Práctica 5: Trabajos diferidos

LIN - Curso 2021-2022

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

Contenido

Contenido

Práctica 5: Trabajos diferidos

Objetivos

- Familiarizarse con:
  - Temporizadores del kernel
  - Mecanismos para diferir el trabajo en el kernel Linux
  - Uso avanzado de mecanismos de sincronización en el kernel

Contenido

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

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

terminal 2

```
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
...
```

Ejercicios (II)

Ejercicio 2

- Modificar el módulo de ejemplo `example_timer.c` para que sea posible trazar (imprimir) usando `bpftrace` el mensaje que el timer imprime periódicamente con `printk()`
  - Para ello se ha de añadir al módulo una función de traza –como las usadas en el artículo *"Introducción a bpftrace"* del campus virtual– que reciba como parámetro el citado mensaje
  - Tras añadir la función de traza, será preciso invocar bpftrace usando un *one-liner* (opción -e) o un script

Ejercicio 3

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

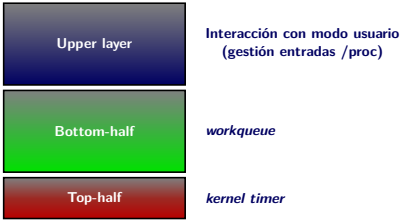
Contenido

Especificación de la práctica (I)

- Implementar un módulo del kernel (`codetimer`) que **genera una secuencia de códigos aleatorios**
  - Los códigos se generan periódicamente y se van insertando en una lista enlazada
- El módulo permitirá que un único programa de usuario **"consuma" los códigos de la lista leyendo de la entrada `/proc/codetimer`**
  - El programa se bloqueará si la lista está vacía al hacer una lectura
  - No se debe permitir que la entrada sea abierta por varios procesos simultáneamente
- El proceso de generación de códigos (gestionado mediante un temporizador) estará activo mientras un programa de usuario esté leyendo de la entrada /proc

Especificación de la práctica (II)

- El módulo constará de tres capas/componentes:
  1. **"Top Half"**: temporizador del kernel que al activarse genera un código aleatorio y lo inserta en un buffer circular acotado
     - No es un manejador de interrupción pero se ejecuta en contexto de interrupción → No es posible invocar funciones bloqueantes
  2. **Bottom Half**: Tarea diferida que transfiere los códigos del buffer circular a la lista enlazada (vacía el buffer)
  3. **Upper Layer**: Implementación de operaciones asociadas a las entradas /proc exportadas por el módulo



Especificación de la práctica (III)

"Top Half" (temporizador)

- 3 parámetros configurables (var. globales modificables vía /proc):
  1. `timer_period_ms`
  2. `code_format`
  3. `emergency_threshold`
- Temporizador se activa cada `timer_period_ms` milisegundos

Especificación de la práctica (IV)

- Formato del código aleatorio establecido por parámetro configurable `code_format`
  - `code_format`: cadena de hasta 8 caracteres formadas por combinaciones de $\{0, a, A\}$:
    - $0 \rightarrow$ número en posición correspondiente
    - a ó A $\rightarrow$ letra minúscula o mayúscula en posición correspondiente
    - Ejemplo: 0000AAA $\rightarrow$ Código aleatorio 5643GTZ
  - Usar función `get_random_int()` que devuelve número aleatorio
    - Declarada en `<linux/random.h>`
    - El mecanismo de generación de códigos ha de invocar el menor número de veces posible a `get_random_int()` (usar bits necesarios de número generado)
- Cada vez que se invoque la función del *timer* se generará un código aleatorio y se insertará en un buffer circular
  - Usar la implementación buffer circular de bytes del kernel (`struct kfifo`)
    - Capacidad máxima del buffer: 32 bytes

Especificación de la práctica (VI)

"Top Half" (cont.)

