Progetto di Sistemi Operativi Avanzati

A. Chillotti*

A.A. 2021/2022

1 Traccia del progetto (traduzione)

Questa specifica è legata ad un driver Linux che implementa flussi di dati a priorità bassa e alta. Attraverso una sessione aperta al devide file, un thread può leggere/scrivere segmenti dati. La consegna dei dati segue una policy First-in-First-out lungo ciascuno dei due diversi flussi di dati (bassa e alta priorità). Dopo le operazioni di lettura, i dati devono scomparire dal flusso. Inoltre, il flusso dati di alta priorità deve offrire operazioni di scrittura sincrone, mentre il flusso dati di bassa priorità deve offrire una esecuzione asincrona (basata su delayed work) delle operazioni di scrittura, pur mantenendo l'interfaccia in grado di notificare in modo sincrono l'esito. Le operazioni di lettura sono tutte eseguite sincronamente. Il device driver supporta 128 device corrispondenti alla stessa quantità di minor number. Il device driver deve implementare il supporto per il servizio `ioctl(..)` in modo tale da gestire la sessione di I/O come segue:

- setup del livello di priorità (alto o basso) per le operazioni;
- operazioni di lettura e scrittura bloccanti vs operazioni di lettura e scrittura non bloccanti;
- setup del timeout che reluga il risveglio delle operazioni bloccanti.

Alcuni parametri e funzioni del modulo Linux dovrebbero essere in grado di abilitare o disabilitare il file del dispositivo, in termini di specifico minor number. Se è disabilitato, un tentativo di apertura della sessione deve fallire (ma le sessioni già aperte devono essere ancora gestire). Ulteriori parametri esposti via VFS devono fornire un'immagine dello stato corrente del device in accordo alle seguenti informazioni:

- abilitiato o disabilitato;
- numero di byte correntemente presente nei due flussi (alta e bassa priorità);
- numero di thread correntemente in attesa di dati lungo i due flussi (alta e bassa priorità).

2 Relazione del progetto svolto

2.1 Rappresentazione dei multi-flow device file

La rappresentazione dei multi-flow device file è definita dalla struttura `object_t` che contiene:

- un puntatore ad una workqueue, ovvero `struct workqueue_struct *workqueue`. Il puntatore alla workqueue è stato inserito all'interno della struttura che rappresenta il multi-flow device file perché è utile solo nel caso in cui si sta lavorando a `LOW PRIORITY`.
- due puntatori ad un buffer, ovvero `dynamic_buffer_t *buffer`.

*alessandro.chillotti@outlook.it

2.2 Rappresentazione del buffer

La rappresentazione del buffer è definita dalla struttura `dynamic_buffer` che contiene:

- un puntatore `head` di tipo `list_head`;
- un mutex che permette di sincronizzare le operazioni relative al buffer;
- una waitqueue.

Quindi, l'idea è che il multi-flow device driver si appoggi su un ulteriore oggetto, il `dynamic buffer`, che ci occupa di scrivere e leggere segmenti dati.

Si può notare come siamo presenti `256` waitqueue, ovvero una per ogni buffer. Questa scelta è stata fatta perché il lavoro svolto su un altro buffer è sicuramente indipendente dal lavoro fatto su un altro buffer. Inoltre, nella funzione di inizializzazione del modulo vengono create delle singlethread workqueue. In questo modo il deferred work viene processato lungo un unico kworker deamon ed esso riporta le operazioni di scrittura sul buffer di bassa priorità nel medesimo ordine con cui sono state schedulate, senza dover implementare meccanismi di ordinamento e sincronizzazione.

2.3 Parametri del modulo

Sono stati definiti dei parametri del modulo ed essi sono stati utilizzati all'interno del driver come variabili operative. In particolare sono stati definiti i seguenti parametri:

- `enabled`: un vettore di `128` elementi di tipo `bool`, ove ogni elemento `i` indica se il multi-flow device file relativo al minor number `i` è attivo o meno;
- `byte_in_buffer`: un vettore di `256` elementi di tipo `long`, ove ogni elemento riporta il numero di byte presente all'interno del buffer relativo ad uno specifico minor number, in particolare:
  - i primi `128` elementi sono relativi al buffer di bassa priorità del multi-flow device file con minor number `i`;
  - i secondi `128` elementi sono relativi al buffer di alta priorità del multi-flow device file con minor number $(\texttt{i} - 128)$.
- `thread_in_wait`: un vettore di `256` elementi di tipo `long`, ove ogni elemento riporta il numero di byte presente all'interno del buffer relativo ad uno specifico minor number, secondo la medesima regola di `byte_in_buffer`.

