garantiza la implementación exclusión mutua en el acceso a la variable clipboard? En caso de que no sea así propón una solución que garantice que la implementación sea *SMP-safe*.

3 Desarrollo de la práctica

Esta práctica consta de tres partes: A, B y C.

3.1 Parte A

Realizar una implementación *SMP-safe* de la Práctica 1 (módulo `modlist`) usando *spin locks*. Para ello se ha de garantizar exclusión mutua entre las distintas regiones de código que acceden a la lista enlazada de enteros (estructura compartida). Asimismo el módulo del kernel debe modificarse para impedir que se lleve a cabo su descarga si el módulo está en uso (p.ej., si uno o varios procesos están accediendo a la entrada `/proc/modlist`).

Al desarrollar esta práctica ha de tenerse en cuenta que no es posible invocar funciones bloqueantes, como `kmalloc()`, dentro de `spin_lock()` y `spin_unlock()`.

Para verificar la robustez del código desarrollado ha de crearse uno o varios scripts BASH (a lanzar desde varios terminales simultáneamente) que permitan realizar accesos concurrentes a la lista enlazada. Se deja a elección del estudiante determinar el tipo de procesamiento realizado por el script (o scripts) a desarrollar.

3.2 Parte B

Implementar el módulo `ProdCons` descrito en clase, que gestiona un buffer circular acotado de enteros, y permite a los procesos de usuario insertar y eliminar números en ese buffer realizando escrituras y lecturas en un fichero especial de caracteres (`/dev/prodcons`). Estas operaciones de lectura (eliminación) y escritura (inserción) han de tener semántica productor consumidor:

- Para insertar un número (por ejemplo 7) al final del buffer se debe ejecutar el siguiente comando: `$ echo 7 > /dev/prodcons`
  - Si el buffer circular está lleno, esta operación debe bloquear al proceso, hasta que haya de nuevo hueco para realizar la operación.
- Para extraer el primer elemento del buffer debe realizarse una lectura del fichero especial de caracteres: `$ cat /dev/prodcons`
  - Esta operación bloqueará al proceso lector hasta que haya elementos que consumir del buffer. En dicho instante se debería mostrar el valor extraído del buffer por pantalla (esto requiere rellenar adecuadamente el buffer pasado como parámetro a la operación `read()` del driver).

Ejemplo de ejecución (se asume que buffer puede alojar como mucho 4 enteros)

```
kernel@debian:~/ProdCons$ echo 4 > /dev/prodcons
kernel@debian:~/ProdCons$ echo 5 > /dev/prodcons
kernel@debian:~/ProdCons$ echo 6 > /dev/prodcons
kernel@debian:~/ProdCons$ cat /dev/prodcons
4
kernel@debian:~/ProdCons$ cat /dev/prodcons
5
kernel@debian:~/ProdCons$ cat /dev/prodcons
6
kernel@debian:~/ProdCons$ cat /dev/prodcons
<<proceso se queda bloqueado>>
```

Al igual que en el código a desarrollar en la parte A, se debe garantizar que el módulo NO pueda descargarse si algún proceso de usuario está utilizando sus servicios.

3.2.1 Recursos para la implementación

3.2.1.1 Buffer circular

El buffer circular de enteros gestionado por el módulo se implementará utilizando la estructura de datos `struct kfifo` del kernel, que representa un buffer circular de bytes. Para utilizar esta estructura de datos genérica del kernel ha de incluirse el fichero de cabecera `<linux/kfifo.h>`. Aunque existen varias formas para inicializar la estructura, se recomienda definir una variable global del tipo adecuado:

```
struct kfifo nombre_variable;
```

y a continuación inicializar la estructura con `kfifo_alloc()`. La memoria debe liberarse con `kfifo_free()` preferentemente en la función de *cleanup* del módulo.

En la práctica se reservará espacio para alojar hasta 4 u 8 enteros en el buffer. El tamaño máximo del buffer circular se podrá establecer mediante un parámetro configurable del módulo, a fijar en tiempo de carga.

Como un entero ocupa 4 bytes, las inserciones y eliminaciones de números se realizarán mediante las operaciones de inserción y eliminación múltiple de esta estructura de datos: `kfifo_in()` y `kfifo_out()`.

Ejemplo inserción y eliminación (extración) de entero en `struct kfifo`

```
#include <linux/kfifo.h>

void insertar_entero(struct kfifo* cbuf, int n){
  kfifo_in(cbuf,&n,sizeof(int));
}

int extraer_entero(struct kfifo* cbuf, int n){
  int n;
  kfifo_out(cbuf,&n,sizeof(int));
  return n;
}
```

Breve descripción de operaciones de `struct kfifo` (`pk` denota un parámetro puntero a `struct kfifo`)

Función/Macro	Descripción
`kfifo_alloc(pk,size,mask)`	Inicializa kfifo y reserva memoria para almacenamiento interno. `size` ha de ser potencia de 2. `mask` es el parámetro que se pasa a la llamada subyacente a `kmalloc()` (opciones de reserva de memoria). Pasar `GFP_KERNEL` como tercer parámetro.
`kfifo_free(pk)`	Libera memoria asociada al kfifo
`kfifo_len(pk)`	Devuelve número de elementos en kfifo
`kfifo_avail(pk)`	Devuelve número de huecos libres en kfifo
`kfifo_size(pk)`	Devuelve capacidad máxima de kfifo
`kfifo_is_full(pk)`	Devuelve !=0 si kfifo lleno, y 0 en caso contrario
`kfifo_is_empty(pk)`	Devuelve !=0 si kfifo vacío, y 0 en caso contrario
`kfifo_in(pk,from,n)`	Inserta `n` elementos (bytes) en kfifo. Los bytes se leen del buffer pasado como parámetro (`void* from`)
`kfifo_out(pk,to,n)`	Elimina `n` elementos (bytes) del kfifo. Los bytes eliminados se copian en el buffer pasado como parámetro (`void* to`)
`kfifo_reset(pk)`	Elimina todos los elementos de kfifo (vacía buffer) sin liberar la memoria.

