Progetto Sistemi Operativi Avanzati 2021/2022

Multiflow Device Driver

Danilo Dell’Orco 0300229

­

# Introduzione

Lo scopo del documento è quello di presentare l’implementazione del multiflow\_driver, documentando i componenti del modulo, come interagire con esso, e motivando le scelte progettuali effettuate.

## Specifica – Multiflow Device

La specifica è relativa a un device driver Linux che implementa flussi di dati a bassa e alta priorità. Attraverso una sessione aperta al device file un thread può leggere e scrivere segmenti di dati. La trasmissione dei dati segue una politica di First-in-First-out lungo ciascuno dei due diversi flussi di dati (bassa e alta priorità).

Dopo un’ operazione di lettura, i dati letti 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 su delayed work) delle operazioni di scrittura, mantenendo comunque l'interfaccia in grado di notificare in modo sincrono il risultato. Le operazioni di lettura sono invece eseguite tutte in modo sincrono.

Il driver del dispositivo dovrebbe supportare 128 dispositivi corrispondenti alla stessa quantità di minor number. Il driver del dispositivo dovrebbe implementare il supporto per il servizio ioctl() al fine di gestire la sessione di I/O come segue:

* Impostare il livello di priorità (alto o basso) per le operazioni.
* Operazioni di lettura e scrittura bloccanti o non bloccanti.
* Configurare un timeout che regoli il risveglio delle operazioni bloccanti.

Devono essere implementati anche alcuni parametri del modulo Linux ed alcune funzioni per abilitare o disabilitare il device file, in termini di uno specifico minor number. Se disabilitato, qualsiasi tentativo di aprire una sessione dovrebbe fallire (ma le sessioni già aperte saranno ancora gestite). Ulteriori parametri aggiuntivi esposti tramite VFS dovrebbero fornire un’immagine dello stato attuale del dispositivo in base alle seguenti informazioni:

* Abilitato o disabilitato;
* Numero di byte attualmente presenti nei due flussi (alta o bassa priorità);
* Numero di thread attualmente in attesa di dati lungo i due flussi (alta o bassa priorità).

# Strutture Dati e Parametri

In questa sezione descriviamo le strutture dati utilizzate per rappresentare i principali componenti del modulo. L’effettivo utilizzo di tali strutture e come sono implementate le operazioni sui dispositivi verranno analizzate nelle sezioni successive del documento.

## Stream di Dati

Tramite l’utilizzo del device driver un thread può leggere o scrivere segmenti di dati. Per questo lo stream è implementato come una lista collegata di stream\_block: il contenuto di ogni scrittura viene salvato in un nuovo stream\_block, che viene inserito in coda allo stream. La struttura che rappresenta un blocco di stream è la seguente:

* read\_offset: tiene traccia della posizione dell’ultima lettura all’interno del singolo blocco. Questo perché potrebbero esserci letture che non consumano totalmente i dati di un blocco ed una lettura successiva deve quindi continuare dal primo byte non letto precedentemente.

typedef struct \_stream\_block {

int read\_offset;

char \*stream\_content;

struct \_stream\_block \*next;

int id;

} stream\_block;

* stream\_content: puntatore all’area di memoria che contiene gli effettivi dati scritti sul blocco.
* next: puntatore al blocco successivo dello stream.
* id: codice numerico che identifica il codice nello stream. Non svolge nessuna effettiva funzione in lettura/scrittura, ma è utile nel debugging per analizzare i singoli blocchi.

## Gestione del Dispositivo

Ogni dispositivo è rappresentato in memoria tramite una struttura object\_state, che mantiene tutte le informazioni per poter operare sul device. In totale devono essere gestiti 128 dispositivi, quindi si utilizza un array objects contenente 128 strutture object\_state; objects[N] mantiene lo stato del dispositivo con minor number N.

La struttura è così definita:

typedef struct \_object\_state {

long available\_bytes;

flow\_state priority\_flow[NUM\_FLOWS];

} object\_state;

* available\_bytes: tiene traccia dei bytes liberi sul dispositivo. Questa viene inizializzata al valore MAX\_SIZE\_BYTES, e viene aggiornata a seguito di ogni scrittura e lettura su uno dei due flussi di priorità. Questo campo è fondamentale nelle *deferred write* poiché permette di verificare se c’è spazio disponibile sul device prima di schedulare l’operazione, e riservare dei bytes per tale scrittura, che non potrà fallire. Questo meccanismo verrà descritto in dettaglio successivamente.
* priority\_flow[NUM\_FLOWS]: mantiene i dati scritti sui flussi ad alta e bassa priorità, ed i relativi metadati per la gestione dei blocchi. Si mantengono due strutture distinte in modo da poter operare in parallelo sui due flussi di priorità
  + priority\_flow[0]: flusso a bassa priorità.
  + priority\_flow[1]: flusso ad alta priorità.