La scelta di creare un vettore di `256` elementi anziché `128` è stata fatta perché ha permesso la realizzazione di un codice migliore dal punto di vista della leggibilità. Questo perché, indipendemente dalla priorità, prima di andare ad effettuare una scrittura o una lettura si effettua un controllo sul contento del buffer (i.e. spazio libero in caso di scrittura, byte nel buffer in caso di scrittura). Infatti, nel caso in cui si fossero creati due vettori da `128` elementi si sarebbero dovuti differenziare i casi. A tal proposito è stata realizzata la seguente macro:

```
1 #define get_byte_in_buffer_index(priority, minor)   \
2         ((priority * MINOR_NUMBER) + minor)
```

Inoltre, si può notare come in questo modo non venga effettuato nessun check, ma una semplice operazione matematica.

Per lavorare con i parametri del modulo sono stati sviluppati degli script bash:

- lo script `enable_set` consente di settare l'abilitazione del multi-flow device per un certo minor;
- lo script `enable_query` consente di sapere se il multi-flow device è abilitato o meno per un certo minor;
- lo script `byte_query` consente di sapere, data la priorità ed il minor, il numero di byte presente nel buffer associato;
- lo script `thread_query` consente di sapere, data la priorità ed il minor, il numero di thread in attesa sul flusso associato.

2.4 Operazioni sui multi-flow device file

Nelle seguenti sottosezioni sono descritte le funzioni che compongono il driver.

2.5 Operazione d'apertura

La funzione `dev_open` consente l'apertura di una sessione per poter lavorare con un determinato multi-flow device file. In questa funzione vengono effettuate le seguenti operazioni:

1. Viene controllato che il `minor` inserito sia effettivamente minore o uguale rispetto al `minor` massimo gestibile dal driver.
2. Viene controllato la componente del vettore `enabled`, ovvero si controlla che il `minor` verso il quale si vuole aprire una sessione abbia il multi-flow device controller effettivamente abilitato.
3. Nel caso in cui entrambi i controlli abbiano avuto esito positivo, si prepara la struttura ad-hoc `session_t` settando di default i seguenti parametri:
   - come priorità viene inserita `HIGH PRIORITY`;
   - viene settata come sessione bloccante, ovvero viene settato il campo `flags`[1] a `GFP_KERNEL`;
   - viene settato il timeout a `MAX_SECONDS`.
4. viene collegata questa istanza `session_t` alla componente `file->private_data`.

È importante precisare che la macro `MAX_SECONDS` è stata settata al valore `17179869` perché, nel momento in cui si inserisce il valore da attendere all'interno della `wait_event_interruptible_exclusive_timeout`[2], bisogna inserire il numero di secondi moltiplicato per la costante `HZ`, ove in questo caso è pari a `250`. Il ragionamento effettuato è il seguente:

$$2^{32} = 4294967296 \rightarrow \frac{4294967296}{250} = 17179869.184 \rightarrow \texttt{MAX\_SECONDS} = 17179869$$

In questo modo si evita l'overflow nel momento in cui si effettua l'operazione di settaggio del timeout.

2.6 Operazione di rilascio

La funzione `dev_release` consente il rilascio della sessione per poter lavorare con un determinato multi-flow device file. In questa funzione viene effettuata la `kfree` di ciò presente all'indirizzo inserito in `file->private_data`, quindi dell'indirizzo della `session_t` preparata all'interno dell'operazione d'apertura.

2.7 Operazione di scrittura

La funzione `dev_write` consente la scrittura del contenuto di un buffer di livello user all'interno del buffer di livello kernel. In questa funzione avvengono le seguenti operazioni:

- Una prima fase di allocazione di aree di memoria:
  1. Viene allocato un buffer temporaneo di livello kernel e subito dopo viene effettuato la `copy_from_user`. Questo viene fatto prima di prendere il lock perché, nel caso in cui l'utente avesse dato un'area di memoria non materializzata, si sarebbe dovuto attendere il gestore di page fault e avrebbe causato problemi di prestazioni.
  2. Viene allocata l'area di memoria per il `data_segment` corrente, ovvero l'oggetto da inserire all'interno del `dynamic buffer`.
  3. Nel caso in cui la priorità sia bassa, viene allocata l'area di memoria per poter inserire deferred work.

