Multi- flow Device Driver

Progetto del corso Advance Operating System – A.A 2021/2022

Martina Salvati
mtr. 0292307
Università degli studi di Roma Tor Vergata
Roma, Lazio, Italia
salvatimartina97@gmail.com

Introduzione

Questo progetto è correlato a un driver di dispositivo Linux che implementa flussi di dati a bassa e alta priorità. Attraverso una sessione aperta al file del dispositivo un thread può leggere / scrivere segmenti di dati. La consegna dei dati segue una politica First-in-First-out lungo ciascuno dei due diversi flussi di dati (bassa e alta priorità). Dopo le operazioni di lettura, i dati di lettura scompaiono dal flusso. Inoltre, il flusso di dati ad alta priorità deve offrire operazioni di scrittura sincrone mentre il flusso di dati a bassa priorità deve offrire un'esecuzione asincrona (basata sul lavoro ritardato) delle operazioni di scrittura, pur mantenendo l'interfaccia in grado di notificare in modo sincrono il risultato. Le operazioni di lettura vengono tutte eseguite in modo sincrono. Il driver di periferica deve supportare 128 dispositivi corrispondenti alla stessa quantità di numeri minori.

Installazione e organizzazione di progetto

Per installare il multi-flow device driver è necessario effettuare il clone dal repository :

https://github.com/msalvati1997/mf_devfile.git

Inoltre è stato implementato uno script (/driver/install.sh) che permette di effettuare l'installazione del device. Dallo script di installazione è possibile richiamare un ulteriore script necessario per l'avvio dei test (test/test.sh).

Descrizione del modulo

Per il progetto è stato implementato un modulo 'multiflow-driver' che contiene la logica di avvio del device driver. Nella 'module_init' vengono inizializzate la struttura dati '**struct device**' per ogni minor. La struttura 'device' contiene :

- Stream ad alta e bassa priorità
- Un oggetto di sincronizzazione (mutex) per ogni stream (alta e bassa priorità)
- Coda di attesa (waitqueue) per ogni stream (alta e bassa priorità)
- Code di lavoro(workqueue) per ogni stream (alta e bassa priorità)
- Numero di byte validi per ogni stream

Parametri del modulo

Il modulo contiene dei parametri, necessari per il soddisfacimento della richiesta.

PARAM NAME	***TYPE***	***DESCRIPTION***
devices_state	array of int	Questo parametro imposta lo stato del file del dispositivo. Se impostato su 0(enable), è possibile creare una nuova sessione. Se impostato 1 (disable), non è possibile creare una nuova sessione.
low_waiting	array of int	Questo parametro indica il numero di threads attualmente in stato di sleep sulla coda di attesa, riservata ai processi che lavorano sul flusso a bassa priorità.
hi_waiting	array of int	Questo parametro indica il numero di threads attualmente in stato di sleep sulla coda di attesa riservata ai processi che lavorano al flusso ad alta priorità.
high_bytes	array of int	Questo parametro indica il numero di byte presenti sul flusso ad alta priorità.
low_bytes	array of int	Questo parametro indica il numero di byte presenti sul flusso a bassa priorità.

Per l'implementazione è stato scelto di utilizzare il param array. Per effettuare la modifica dei parametri devices_state è stata implementata una ioctl() apposita tramite macro che sarà discussa più avanti.

Nella 'module_exit' viene implementata la logica di deallocazione delle strutture dati del driver : streams, code di

attesa, code di lavoro. Inoltre viene effettuato l'annullamento della registrazione del device.

Il Driver

Strutture dati

Multiflow-driver.h contiene tutta la struttura dei dati del driver di periferica.

Quando un programma apre una sessione, è necessario allocare una struttura privata che possa mantenere i parametri impostati dall'utente **(struct _session_data)**:

- La priorità del flusso di lavoro (alta/bassa)
- Il tipo di operazione (bloccante/non bloccante)
- Il timeout per le operazioni bloccanti (espresso in millisecondi o jiffies)

La struttura sessione può essere allocata solo se il parametro devices_state è correttamente impostato su enable, altrimenti non è possibile allocare una nuova sessione.

Per implementare la logica del deferred work sono state utilizzate le **workqueues**. E' stata implementata una struttura (**struct _deferred_work_item)** che contiene i dati necessari per le operazioni di scrittura del flusso a bassa priorità : stream di dati da consegnare e informazioni della sessione privata correlate. Questa struttura rappresenta il lavoro ritardato che dovrà svolgere un kernel thread.

Per implementare la logica delle operazioni bloccanti sono state utilizzate le **waitqueues**.