Le informazioni sullo stream sono rappresentate in una struttura flow\_state. Questa struttura è definita dai seguenti campi:

* operation\_synchronizer: mutex che permette di sincronizzare le operazioni di lettura e scrittura sul flusso. Il driver può essere utilizzato anche su sessioni di I/O differenti, ed è necessario quindi gestire la concorrenza sul singolo device file per evitare che un solo thread per volta possa operare in lettura/scrittura.

typedef struct \_flow\_state {

struct mutex operation\_synchronizer;

stream\_block \*head;

stream\_block \*tail;

wait\_queue\_head\_t wait\_queue;

} flow\_state;

* head: puntatore al primo blocco dati dello stream. La lettura parte sempre dal blocco puntato da questo campo, seguendo la politica FIFO.
* tail: puntatore all’ultimo blocco dati dello stream. Quando si effettua una scrittura viene appeso un blocco in coda allo stream, e tramite questo campo si velocizza tale operazione poiché evita di dover effettuare la scansione dell’intera lista collegata di blocchi dello stream.
* wait\_queue: Mantiene la Wait Queue del singolo flusso di priorità. Nelle operazioni bloccanti se il mutex non viene disponibile il task viene messo in sleep e inserito in questa Wait Queue. La gestione di questo meccanismo verrà descritta nel dettaglio in seguito.

## Gestione della Sessione

Il driver permette di modificare la sessione di I/O tramite l’utilizzo di ioctl(). Le informazioni della sessione vengono tenute in una struttura session\_state, associata al campo private\_data del file. La struttura è definita come segue:

* blocking: indica se la sessione utilizza operazioni bloccanti o non-bloccanti

typedef struct \_session\_state {

int blocking;

int priority;

int timeout;

} session\_state;

* + 0 (BLOCKING) / 1 (NON\_BLOCKING)
* priority: indica se la sessione opera sullo stream ad alta o bassa priorità.
  + 0 (LOW\_PRIORITY) / 1 (HIGH\_PRIORITY)
* timeout: mantiene il valore del timeout sul lock per le operazioni bloccanti.

## Packed Work Struct

Sul flusso a bassa priorità le scritture avvengono in maniera asincrona, sfruttando le work queue come meccanismo di deferred work. Qui viene utilizzata una struttura packed\_work\_struct, che contiene la work\_struct del lavoro schedulato e altre informazioni sulla scrittura. Quando la scrittura viene schedulata si può risalire dalla work\_struct alla packed\_work\_struct tramite container\_of().

* data: puntatore al buffer kernel temporaneo che mantiene i dati da scrivere sullo stream.

typedef struct \_packed\_work\_struct {

const char \*data;

int minor;

size\_t len;

session\_state \*session;

struct work\_struct work;

} packed\_work\_struct;

* minor: minor number del dispositivo su cui si sta operando.
* len: quantità di byte da scrivere sullo stream.
* session: puntatore alla sessione verso il device file.
* work: struttura work\_struct di deferred work. Viene inizializzata tramite \_\_INIT\_WORK() e schedulato tramite schedule\_work().

## Parametri del Modulo

Il modulo deve esporre diversi parametri tramite VFS per mantere lo stato di un dispositivo. Tali parametri sono stati dichiarati tramite module\_param\_array, definendo così un array di unsigned long con un numero di elementi pari al numero di dispositivi controllati dal driver. In particolare gli array definiti sono i seguenti:

* device\_enabling: specifica nell’elemento i-esimo se il device con minor i è abilitato oppure disabilitato.
  + device\_enabling[i]=0: il dispositivo i-esimo è disabilitato e non è possibile aprire delle sessioni verso il device file.
  + device\_enabling[i]=1: il dispositivo i-esimo è abilitato ed è possibile aprire delle sessioni verso il device file.
* total\_bytes\_low, total\_bytes\_high: specificano nell’elemento i-esimo il numero di bytes disponibili per la lettura sui due flussi di priorità. Il valore di questi parametri incrementati e decrementati a seguito di ogni operazione di read e write
* waiting\_threads\_low, waiting\_threads\_high: specificano nell’elemento i-esimo il numero di thread che sono in attesa per leggere o scrivere dati sui due flussi di priorità. Il valore di questi parametri viene incrementato di una unità ogni volta che un thread si mette in attesa del lock, e viene decrementato ogni volta che un thread acquisisce il lock o scade il suo timeout di attesa.

