Relazione del Progetto di Sistemi Operativi Avanzati

AA 2021-2022

Falcone Gian Marco 0300251

Sommario

Introduzione

Lo scopo del seguente documento è esporre e spiegare le scelte progettuali effettuate e i meccanismi adottati nello sviluppo di un modulo per il kernel Linux. Tale modulo deve rispettare la traccia riportata di seguito.

Multi-flow device file

La specifica è relativa a un driver di dispositivo 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 le operazioni 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 sul lavoro ritardato) delle operazioni di scrittura, pur mantenendo l'interfaccia in grado di notificare in modo sincrono il risultato. Le operazioni di lettura sono tutte eseguite 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.

Alcuni parametri e funzioni del modulo Linux dovrebbero essere implementati 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).

Struct *object_state* e stato del device file

Per mantenere e gestire le informazioni riguardo lo stato corrente dei diversi device file è stata usata una struttura dati *object_state*, riportata nell'immagine seguente. Ogni struttura viene associata ad un unico minor number e salvata all'interno di un array, di lunghezza pari al numero di minors che, come da specifiche, viene impostato a 128.

```
object_state objects[MINORS];
```

```
typedef struct _object_state{
#ifdef SINGLE_SESSION_OBJECT
    struct mutex object_busy;
#endif
    int minor;
    struct mutex operation_synchronizer[2];
    int valid_bytes[2];
    int offset[2];
    list_stream *stream_content[2];
    int reserved_bytes;
    wait_queue_head_t wait_queue[2];
} object_state;
```

Vengono qui di seguito esposti alcuni dei campi mantenuti nella struttura:

- *operation_synchronizer:* un array di due mutex, uno per ognuno dei due flussi. Essendo la struttura dati potenzialmente accessibile in parallelo da più thread operanti sullo stesso device file, i mutex sono necessari per garantire un corretto aggiornamento delle informazioni, nonché utilizzo del modulo. La scelta di usare due mutex distinti per i due flussi di dati permette a thread operanti a priorità diverse di poter comunque lavorare contemporaneamente.
- *valid_bytes:* array di due interi che indica il numero di byte disponibili in lettura in ogni flusso di dati. Viene aggiornato ogni qual volta il dispositivo compie un'operazione di lettura o di scrittura.
- *offset:* array di due interi che indica, per ogni flusso, il primo byte valido per l'operazione di lettura. Per ricavare il byte da qui iniziare un'operazione di scrittura si sommano tra loro il campo *offset* e il campo *valid_bytes*.
- *stream_content:* array composto da due strutture di tipo *list_stream*, una per ogni flusso. La discussione sul contenuto e la gestione viene rimandata al prossimo paragrafo.
- *wait_queue:* un campo il cui compito è mantenere la struttura dati necessaria per avere la possibilità di rendere bloccanti le operazioni, tramite il meccanismo delle wait queue. Anche in questo caso ulteriori dettagli verranno presentati nel seguito della trattazione.

Struct *list_stream* e gestione dei flussi di dati

Vengono mantenuti nel modulo, per ogni device file, due flussi dati indipendenti per le due priorità di riferimento (alta e bassa). Per consentire una gestione duttile e dinamica della memoria, in opposizione all'assegnazione di una quantità di byte finita e predeterminata per ogni flusso, si è scelto di gestire lo spazio disponibile secondo una lista di buffer doppiamente collegati. Inoltre, per evitare continue allocazioni e rilasci di memoria, seppure dovuti ad interazioni con lo slub allocator e quindi più rapidi ed efficienti, i buffer hanno dimensione fissa di una pagina (4096 byte), riservata tramite un'invocazione verso *__get_free_page*.

La nuova area di memoria viene allocata nel momento in cui lo spazio già disponibile non risulti più sufficiente per permettere un'operazione di scrittura. Tale area viene quindi collegata in coda alla lista. Una macro viene utilizzata per stabilire una dimensione massima alla lunghezza della lista

collegata, in modo da impedire uno scorretto utilizzo del modulo che porti alla saturazione dell'intera memoria disponibile sulla macchina.