[1] In caso di operazione non bloccante, si avrà settato questo campo come `GFP_ATOMIC`.
[2] Questa macro è stata sviluppata durante la stesura del progetto, più avanti verrà analizzata e motivata.

Come si può notare, questa fase viene effettuata prima di provare ad acquisire il lock in modo tale che non ci possano essere problemi di allocazione nel momento in cui il lock venga acquisito.

- Una seconda fase in cui si cerca di acquisire il lock e ci sono due casi:
  - Nel caso in cui le operazioni sono bloccanti:
    1. Viene incrementata la variabile che tiene conto dei thread in attesa.
    2. Si va nella `wait_event_interruptible_exclusive_timeout` con la condizione che è dettata dalla macro `lock_and_awake`.
       La macro `lock_and_awake` è la seguente:

```
#define lock_and_awake(condition, mutex)        \
({                                              \
        int __ret = 0;                          \
        if (mutex_trylock(mutex)) {             \
                if (condition)                  \
                        __ret = 1;              \
                else                            \
                        mutex_unlock(mutex);    \
        }                                       \
        __ret;                                  \
})
```

Snippet 1: Macro `lock_and_awake`

       Questa macro ritorna `0` se il thread non è riuscito a prendere il lock o, nel caso in cui ci fosse riuscito, non abbia soddisfatto la condizione. In questo caso la condizione riguarda il fatto che nel buffer ci sia spazio o sia interamente occupato.
       Da notare che viene chiamata la `wait_event_interruptible_exclusive_timeout`[3] che non è definita nel kernel Linux, ma questo è stato fatto perché serviva una soluzione che potesse risolvere i seguenti problemi:
       * Utilizzando `wait_event_interruptible_timeout` internamente i thread si sarebbero svegliati tutti e poi avrebbero controllato la condizione, ma solamente uno riesce ad uscire dalla wait e quindi questo avrebbe provato uno spreco di cicli di CPU.
       * Utilizzando `wait_event_interruptible_exclusive` si sarebbe dovuto implementare un meccanismo separato di gestione del timeout andando ad aggiungere complessità al prodotto software.

       Si è pensato che in questo modo la complessità del prodotto software non sia stata alterata e che risulti anche abbastanza elegante per la risoluzione del primo problema. Infatti, è stata modificata la funzione `__wait_event_interruptible_timeout`, in particolare:

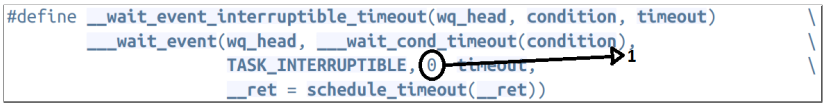

Figura 1: Modifica della `wait_event_interruptible_timeout`

       In relazione alla documentazione (link, *6.2.5.3 Exclusive waits*), si può notare che gli sviluppatori del kernel hanno aggiunto un'opzione di "attesa esclusiva" al kernel. Un'attesa esclusiva si comporta in modo molto simile a un normale sonno, con due importanti differenze:
       * Quando una voce della coda di attesa ha il flag `WQ_FLAG_EXCLUSIVE` impostato, viene aggiunta alla fine della coda di attesa. Le voci senza quel flag vengono, invece, aggiunte all'inizio.
       * Quando `wake_up` viene chiamato su una coda di attesa, si interrompe dopo aver riattivato il primo processo con il flag `WQ_FLAG_EXCLUSIVE` impostato.

       In figura 1 si può notare come ogni thread venga mandato a dormire con il flag `WQ_FLAG_EXCLUSIVE` impostato, ma inoltre rimane valido lo `schedule_timeout`.
    3. Quando il thread riprende l'esecuzione decrementa la variabile precedentemente incrementata e, se si è svegliato perché `lock_and_awake` ha ritornato `1`, allora prosegue l'esecuzione nella terza fase.
  - Nel caso in cui le operazioni non sono bloccanti:
    1. Si prova a prendere il lock ed in caso negativo si rilasciano le aree allocate e si ritorna `EBUSY`.

[3] Le macro chiamate da essa, ovvero la `__wait_event` e la `__wait_cond_timeout`, sono definite a parire dal kernel 3.13, infatti all'interno del codice è presente il check sulla versione e, se inferiore a 3.13, viene chiamata semplicemente la `wait_event_interruptible_timeout`.

