Relazione Progetto “Multi-Flow Device File”
Corso Advanced Operating System (and Software Security) – 9 CFU
Prof. Francesco Quaglia

Traccia

This specification is related to a Linux device driver implementing low and high priority flows of data. Through an open session to the device file a thread can read/write data segments. The data delivery follows a First-in-First-out policy along each of the two different data flows (low and high priority). After read operations, the read data disappear from the flow. Also, the high priority data flow must offer synchronous write operations while the low priority data flow must offer an asynchronous execution (based on delayed work) of write operations, while still keeping the interface able to synchronously notify the outcome. Read operations are all executed synchronously. The device driver should support 128 devices corresponding to the same amount of minor numbers.
The device driver should implement the support for the ioctl(..) service in order to manage the I/O session as follows:

- setup of the priority level (high or low) for the operations;
- blocking vs non-blocking read and write operations;
- setup of a timeout regulating the awake of blocking operations.

A a few Linux module parameters and functions should be implemented in order to enable or disable the device file, in terms of a specific minor number. If it is disabled, any attempt to open a session should fail (but already open sessions will be still managed). Further additional parameters exposed via VFS should provide a picture of the current state of the device according to the following information:

- enabled or disabled;
- number of bytes currently present in the two flows (high vs low priority);
- number of threads currently waiting for data along the two flows (high vs low priority).

Installazione

Per installare il modulo viene provvisto un makefile contenente tutte le informazioni utili per la compilazione. Pertanto, per compilare è sufficiente eseguire:

make all

Per installare il modulo, invece, è sufficiente digitare:

sudo insmod my_dev.ko

Inoltre, al fine di poter interagire correttamente col driver è necessario generare i corrispondenti device files all’interno della directory /*dev*/. A tal proposito viene fornito uno script bash che consente di generare il massimo numero di devices supportati dal driver (128). Prima di essere eseguito, tuttavia, è necessario specificare all’interno dello script il MAJOR number associato al device driver. Il MAJOR viene assegnato dinamicamente nel momento in cui il modulo viene inizializzato. Una volta montato il modulo, è possibile accedere al MAJOR mediante il comando:

sudo dmesg

Per eseguire lo script bash, infine, è sufficiente eseguire (dopo avervi inserito il MAJOR):

./user/mk_devs.sh

Utilizzo

Al fine di semplificare l’interazione col device driver, viene anche fornita un’applicazione di livello user tramite la quale è possibile operare sui devices. È possibile compilare l’applicazione eseguendo:

make all

È possibile utilizzare l’applicazione utilizzando la sintassi:

./user.o device (write | read | ioctl) [data]

Di seguito ne vengono riportati alcuni esempi di utilizzo:

./user.o /dev/my_dev0 read
./user.o /dev/my_dev0 write Ciao
./user.o /dev/my_dev0 ioctl set_high_priority_flow
./user.o /dev/my_dev0 ioctl set_timeout 4

Implementazione

L'implementazione fornita del device driver consiste sostanzialmente in un modulo per il kernel Linux, generato mediante l'utilizzo delle librerie <*linux/kernel.h*> e <*linux/module.h*>. Tale modulo utilizza le funzionalità contenute in <*linux/list.h*> al fine di creare due code FIFO - una per il flusso a bassa priorità ed un'altra per il flusso ad alta priorità - per ogni device supportato. Il numero di devices supportati è specificato nella macro *MINORS*. L' elemento fondamentale di una coda è realizzato mediante una *struct node*, contenente un puntatore all'elemento precedente ed a quello successivo, rappresentati da una struttura *list_head*. Inoltre, ciascun nodo contiene un segmento di dati rappresentati da un array di *char*. La dimensione massima del segmento dati è specificato dalla macro *MAX_SEGMENT_SIZE*, il cui valore è attualmente settato a 16 (bytes) al fine di eliminare la frammentazione interna causata dal fatto che la dimensione minima del chunck di memoria allocabile tramite *kmalloc()* è al minimo 32 bytes.
È possibile osservare e/o modificare in tempo reale alcuni dei parametri del modulo tramite dei files presenti nella directory */sys/module/my_dev/parameters*. Tali parametri, realizzati utilizzando la libreria <*linux/moduleparam.h*>, provvedono a fornire informazioni sullo stato dei devices e del driver. In particolare, è possibile osservare quanti threads sono in attesa di operare rispettivamente sulla coda a bassa priorità e sulla coda ad alta priorità di ciascun device mediante il file *waiting_threads_low* e *waiting_threads_high*. Inoltre, è possibile osservare la dimensione (in bytes) dei flussi ad alta e bassa priorità di ciascun device mediante i files *high_flows_size* e l*ow_flows_size*. In questo caso, è opportuno precisare che se un thread scrive un certo numero di bytes x < *MAX_SEGMENT_SIZE* all'interno di uno dei flussi di un qualsiasi device, la dimensione di tale flusso dopo l'operazione di scrittura risulterà comunque essere incrementata di una quantità pari a *MAX_SEGMENT_SIZE*, in accordo col fatto che tali bytes di memoria vengono comunque allocati dal kernel anche se non completamente utilizzati.
Infine, è possibile osservare e modificare lo stato di ciascun device (0 - disabilitato, 1 - abilitato) tramite il file *devices_state*. Inserire un valore diverso da 0 oppure 1 in una entry del file non causa un malfunzionamento del driver; tuttavia, il device corrispondente viene considerato disabilitato.
Il modulo, inoltre, supporta il kernel logging generando messaggi tramite la funzione *printk()* ed utilizzando 3 diversi livelli: *KERN_INFO* (messaggi che indicano il successo di un'operazione effettuata), *KERN_WARNING* (messaggi che indicano il fallimento di un'operazione effettuata, ma che non causano un malfunzionamento del modulo), *KERN_ERROR* (messaggi che causano un malfunzionamento del modulo). È possibile ridurre considerevolmente il numero di messaggi stampati modificando opportunamente la macro *AUDIT*.
Il modulo, infine, provvede a fornire un' implementazione delle funzioni necessarie per operare sui devices. Tali funzioni sono state definite mediante una *struct file_operations*, definita in <*linux/fs.h*>. Di seguito vengono riportati i dettagli riguardanti ciascuna delle funzioni implementate:

- *mydev_open()*, consente di aprire una sessione di I/O con uno dei devices. Il device, tuttavia, deve essere abilitato mediante l'opportuna entry del parametro *devices_state*, altrimenti viene ritornato un errore *EACCESS*;
- *mydev_close()*, consente di chiudere una sessione di I/O con un device;
- mydev_read(), consente di leggere in maniera sincrona un segmento da un device. Il flusso di "default" su cui effettuare le operazioni di lettura/scrittura è memorizzato all'interno dell'array *priority[MINORS]*. In particolare, un thread di livello user che intende leggere un segmento tramite l'invocazione della funzione *read()* viene posizionato all'interno di una

wait queue tramite la funzione *wait_event_idle_exclusive_timeout()*, specificando anche la condizione per la quale attendere ed un valore del timeout. Tutto ciò che riguarda le wait queues è definito in *<linux/wait.h>*. Nel driver è presente una *struct wait_queue_head_t* per ciascun flusso di ogni device, rappresentate dall' array *my_wq[MINORS][FLOWS]*. Se allo scadere del timeout non vi è alcun segmento da leggere nel flusso selezionato (ovvero non si è verificata la condizione *low_flows_size[minor]* > 0 oppure *high_flows_size[minor] > 0*) allora tale funzione ritorna il valore 0, poiché di fatto non è stato letto alcun carattere. Altrimenti, nel momento in cui vi sono uno o più segmenti disponibili nel flusso, il thread viene risvegliato dalla wait queue e legge un segmento dal flusso selezionato. Dopo essere stato letto, tale segmento viene eliminato dal flusso, liberando la memoria utilizzata (tramite la funzione *kfree())* ed aggiornando opportunamente i parametri del modulo. L'aggiornamento dei parametri avviene in maniera atomica utilizzando le funzioni built-in di GCC s*ync_fetch_and_add()* e *sync_fetch_and_sub(),* al fine di eliminare eventuali race conditions causate dall' azione contemporanea di due o più thread operanti su uno stesso device. Infine, viene eseguita un'operazione *wake_up()* poiché a seguito dell'operazione di lettura di un segmento potrebbe essere cambiato il risultato della condizione di uno dei thread posizionati all'interno della wait queue del device. L'invocazione della funzione *wait_event_idle_exclusive_timeout()* consente di porre in stato di sleep il thread chiamante in maniera esclusiva, ovvero settando la *WQ_FLAG_EXCLUSIVE* e non contribuendo al carico della CPU (*TASK_IDLE)*. In questo modo, nel momento in cui viene invocata la funzione wake_up() non vengono risvegliati tutti i thread presenti nella wait queue ma solamente un unico thread con tale flag settata. In questo modo, quindi, si sono evitati effetti che impattano negativamente sulle prestazioni quali ad esempio context switch non necessari e CPU waste. L'intera situazione che si è evitata prende il nome di Thundering Herd Problem. La concorrenza tra operazioni di lettura e scrittura ed in particolare l'inserimento o la rimozione di nodi all'interno delle code che rappresentano i due differenti flussi di ciascun device è gestita mediante spinlocks. Nel driver sono presenti due spinlocks per ciascun device, rappresentati dall'array *my_lock[MINORS][FLOWS]*.