Waitqueues

Ci sono due code di attesa diverse per ogni stream (bassa/alta priorità). La funzione utilizzata per gestire le code:

wait_event_timeout(wq, condition, timeout);

Il processo viene messo a dormire fino a quando il blocco non viene preso:

*int mutex_trylock (struct mutex * lock);*

Quando il timeout è scaduto, il processo esce dal flusso di esecuzione. Se la condizione viene valutata vera (il lock è libero) prima della scadenza del timeout, il processo continua il flusso di esecuzione.

La *condition* della wait_event_timeout viene verificata ogni qual volta viene effettuato il risveglio dei threads che si trovano nella coda di attesa attraverso la funzione :

void wake_up(struct wait_queue_head);

La funzione wake_up viene chiamata quando viene effettuata un'operazione di *mutex_unlock(struct mutex *lock).* Quando la risorsa risulta disponibile, è possibile risvegliare i threads in attesa sulla coda. In seguito i threads risvegliati provano a prendere il lock. La risorsa che viene contesa rappresenta il mutex – che può essere il mutex dello stream ad alta o bassa priorità. I due streams gestiscono la sincronizzazione in maniera separa, parallelamente. Due processi possono scrivere contemporaneamente nei due diversi stream. Ma due processi che provano a scrivere in uno stesso stream, alta o bassa priorità, si contendono l'acquisizione del lock.

Workqueues

Per implementare il lavoro differito asincrono è stato necessario lavorare con le code di lavoro (workqueue) . Per ogni dispositivo è assegnata una coda di lavoro. E' stata utilizzata l'API :

alloc_workqueue(fmt, bandiere, max_active, args…);

Si è deciso di utilizzare questo tipo di API perché permette di gestire un buon livello di concorrenza. In particolare viene gestito in maniera autonoma dal kernel il pool di kthreads che si occupano dell'esecuzione del deferred work. Il _deferred_work_item viene riempito delle informazioni necessarie a effettuare l'operazione. La seguente funzione viene utilizzata per mettere in coda il __deferred_work_item :

*bool queue_work(struct workqueue_struct * wq, struct work_struct * work);*

Fops

La device driver table implementata per il device :

Driver file operations
static int dev_open(struct inode *, struct file *);
static int dev_release(struct inode *, struct file *);
static ssize_t dev_write(struct file *, const char *, size_t, loff_t *);
static ssize_t dev_read(struct file *filp, char *buff, size_t len, loff_t *off);
static long dev_ioctl(struct file *filp, unsigned int command, unsigned long arg);

Open

L'operazione di apertura del dispositivo assegna una struttura di sessione privata. L'allocazione della struttura è consentita solo se il devices_state dell'oggetto è impostato su ENABLE, altrimenti (devices_state impostato su DISABLE) non è possibile creare nuove sessioni.

Release

Questa operazione viene utilizzata per chiudere un dispositivo specifico e per deallocare i dati privati della sessione.

Write

L'operazione permette di aggiungere allo stream n byte. In base alle impostazioni di sessione si può lavorare sullo stream ad

alta o bassa priorità. L'operazione di scrittura di n bytes comporta l'allocazione di n bytes nello stream. La memoria viene allocata dinamicamente attraverso le funzioni:

*void * krealloc(const void * p, size_t new_size, gfp_t flags); //used to add new bytes*

*void *memset(void *s, int c, size_t n);//used to initialize the memory allocated*

Per copiare i dati dallo spazio utente allo spazio del kernel:

*unsigned long copy_from_user(void * to, const void __user * from, unsigned long n);*

L'operazione di scrittura ha un comportamento diverso in base ai parametri di impostazione di ogni sessione:

Stream ad alta priorità

- OPERAZIONE BLOCCANTI : Le operazioni di blocco funzionano con le code di attesa (waitqueue). Il processo attende finché la condizione è vera e il timeout non è scaduto. La condizione diventa vera quando il lock è stato preso (riservato del flusso di alto livello) ed è possibile effettuare l'operazione.
- OPERAZIONE NON BLOCCANTE : L'operazione non bloccante funziona con l'API mutex_trylock. Se la risorsa è occupata, il controllo ritorna all'utente.

Stream a bassa priorità

Viene allocato un elemento della coda di lavoro differito (workqueue). Quindi il lavoro viene messo in coda alla workqueue dello specifico device.

La funzione deferred_work effettua il lavoro differito in base alle impostazioni di sessione. Le impostazioni di sessione si trovano all'interno della struttura filp ->private_data, passata alla struttura (item) del lavoro differito.

- OPERAZIONE BLOCCANTI : Le operazioni di blocco funzionano con le code di attesa. Il processo è messo in stato di SLEEP fin tanto che la condizione è soddisfatta ed il timeout non è scaduto : l'operazione di scrittura può avvenire. Altrimenti, allo scadere del timeout il processo esce dal flusso di esecuzione.
- OPERAZIONE NON BLOCCANTI : L'operazione non bloccante funziona con l'API mutex_trylock. Se la risorsa è occupata, il controllo ritorna all'utente.