- Cuando el buffer de códigos haya alcanzado un cierto grado de ocupación → activar *tarea de vaciado* del buffer (*bottom half*)
  - Parámetro `emergency_threshold` indica el porcentaje de ocupación que provoca la activación de dicha tarea
  - La tarea diferida se debe planificar usando una workqueue privada del módulo del kernel (`create_workqueue()`) y deberá ejecutarse en una CPU distinta de donde se ejecutó la función del timer
    - `int cpu_actual=smp_processor_id();`
    - Si CPU actual es par, planificar la tarea en una CPU impar
    - Si CPU actual es impar, planificar la tarea en una CPU par
  - No se debe planificar la tarea diferida de nuevo hasta que no se haya completado la ejecución de la última tarea planificada y el umbral de emergencia se vuelva a alcanzar

Especificación de la práctica (VII)

Bottom Half

- Tarea (`struct work_struct`) que se encolará en una workqueue privada del módulo del kernel
- La función asociada a la tarea volcará los datos del buffer a la lista enlazada
  1. Se extraerán todos los elementos del buffer circular (vacía buffer)
  2. Se ha de reservar memoria dinámica para los nodos de la lista vía `vmalloc()`
  3. 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 (VIII)

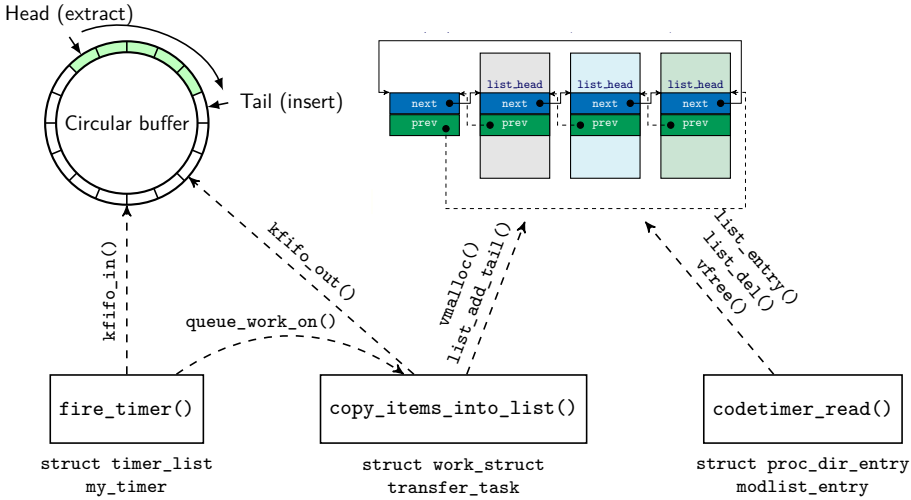
Upper Layer

- Código del módulo que implementa las operaciones sobre entradas /proc del módulo
- Dos entradas /proc:
  1. /proc/codeconfig: permite cambiar/consultar valor de parámetros de configuración timer_period_ms, emergency_threshold y code_format

```
kernel@debian:~$ echo timer_period_ms 500 > /proc/codeconfig
kernel@debian:~$ cat /proc/codeconfig
timer_period_ms=500
emergency_threshold=75
code_format=aA00
```

  2. /proc/codetimer: Al leer de esta entrada (cat) se consumen elementos de la lista enlazada de códigos
     - Implementar operaciones open(), release() y read()

Implementación



Implementación

Necesarios 3 recursos de sincronización

1. 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()`
2. 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
3. Se hará uso de un semáforo (cola de espera) para bloquear al programa de usuario mientras la lista esté vacía
   - Actúa como cola de una "variable condición" que tiene como "mutex" asociado el spin lock o semáforo de la lista enlazada

Implementación

Comentarios adicionales

- No permitir descarga del módulo mientras programa de usuario esté usando sus funciones
  - Incrementar el contador de referencias (CR) del módulo cuando programa haga `open()` y decrementarlo al hacer `close()`
    - Incrementar CR ⇒ `try_module_get(THIS_MODULE);`
    - Decrementar CR ⇒ `module_put(THIS_MODULE);`
- Cuando el programa de usuario cierre `/proc/codetimer` hacer lo siguiente
  1. Desactivar el temporizador con `del_timer_sync()`
  2. Esperar a que termine todo el trabajo planificado hasta el momento en la workqueue creada
  3. Vaciar el buffer circular
  4. Vaciar la lista enlazada (liberar memoria)
  5. Decrementar contador de referencias del módulo