Tutti i parametri sono accessibili tramite pseudofile nella directory /sys/modules/multistream-driver/parameters. Il parametro device\_enabling viene creato con i permessi di lettura e scrittura (0660) in quanto deve essere possibile abilitare o disabilitare un dispositivo cambiando il valore dell’apposita entry. Invece i restanti parametri hanno soltanto permessi di lettura, in quanto è possibile consultare tali valori, ma non deve essere possibile modificarli manualmente poiché renderebbero inconsistenti le operazioni di lettura e scrittura.

Per abilitare o disabilitare il dispositivo si manipola direttamente tramite la CLI il file /sys/modules/multistream-driver/parameters/device\_enabling. Il file contiene 128 valori binari su una singola linea e separati da una virgola, ognuno associato allo stato di un dispositivo. Si sfruttano quindi getline e fputs per modificare il valore nella posizione 2\*minor associato allo specifico dispositivo che si vuole abilitare/disabilitare.

In alternativa si potrebbe utilizzare ioctl per comunicare con il modulo e modificare direttamente il valore dell’array device\_enabling; tuttavia poiché tale parametro è esposto nel VFS si è deciso di evitare l’interazione con il kernel.

# Funzioni secondarie del Modulo

Prima di descrivere come sono implementate le operazioni di lettura, scrittura e controllo della sessione descriviamo la realizzazione delle seguenti funzioni:

* Inizializzazione del modulo
* Cleanup del modulo
* Apertura di un device
* Chiusura di un device

Queste infatti risultano importanti perché definiscono in qualche modo anche poi il comportamento delle operazioni successive.

## Inizializzazione e cleanup del modulo

Quando si monta il modulo viene invocata la funzione init\_module, definita nel file multiflow\_device.c. In questa funzione viene inizializzato lo stato di tutti i 128 dispositivi, andando a settare i campi della struttura object\_state e dei relativi flow\_state. Inoltre tutti i dispositivi vengono abilitati di default settando device\_enabling[i] = 1.

Successivamente si registra il Char Device tramite \_\_register\_chrdev. Si specifica 0 come major number, per cui questo verrà scelto dinamicamente dal sistema. Il Major assegnato al device viene stampato tramite printk sul buffer del kernel, in modo che l’utente possa recuperarlo per poi lavorare sui dispositivi tramite CLI. Le funzioni del driver vengono specificate tramite la seguente struttura di file\_operations:

static struct file\_operations fops = {

.owner = THIS\_MODULE,

.write = dev\_write,

.read = dev\_read,

.open = dev\_open,

.release = dev\_release,

.unlocked\_ioctl = dev\_ioctl

};

Quando il modulo viene rimosso viene invece chiamata la cleanup\_module. Questa funzione va a rimuovere il device driver tramite unregister\_chrdev e notifica tramite printk l’avvenuta rimozione. VALUTARE COME DEALLOCARE MEMORIA PENDENTE DA READ WRITE, fare un po di kfree insomma.

## Apertura e chiusura di un device (riscrivi bene)

Quando lato utente si apre un dispositivo, lato kernel viene invocata la dev\_open. Quando il dispositivo viene aperto si ricava tramite l’apposita macro il minor number a partire dal file. Si verifica se il file è abilitato o disabilitato e si alloca una struttura session\_state per aprire la sessione verso quel dispositivo. Di default la sessione viene aperta come HIGH\_PRIORITY e NON\_BLOCKING, ma questo può essere cambiato da CLI tramite ioctl.

L’apertura avviene con successo se è stato specificato un dispositivo abilitato e se c’è sufficiente memoria per allocare la sessione. A questo punto si può procedere con le effettive operazioni sul dispositivo.

Quando invece si chiude un dispositivo precedentemente aperto, termina di fatto la sessione di I/O e tramite kfree si libera l’area di memoria precedentemente allocata per la struttura session\_state.

# Operazioni a Bassa Priorità

// Una scrittura low priority non può fallire, quindi il processo attende attivamente di ottenere il lock. Infatti viene controllato prima se c'è spazio disponibile sul device.

// - Anche se non-bloccante, devo notificare in modo sincrono il risultato della write al client.

// - Quindi si cerca di prendere il lock solo quando viene schedulato il lavoro deferred.

// - Non è possibile prevedere se il lock verrà preso e quindi se la scrittura verrà effettuata.