Un'operazione di lettura che renda un buffer precedentemente allocato non più necessario in quanto il suo contenuto è già stato letto completamente, comporta un rilascio della memoria occupata da quest'ultimo, nonché dell'intero nodo presente nella lista.

Gestione della sessione e struct *session_state*

Per mantenere le informazioni riguardo una sessione attiva verso il device file, si utilizza il campo *private_data* presente all'interno della struct *file.* Tale struttura viene creata dal kernel all'apertura del file e viene passato a qualsiasi funzione operante sul file stesso, fino alla sua chiusura.

In questo modo si ha una struct *file* unica per ogni sessione attiva che può essere sfruttata per mantenere informazioni riguardo la sessione stessa, così da salvare le informazioni di stato tra le diverse funzioni. Nell'operazione di apertura (*dev_open*) viene infatti creata e inizializzata una struttura di tipo *session_state*, alla quale si fa puntare il campo *private_data*. Si fa attenzione a liberare la memoria nell'operazione di chiusura (*dev_release),* prima che la struct *file* sia distrutta dal kernel.

```
typedef struct _session_state{
    enum priority priority;
    bool blocking;
    unsigned long timeout;
} session_state;
```

Di seguito una spiegazione dei campi presenti in tale struttura dati:

- *priority:* indica la priorità del flusso dati su cui effettuare le operazioni (alto o basso);
- *blocking:* valore booleano per indicare se le operazioni devono essere effettuate in maniera bloccante (*true)* o non bloccante (*false);*
- *timeout:* in caso di operazioni bloccanti indica il timeout massimo da aspettare entro cui le condizioni devono essere soddisfatte. Se al termine del timeout non è ancora possibile effettuare l'operazione richiesta, essa fallisce.

Operazione di *ioctl()*

È possibile configurare i parametri sopracitati tramite la funzione *dev_ioctl().* Si possono in questo modo cambiare le modalità con cui le operazioni vengono effettuate, nonché il flusso dati da utilizzare in base alla priorità assegnata. Poiché la funzione modifica solo i campi della struct *session_state* e poiché tale struct è unica per ogni sessione aperta, non è necessario inserire sezioni critiche, e quindi mutex, al suo interno.

È inoltre possibile, tramite la *dev_ioctl()*, abilitare o disabilitare il device file in questione. Se disabilitato, qualsiasi tentativo di aprire una nuova sessione fallisce, ma le sessioni già stabilite possono comunque funzionare correttamente.

Le operazioni sul device driver

Operazione di scrittura

La funzione *dev_write* permette la scrittura del contenuto di un buffer a livello user all'interno del buffer a livello kernel gestito dal modulo stesso. Ogni qual volta un thread voglia effettuare un'operazione di scrittura nel flusso di un certo device (associato ad un minor number univoco), è necessario che esso prenda un lock esclusivo che blocchi sia eventuali altri accessi sui dati stessi che sui metadati di gestione per il dispositivo in questione (lock limitato al flusso di priorità considerato).

Prima di essere copiati nel buffer di destinazione, i dati vengono copiati a livello kernel attraverso un buffer temporaneo, tramite la *copy_from_user*. Tale invocazione avviene precedentemente al recupero del lock, in modo che la possibile gestione di eventuali page fault avvenga esternamente alla sezione critica. Ciò avviene quando il buffer di livello user, passato come sorgente dei dati, sia mappato in memoria ma non ancora effettivamente materializzato. Si rende in tal modo possibile ad altri thread di continuare ad operare normalmente anche in presenza di tale avvenimento, con vantaggi a livello prestazionali.

L'operazione di scrittura può avvenire su uno dei due flussi dati (alta e bassa priorità) ed essere bloccante o meno, a seconda della configurazione attuale della sessione (struct *session_state*). In tutti i casi l'operazione torna il numero di byte effettivamente scritti nel buffer o, eventualmente, un errore.