Ejemplo de ejecución

terminal1

```
kernel@debian:~$ sudo insmod codetimer.ko
kernel@debian:~$ cat /proc/codeconfig
timer_period_ms=500
emergency_threshold=75
codeFormat=aA00
kernel@debian:~$ cat /proc/codetimer
bH41
hE63
aY52
zX02
yZ97
zD38
gT55
nJ09
jJ53
hY21
^C
```

Depuración y funciones de traza

- Para demostrar que el módulo funciona correctamente se ha de desarrollar un script de `bpftrace` (*debug.bt*) que use al menos las siguientes funciones de traza, que han de incluirse en el código fuente:
  - `void noinline trace_code_in_buffer(char* random_code, int cur_buf_size) { asm(" "); };`
    - A invocar cuando se inserte un código aleatorio en el buffer circular
  - `void noinline trace_code_in_list(char* random_code) { asm(" "); };`
    - A invocar cuando se transfiera un código del buffer circular a la lista enlazada (es decir, justo después de que se produzca la inserción del código en la lista enlazada)
  - `void noinline trace_code_read(char* random_code) { asm(" "); };`
    - A invocar cuando se extraiga el código de la lista enlazada en la *read* callback de la entrada /proc (es decir, justo después de que se produzca la eliminación del código de la lista enlazada)

Se ha de respetar el nombre y prototipo de las funciones de traza

Ejemplo de ejecución con bpftrace

Ejemplo

```
kernel@debian:~$ sudo bpftrace debug.bt
Attaching 4 probes...
Generated code: bH41
Generated code: hE63
Generated code: aY52
Generated code: zX02
Generated code: yZ97
Code moved to the list: bH41
Code moved to the list: hE63
Code moved to the list: aY52
Code moved to the list: zX02
Code moved to the list: yZ97
Code deleted from the list: bH41
Code deleted from the list: hE63
Code deleted from the list: aY52
Code deleted from the list: zX02
Code deleted from the list: yZ97
...
```

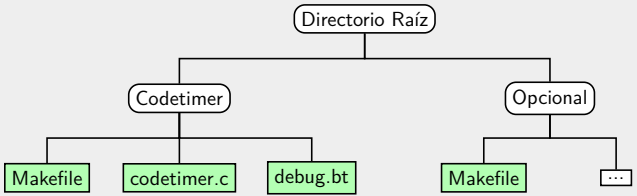
Parte opcional

- Implementar una versión alternativa de la práctica en la cual (1) el formato y longitud de cada código sea aleatorio y (2) Los códigos generados con número de caracteres par se insertarán en una lista enlazada y los impares en otra
  - El módulo permitirá que dos programas de usuario abran la entrada `/proc/codetimer`
  - El primer proceso en abrir la entrada procesará los códigos pares y el segundo los códigos impares
    - Pista: Para poder distinguir entre ambos procesos puede modificarse el campo `private_data` de `struct file` al abrir el fichero
  - La secuencia de códigos comenzará a generarse cuando ambos procesos hayan abierto la entrada /proc
- En la parte opcional, el trabajo diferido (tarea de vaciado del buffer) se encolará en la worqueue por defecto del sistema

Entrega de la práctica

- A través del Campus Virtual → Hasta el 11 de diciembre
- Consideraciones importantes
  - No se permitirán entregas más allá del 17 de diciembre
  - 10% del peso de la práctica → defensa realizada en clase y en plazo
  - Se ha de respetar estrictamente el nombre del módulo (codetimer.ko), el de las entradas '/proc' y los parámetros

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



Licencia

LIN - Práctica 5: Trabajos diferidos
Versión 2.5

Este documento (o uno muy similar) está disponible en https://cvmdp.ucm.es/moodle/course/view.php?id=20152

ArTeCS