Read

Le letture vengono eseguite in modalità FIFO da sinistra a destra. Quando viene eseguita una lettura, i bytes letti vengono rimossi dal flusso. Per copiare i dati dallo spazio del kernel allo spazio utente:

*unsigned long copy_to_user(void __user * to, const void * from, unsigned long n);*

Le funzioni che permettono di implementare la logica di lettura richiesta:

*void *memmove(void *dest, const void *src, size_t n); // used to copyes n-read bytes*

La memmove permette di copiare i bytes del flusso ad esclusione di quelli letti dalla read all'inizio del flusso. In particolare la memmove permette di implementare la logica FIFO delle letture. Quindi effettua uno *shift* a sinistra dei dati del flusso, trascurando i dati letti a inizio flusso. Quindi poi viene effettuata una pulizia della parte finale dello stream tramite :

*void *memset(void *s, int c, size_t n);//clear the last bytes*

Per effettuare una deallocazione della memoria dello stream degli n bytes letti:

*void * krealloc(const void * p, size_t new_size, gfp_t flags); //used to remove readed bytes*

L'operazione di lettura dipende dalla priorità del flusso di lavoro:

Stream ad alta priorità e Stream a bassa priorità

- OPERAZIONE BLOCCANTE: Le operazioni di blocco funzionano con le code di attesa (waitqueue). Il processo attende finché la condizione è vera e il timeout non è scaduto. La condizione diventa vera quando il lock è stato preso (riservato del flusso di alto livello) ed è possibile effettuare l'operazione.
- OPERAZIONE NON BLOCCANTE : L'operazione non bloccante funziona con l'API mutex_trylock. Se la risorsa è occupata, il controllo ritorna all'utente.

Nel caso dello stream a bassa priorità non vengono utilizzate le code di lavoro, in quanto il lavoro non è mai ritardato.

Ioctl

I parametri del dispositivo dei file possono essere manipolati dalla chiamata di sistema ioctl(). Sono state create alcune macro che semplificano l'impostazione dei parametri (driver/multiflow-driver_ioctl.h).

MACRO	**DESCRIZIONE**
IOCTL_RESET	Consente di impostare i parametri di impostazione predefiniti del file del dispositivo.
IOCTL_HIGH_PRIO	Consente di impostare il flusso di lavoro ad alta priorità.
IOCTL_LOW_PRIO	Consente di impostare il flusso di lavoro a bassa priorità.

IOCTL_BLOCKING	Permette di impostare la modalità bloccante di lavoro.
IOCTL_NO_BLOCKING	Permette di impostare la modalità non bloccante di lavoro.
IOCTL_SETTIMER	Permette di impostare il timer per il blocco delle operazione. Il timeout è espresso in millisecondi.
IOCTL_ENABLE	Consente di impostare lo stato del dispositivo su ENABLE.
IOCTL_DISABLE	Consente di impostare lo stato del dispositivo su DISABLE.

Test

Per eseguire i test avviare l'esecuzione dello script bash ./test.sh. Si aprirà una finestra di dialogo: guiderà l'utente attraverso la selezione degli input dei comandi.

Test 1 : Test semplice

Questo è un semplice test in cui un utente prova un'operazione di scrittura e lettura in diversi flussi (alta e bassa priorità) e diverse sessioni.

Test 2 : Test di scrittura e lettura

Questo è un test in cui un utente prova a scrivere e leggere nella stessa sessione e flussi diversi.

Test 3 : Test di concorrenza

Questo è un test in cui diversi utenti provano a scrivere e leggere in diversi flussi e sessioni diverse.

Test 4 : Test di timeout

Questo è un test in cui diversi utenti provano l'operazione di scrittura e lettura, impostando il parametro del timeout su nsec.

Test 5 : Test enable-disable

Si tratta di un test in cui un utente tenta di scrivere e leggere l'operazione. Successivamente viene disabilitato il dispositivo tramite il comando IOCTL. Dopo la disabilitazione, tenta di scrivere e leggere nuovamente l'operazione. In seguito un altro utente prova a scrivere e leggere dopo la disabilitazione del dispositivo. Le richieste delle operazioni nelle sessioni già aperte dovrebbero essere consentite, mentre le richieste di operazioni nelle nuove sessioni non sono consentite.

Test 6 : Param test

Questo è un test in cui diversi utenti provano a scrivere e leggere l'operazione. Durante il flusso del programma, il comando cat viene chiamato per ogni parametro:

cat /sys/module/multiflow_driver/parameters/##param

Consente di vedere il cambiamento dei parametri del modulo a seguito delle chiamate delle operazioni di scrittura o lettura su flusso.

REFERENCES

[1] https://github.com/msalvati1997/mf_devfile