- Para más información:
  1. Consultar la documentación del kernel sobre esta estructura de datos
  2. Consultar código fuente del kernel (`<linux/kfifo.h>`)
  3. Capítulo 6 *"Kernel Data Structures"* de Linux Kernel Development

3.2.1.2 Semáforos

Para implementar las operaciones de bloqueo en esta práctica se emplearán tres semáforos del kernel:

- `mtx`: debe inicializarse a 1, y se usará a modo de *mutex* para garantizar exclusión mutua en el acceso al buffer circular.
- `huecos`: debe inicializarse a 0. Se utilizará para bloquear al productor (proceso que escribe en `/dev/prodcons`) cuando no haya hueco para insertar nuevos elementos en el buffer circular.
- `elementos`: debe inicializarse a la capacidad máxima del buffer (en elementos, no bytes) establecida por el usuario (4 u 8). Se utilizará para bloquear al consumidor (proceso que lee de `/dev/prodcons`) cuando no haya elementos que consumir en el buffer circular.

3.3 Parte C

Crear una variante *SMP-safe* y *thread-safe* del *driver* desarrollado en la práctica 3 ParteC – driver que gestiona el display 7 segmentos de la placa Bee v2.0. Para ello se creará una variante de dicho driver que asegure lo siguiente:

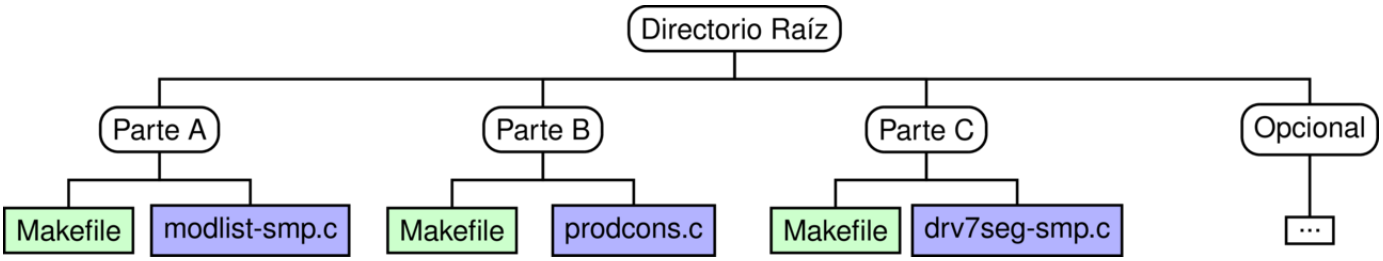
1. El *driver* no debe poder descargarse si algún proceso de usuario está utilizando sus servicios.
2. La implementación debe garantizar que solo un proceso debe poder estar accediendo al fichero de dispositivo `/dev/display7s` en un instante determinado. Para ello, debe mantenerse un contador que lleve la cuenta del número de procesos que están accediendo al dispositivo (a incrementar en `open()`, y decrementar en `release()`). Si un proceso intenta abrir el dispositivo cuando está ya en uso, la operación `open()` devolverá un error (`-EBUSY`). El contador del número de procesos que están usando el fichero especial de dispositivo (valor 0 o 1) debe actualizarse de forma segura usando un recurso de sincronización del kernel. El estudiante puede escoger el tipo de recurso que considere oportuno para permitir la actualización segura del contador.
3. La implementación de la operación `write()` sobre el fichero especial de dispositivo debe ser *thread-safe*; es decir, en caso de que varios hilos de un mismo proceso (con el descriptor compartido) quisieran hacer escrituras simultáneas sobre el fichero especial de dispositivo, debe serializarse la invocación de la función `display7s_write()`, que es la que realiza la modificación del estado del display 7 segmentos. Se deja a elección del estudiante el mecanismo o mecanismos de sincronización a utilizar para imponer esta restricción (que solo un hilo pueda ejecutar `display7s_write()`). **Nota importante**: Se ha de tener en cuenta que la función `msleep()`, que se invoca desde `display7s_write()`, es bloqueante.

3.4 Parte opcional de la práctica

Crear una variante del ejemplo `Clipboard-update` donde se utilicen semáforos del kernel en lugar de *wait queues* para dotar de semántica bloqueante a la operación de lectura en `/dev/clipboard_update`. **Pista**: Se aconseja la utilización de dos semáforos, uno de ellos a utilizar como cerrojo (contador inicializado a 1), y el otro como cola de espera (contador inicializado a 0) para bloquear a los procesos que lean de `/dev/clipboard_update`. Asimismo debe mantenerse un contador global para llevar la cuenta de los procesos esperando en el segundo semáforo.

3.5 Entrega de la práctica

La práctica ha de entregarse a través del Campus Virtual en un fichero comprimido (.tar.gz o .zip) con la siguiente estructura de directorios:



Plazo de entrega: Hasta el **23 de noviembre**.

Es obligatorio mostrar el funcionamiento de la práctica en clase.