- *mydev_write()*, consente di scrivere in maniera sincrona oppure asincrona un segmento su un device. L'operazione di scrittura avviene in maniera sincrona sul flusso a bassa priorità, mentre avviene in maniera asincrona sul flusso ad alta priorità. Analogamente a quanto riportato in precedenza, un thread di livello user che intende scrivere un segmento in un flusso tramite un' operazione di *write()* viene posizionato all'interno di una wait queue. L'operazione di scrittura avviene solo se valgono contemporaneamente le seguenti condizioni logiche, ovvero, se il numero di bytes da scrivere all'interno di un segmento è inferiore rispetto al limite massimo accomodabile rappresentato dalla variabile *MAX_SEGMENT_SIZE* (altrimenti viene ritornato un errore *EFBIG*) e se vi è spazio sufficiente nel device su cui si intende scrivere o, in altre parole, se vi è spazio sufficiente all'interno della RAM della macchina in cui il modulo è installato (altrimenti viene ritornato un errore ENOSPC). Il chunck di memoria all'interno del quale ciascun segmento è memorizzato viene allocato dinamicamente mediante la funzione *kmalloc()* definita in *<linux/slab.h>*. Il passaggio di dati da user-level a kernel-level e viceversa avviene mediante l'ausilio delle fuzioni *copy_from_user()* e *copy_to_user()* definite in *<linux/uaccess.h>*. L'operazione di scrittura sincrona avviene, in modo molto simile all'operazione di lettura, utilizzando un meccanismo di timeout. Il valore del timeout, in secondi, è rappresentato dall' array *timeout[MINORS]*. L'operazione di scrittura asincrona avviene, invece, mediante l'utilizzo della funzione *wait_event_interruptible_exclusive()*, specificando anche la condizione per la quale attendere indefinitamente. Tale operazione, per ragioni legate alle prestazioni del driver ed alla sicurezza, è stata resa interrompibile da segnali. Infatti, in un contesto multi-threaded, un alto numero di threads scrittori in

esecuzione potrebbe contribuire all' aumento del carico della CPU e quindi impattare negativamente sulle performance del sistema (fino a causare, in casi estremi, un attacco DOS). La condizione che consente il risveglio di un thread scrittore presente nella wait queue è che vi sia spazio a sufficienza per memorizzare almeno un segmento nel flusso selezionato, la cui dimensione massima è rappresentata dalla macro MAX_FLOWS_SIZE (pertanto, la condizione diventa low_flows_size[minor] < MAX_FLOW_SIZE per il flusso a bassa priorità e high_flows_size[minor] < MAX_FLOW_SIZE per il flusso ad alta priorità).

- *mydev_ioctl()*, consente di gestire una sessione di I/O già aperta con un device. In particolare, essa permette di:
    1. Scegliere il flusso sul quale scrivere i segmenti tramite i comandi SET_HIGH_PRIORITY_FLOW e SET_LOW_PRIORITY_FLOW.;
    2. Scegliere in che modo effettuare le operazioni di lettura dai flussi tramite i comandi SET_READ_SYNC e SET_READ_ASYNC. Attualmente, è possibile leggere solamente in maniera sincrona per cui nel secondo caso viene ritornato un errore ENOSYS.
    3. Scegliere in che modo effettuare le operazioni di scrittura sui flussi tramite i comandi SET_WRITE_ASYNC e SET_WRITE_SYNC. Attualmente, il flusso a bassa priorità supporta solamente operazioni di scrittura sincrona, mentre il flusso ad alta priorità supporta solamente operazioni di scrittura asincrona (in accordo a quanto richiesto nella traccia). Ragione per cui, in casi diversi da quelli appena descritti, viene ritornato un errore ENOSYS.
    4. Specificare il timeout in secondi per le operazioni sincrone tramite il comando SET_TIMEOUT.

  Nel caso vengano utilizzati comandi diversi da quelli specificati, viene ritornato un errore EINVAL.

Test

Il device driver è stato testato su una macchina fisica ASUS K55VD avente installati un sistema operativo Ubuntu 20.04 (64 bit) con kernel 5.4.0-107-generic ed un sistema operativo Ubuntu 22.04 (64 bit) con kernel 5.15.0-41-generic. I test effettuati sono rappresentati dal file */user/test.c.* In particolare, esso consente la creazione di 600 Threads lettori/scrittori che operano su ciascuno dei 128 devices supportati dal driver. È possibile modificare il numero di threads da generare tramite la macro *THREADS*, mentre è possibile modificare il tipo di thread (lettore, scrittore) tramite la macro PROBABILITY, la quale rappresenta la percentuale di probabilità che un thread sia un lettore. Pertanto, i threads vengono generati randomicamente, tramite l'utilizzo della funzione *rand()*. Al termine dei test, è stato verificato che non vi siano inconsistenze nei parametri del modulo (es. flussi di dimensione negativa o con dimensione maggiore del massimo consentito) e, analizzando i messaggi in output dal kernel, che i dati vengano scritti nei corrispondenti device e che ciascun thread legga effettivamente il dato più vecchio presente nel flusso selezionato (in accordo con la politica FIFO).