2. Se il lock è stato acquisito si controlla se il buffer è vuoto, in tal caso si rilasciano le aree allocate, il lock e si chiama la `wake_up_interruptible`.

- La terza fase riguarda la fase di scrittura del segmento dati e, dopo aver fatto un controllo sulla dimensione dei dati da scrivere e fatto la `init_data_segment`, ci sono due casi:
  - Nel caso di `HIGH PRIORITY`:
    1. Viene invocata la `write_dynamic_buffer` sull'oggetto `dynamic_buffer`, quindi viene effettuata la scrittura.
    2. Viene incrementato il numero di byte nel buffer e viene invocata la `wake_up_interruptible`.
  - Nel caso di `LOW PRIORITY`:
    1. Si chiama la `try_module_get`.
    2. Si collega solamente[4] il contenuto da scrivere al deferred work.
    3. Si chiama `__INIT_WORK`.
    4. Si incrementano i byte prenotati.
    5. Si accoda il lavoro nella workqueue.
- L'ultima fase si occupa solo del rilascio del lock acquisito e ritorna i byte scritti.

2.7.1 Scrittura differita

È stata creata una funzione ad-hoc che si occupa della scrittura differita ed essa effettua le seguenti operazioni:

1. Si effettua la `mutex_lock`. In questo caso si è bloccanti perché una volta che i byte sono stati prenotati devo essere assolutamente scritti.
2. Si chiama la `write_dynamic_buffer`.
3. Si decrementa il numero di byte prenotati e si incrementa il numero di byte nel buffer.
4. Si rilascia il lock e si rilascia l'area di memoria della `work_struct`.
5. Si chiama `wake_up_interruptible` e `module_put`.

2.8 Operazione di lettura

La funzione `dev_read` consente la lettura del contenuto del buffer di kernel e spostare il contenuto letto all'interno del buffer di livello user. In questa funzione avvengono le seguenti operazioni:

1. Il primo controllo effettuato riguarda il numero di byte che l'utente vuole leggere, infatti se il numero di byte è pari a `0` si ritorna immediatamente `0` in modo tale da non consumare inutilmente cicli di CPU.
2. Viene allocata l'aria di memoria per un buffer temporaneo.
3. Anche qui ci sono due strade:
   - Nel caso bloccante, si segue lo stesso approccio della funzione `dev_write`, ma la condizione con cui si va a dormire riguarda ovviamente la presenza di byte da leggere.
   - Anche il caso non bloccante è simile all'approccio della funzione `dev_read`.
4. Dopo un controllo sulla dimensione dei byte da leggere, viene invocata la `read_dynamic_buffer`.
5. Viene decrementato il numero di byte nel buffer.
6. Si fa l'`unlock` e si effettua la `copy_to_user`.
7. Si ritorna il numero di byte consegnati a livello di user.

[4] Questo è un punto in favore in termini di performance perché all'interno della sezione critica si ha bisogno solamente di collegare il contenuto da scrivere alla struttura del deferred work.

2.9 Operazione `ioctl`

La funzione che risponde alle richieste di gestione di un multi-flow device file è la funzione `dev_open`. Sono stati definiti dei codici che permettono di identificare i comandi:

- il codice `TO_HIGH_PRIORITY` permette di settare il livello di priorità come alto;
- il codice `TO_LOW_PRIORITY` permette di settare il livello di priorità come basso;
- il codice `BLOCK` permette di settare le operazioni come bloccanti;
- il codice `UNBLOCK` permette di settare le operazioni come non bloccanti;
- il codice `TIMEOUT` permette di settare un timeout, infatti viene chiesto un argomento da passare alla funzione di `ioctl`.

Inoltre, è stata definita una piccola libreria che permette di rendere trasaparente l'invio di richieste I/O control, ovvero senza dover conoscere i codici. La libreria consiste in delle macro che espandono in invocazione di `ioctl` con il relativo codice ed eventualmente con l'apposito argomento.

2.10 Ulteriore osservazione

All'interno del codice è stata utilizzato il costrutto `unlikely` nei check degli errori riguardanti l'allocazione di memoria. Se ci fosse stato a disposizione un operation profile sarebbe stato interessante andare ad usare `likely/unlikely` per favorire la predizione dei salti, ad esempio se si fosse saputo che il 90% dei thread fosse `HIGH PRIORITY`, si sarebbe potuto mettere `likely` sul controllo `priority == HIGH PRIORITY` e `unlikely` sul duale.