Nel caso di operazione non bloccante, l'impossibilità di recuperare il lock (*mutex_trylock()*) o la mancanza di byte disponibili hanno come risultato il fallimento immediato dell'operazione. Nel caso di operazione bloccante invece si attende, fino al termine di un timeout impostabile, che le condizioni per la scrittura vengano soddisfatte.

Da notare che l'operazione ha successo solo se si ha a disposizione abbastanza spazio da scrivere tutti i dati richiesti: per scelta implementativa non è possibile scrivere sul buffer solo una parte delle informazioni, in quanto ciò potrebbe portare al successivo recupero di un'informazione incompleta ed eventualmente scorretta.

Operazione di lettura

L'operazione di lettura è implementata dalla funzione *dev*_read. Al fine di consentire un corretto aggiornamento dei dati e dei metadati di gestione, anche in questa situazione è necessario prendere un lock sulla struct *object_state* per il flusso dati corrispondente alla priorità impostata, onde evitare situazioni non volute sullo stato del device o sui dati in esso contenuti.

Similmente all'operazione di scrittura, anche nelle letture si fa utilizzo di un buffer temporaneo in cui copiare i dati recuperati dal buffer a livello kernel prima di passarli a livello user con la funzione *copy_to_user().* Le motivazioni di questo passaggio sono le stesse descritte nel paragrafo precedente. Al fine di garantire che tutti i byte copiati nel buffer temporaneo vengano effettivamente passati al buffer utente (in modo da non eliminare dal flusso byte poi non consegnati) si fa uso, come controllo preliminare, della funzione *clear_user()*. Essa ritorna il numero di byte che non è possibile azzerare che, in caso di buffer utente conforme, dovrebbe risultare pari a 0.

Le operazioni di lettura possono essere bloccanti o meno, sempre a seconda della configurazione dei parametri per la sessione attuale. In questo caso non è più richiesto che venga letto tutto l'ammontare di byte richiesto: se nel buffer sono presenti un quantitativo i byte inferiore a quello richiesto, si portano a livello user solo i byte disponibili. Solo nel momento in cui il buffer è completamente vuoto si restituisce un messaggio di errore (caso non bloccante) o si porta il thread a dormire in attesa dei dati (caso bloccante).

I dati effettivamente letti vengono infine eliminati logicamente dal flusso di riferimento, spostando in avanti l'offset dell'ammontare di byte necessario.

Scritture asincrone e *work queue*

Le scritture effettuate sul flusso a bassa priorità vengono eseguite in maniera asincrona tramite il meccanismo delle work queue e quindi del lavoro deferred.

La scelta riguardo l'uso delle work queue rispetto ad altri meccanismi simili è dovuta alla presenza nel lavoro schedulato di una *mutex_lock,* ossia un'operazione bloccante. Nonostante ne sia comunque sconsigliato l'uso, essa si rende necessaria per impedire che la scrittura fallisca trovando il mutex già occupato (la *mutex_trylock* avrebbe comportato la possibilità di fallimento nel momento in cui non fosse stato possibile ottenere il lock).

È infatti necessario notificare in maniera sincrona il risultato dell'operazione e, per tale motivo, tutti i controlli sulla dimensione dei byte da scrivere devono essere effettuati precedentemente all'inserimento del lavoro nella coda. Il campo *reserved_bytes* presente nella struct *object_state* ha il compito di tenere memoria della quantità di byte ancora non effettivamente scritti ma di cui si è già preso carico. In tal modo è garantito che il deferred work scriva la esatta quantità di byte richiesta.

La scrittura effettiva è molto simile alla funzione *dev_write* ma senza che si effettuino controlli sulla priorità attuale, poiché si è certi che essa debba avvenire sempre sul flusso a bassa priorità.

Il passaggio dei metadati di riferimento nonché dei byte da scrivere avviene mediante l'uso della macro *container_of*.