// - Si assume che nessuna scrittura low priority possa fallire.

# Utilizzo del Modulo

## Organizzazione della Repository

La directory principale del progetto è soa-project, che mantiene al suo interno due directory driver e user.

* **/driver**: contiene il codice multiflow\_driver.c del modulo e lo script reinstall\_module.sh per smontare versioni precedenti del driver, ricompilare il modulo e montarlo nuovamente nel kernel.
* **/user**: contiene il codice user\_cli.c e l’eseguibile user\_cli che implementa una semplice CLI per interagire con i dispositivi del driver.

## Montaggio e Rimozione del Modulo

Per installare il modulo si può eseguire lo script driver/reinstall\_module.sh, oppure eseguire manualmente make all per compilare il modulo, e insmod multiflow\_driver.ko per montarlo. Quando il modulo viene montato, nella init\_module viene registrato il char device tramite \_\_register\_chrdev, allocando dinamicamente un nuovo major number per il dispositivo.

Il major number assegnato viene stampato tramite printk sul buffer del kernel, quindi può essere recuperato lato utente con il comando dmesg. si verifica il major number assegnato al driver, stampato tramite printk nella funzione init\_module().

Per rimuovere il modulo è possibile sfruttare il comando rmmod multiflow\_driver, mentre tramite make clean vengono rimossi tutti i file generati in fase di compilazione.

## User CLI

Lanciando tramite sudo il programma user/user\_cli è possibile interagire con il modulo multiflow\_device. Questo accetta due argomenti da riga di comando, che sono:

* major (argv[1]): Major number del device installato, che deve essere recuperato tramite dmesg.
* device\_path (argv[2]): Percorso del VFS dove verranno installati i dispositivi. Se non viene passato questo parametro si utilizza un path di default (/dev/mflow-dev).

Prima di operare con il Char Device è necessario creare i device file che rappresentano i dispositivi sul VFS. Questo può essere fatto:

* Manualmente tramite mknod dev/nome\_device MAJOR MINOR
* Utilizzando il comando 11 *(Create device nodes)* da user\_cli, che genera automaticamente *128* file sempre utilizzando mknod. I file avranno tutti il major number passato da riga di comando, minor numbers da *0* a *127*, e saranno nominati c.

### Operazioni sui device

Descriviamo le operazioni offerte dalla CLI per operare con un dispositivo.

* **Open a device file** (**0**): Chiede all’utente di inserire un minor number N e viene aperto il relativo file /dev/mflow-devN.
* **Write on the device file** (**1**): Effettua la scrittura sul file aperto tramite la system call write().
* **Read from the device file** (**2**): Effettua la lettura dal file aperto, tramite la system call read().

### Operazioni sulla sessione

Tramite CLI è possibile modificare alcuni parametri della sessione, che vanno a cambiare il comportamento delle operazioni di write/read. Tutti questi comandi fanno utilizzo di ioctl per andare ad operare sullo stato della sessione.

* **Switch to LOW/HIGH priority** (**3**/**4**): Modifica il parametro priority della sessione, cambiando quindi il flusso dati da HIGH a LOW o viceversa.
* **Use BLOCKING/NON-BLOCKING operations** (**5**/**6**): Modifica il parametro blocking della sessione, cambiando quindi il tipo delle operazioni successive da non-bloccanti a bloccanti o viceversa.
* **Set timeout** (**7**): Modifica il timeout di attesa del lock nelle operazioni bloccanti.

### Gestione dei dispositivi

Tramite VFS vengono esposti diversi parametri che rappresentano lo stato del dispositivo, che possono essere letti o manipolati direttamente dalla CLI.

* **Enable/Disable a device file** (**8**/**9**): Richiede un minor number all’utente e abilita o disabilita il dispositivo associato a quel minor. Per fare ciò scrive il valore 0 o 1 nel file /sys/module/multiflow\_driver/parameters/device\_enabling, nella posizione specifica associata al dispositivo.
* **See device status** (**10**): Visualizza tutte le informazioni sullo stato di un dispositivo, specificato dall’utente tramite il minor number. Si accede in lettura ai parametri del modulo /sys/module/multiflow\_driver/parameters/, mostrando le seguenti informazioni:
  + Stato del dispositivo: ENABLED/DISABLED
  + Spazio libero disponibile in bytes.
  + Bytes presenti nei due flussi ad alta e bassa priorità
  + Thread in attesa sui due flussi ad alta e bassa priorità
  + Valore del timeout per il lock in millisecondi.