Per essere sicuri che il lavoro differito non venga eseguito successivamente alla rimozione del modulo, si incrementa lo usage count del modulo stesso tramite la funzione *try_module_get()*. Si permetterà la rimozione del modulo solo al completamento dell'operazione.

Operazioni bloccanti e *wait queue*

Entrambe le operazioni di scrittura e di lettura hanno una loro versione bloccante. In questa situazione il thread che non può eseguire subito il lavoro richiesto rimane in attesa per un certo timeout impostato dall'utente o finché non venga risvegliato da un altro thread che, eventualmente, ha modificato la sua condizione di risveglio.

Nel caso di scrittura, l'attesa può avvenire su tre diverse condizioni:

- il mutex per il flusso di dati sul quale si vuole scrivere risulta già preso da un altro thread;
- si vuole scrivere sul buffer a priorità alta ma il numero di byte disponibili non è sufficiente a contenere l'informazione passata lato user;
- si vuole scrivere sul buffer a priorità bassa ma il numero di byte disponibili, sottraendo anche i byte riservati per scritture asincrone già programmate, non è sufficiente a contenere l'informazione passata lato user;

Nel caso di lettura, l'attesa può dipendere da:

- il mutex per il flusso di dati dal quale si vuole leggere risulta già preso da un altro thread;
- si vogliono leggere dati da un buffer a qualsiasi priorità il cui numero di byte validi in lettura è uguale a 0 (nel caso in cui sia presente almeno un byte l'operazione viene comunque eseguita correttamente).

Tutti i thread, in caso di sleep, vengono inseriti all'interno dell'apposita wait queue del device file (minor number) in base al flusso di priorità al quale appartengono. Le teste di queste code sono mantenute nel campo *wait_queue* della struct *object_state* e vengono inizializzate tramite la *init_waitqueue_head,* presente nella *init_module*. La scelta di distinguere le code in base alla priorità del flusso dati è necessaria onde evitare che un thread ad una certa priorità, impossibilitato a completare l'operazione richiesta, blocchi altrimenti tutti i successivi thread in coda ma in attesa sull'altro flusso dati che potrebbero avere già le condizioni di risveglio soddisfatte.

L'inserimento in coda avviene, per versioni del kernel a partire dalla 3.13, tramite la funzione *my_wait_event_timeout*, definita, sulla base della *wait_event_timeout,* come mostrato nella seguente figura.

```
//defined in order to put task in state exclusive
#define __my_wait_event_timeout(wq_head, condition, timeout)          \
    ___wait_event(wq_head, ___wait_cond_timeout(condition),          \
                  TASK_UNINTERRUPTIBLE, 1, timeout,                  \
                  __ret = schedule_timeout(__ret))


#define my_wait_event_timeout(wq_head, condition, timeout)            \
({                                  \
    long __ret = timeout;                          \
    might_sleep();                          \
    if (!___wait_cond_timeout(condition))                  \
        __ret = __my_wait_event_timeout(wq_head, condition, timeout);  \
    __ret;                                  \
})
```

La differenza principale rispetto alla macro offerta dal kernel è legata al fatto che i thread vengono messi in attesa (*TASK_UNINTERRUPTIBLE*) con il flag *WQ_FLAG_EXCLUSIVE* impostato (bit a 1 nella funzione *__wait_event)*.

Il motivo di tale scelta è dovuto principalmente a due condizioni:

1. la funzione *wake_up*, necessaria per il risveglio dei thread in sleep e posta al termine di ogni operazione di lettura o scrittura (sincrona o asincorna), ha il comportamento di svegliare tutti i thread in stato *non-exclusive* ed un unico thread in stato *exclusive* (il primo che si incontra nella lista). Poiché, tramite la *wait_event_timeout,* i thread possono essere messi in coda solo in stato non *exclusive*, la *wake_up* avrebbe comportato il risveglio, inutile, di tutti i thread posti in attesa. Ad un unico thread sarebbe, eventualmente, permesso prendere il lock sul flusso dati di riferimento.
   Secondo la macro definita invece, i thread vengono messi in sleep con il flag WQ_FLAG_EXCLUSIVE impostato. In questo modo la *wake_up* può, ad ogni chiamata, risvegliare uno solo dei thread in attesa, evitando i cicli di clock necessari per il controllo della condizione di uscita da parte di tutti gli altri.
2. Era inoltre richiesto che la consegna dei dati seguisse una politica FIFO. La scelta di mettere i thread in sleep con lo stato *exclusive* ha come conseguenza un inserimento in coda alla lista, garantendo una politica FIFO anche riguardo la schedulazione di tali thread.
   Al contrario, un inserimento in stato *non-exclusive* avrebbe comportato un inserimento in testa, con la conseguenza che i thread entrati in stato di attesa per ultimi vengano anche eseguiti prima.

Tale meccanismo non è supportato per versioni del kernel più vecchie delle 3.13. Di conseguenza, in tale caso, vengono risvegliati tutti i thread in coda ma al più uno solo di questi potrà portare a termine correttamente l'esecuzione.

Parametri del modulo

All'interno del modulo sono state definiti anche alcune variabili globali con il ruolo di parametri. Essi corrispondo a file all'interno dello pseudo-file system */sys* e sono quindi accessibili e modificabili anche esternamente al modulo stesso, nel rispetto dei permessi assegnati. Sono state in particolare definiti 5 parametri di tipo array, tutti di lunghezza pari al numero massimo di *minor* gestibile dal modulo, tramite la macro *module_param_array():*

- *enabled:* array di tipo boolenano che indica, per ogni *minor number*, se il rispettivo device file è abilitato o meno a ricevere nuove sessioni. Si è deciso di creare tale parametro lasciando la possibilità ad un utente o ad un gruppo autorizzato sia di leggere il valore contenuto che, eventualmente, di modificarlo (*S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP*).
- *high_priority_valid_bytes:* array di *long* contenente il numero di byte validi, e quindi leggibile, per il flusso ad alta priorità, riferito ad ogni device file. È possibile leggere il contenuto di tale variabile direttamente dal file system */sys* ma non ne è consentita la modifica, in quanto legata ai dati effettivamente contenuti nel buffer (*S_IRUSR | S_IRGRP*).
- *low_priority_valid_bytes:* similmente al caso precedente ma per il flusso a bassa priorità.
- *high_priority_waiting_threads:* array di *long* che indica il numero di thread in attesa del mutex, di dati da leggere o di byte disponibili in scrittura per il flusso ad alta priorità. Da notare quindi che tale variabile tiene conto dei thread in sleep qualunque sia il motivo della loro attesa, nonostante sia molto improbabile che al suo interno siano presenti, nello stesso momento, thread bloccati in operazioni diverse. Anche in questo caso è possibile legge-

re il contenuto di tale variabile direttamente dal file system */sys* ma non ne è consentita la modifica, in quanto legata allo stato del device file.
- *low_priority_ waiting_threads:* similmente al caso precedente ma per il flusso a bassa priorità.

Inizializzazione e rilascio del modulo

Nella funzione *init_module* si inizializza lo stato interno del device driver, tra cui le strutture dati che esso utilizza. In particolare:

- si riserva lo spazio necessario per la testa della lista collegata per ogni flusso di dati, preallocando, per ognuno, una pagina di memoria;
- si impostano i valori di alcuni dei metadati del driver;
- si inizializzano i mutex necessari per la sincronizzazione;
- si inizializzano le *wait queue* per le operazioni bloccanti.

Si fa quindi affidamento sulla funzione *__register_chrdev()* in modo da creare e registrare un driver di dispositivo a caratteri, il cui *major number* viene dinamicamente assegnato dal sistema operativo.

Nella funzione *cleanup_module* si libera la memoria precedentemente allocata nell'inizializzazione del modulo e, tramite la funzione *unregister_chrdev()* si elimina il driver. Da notare che si procede anche ad eliminare gli eventuali ulteriori elementi presenti nelle liste collegate allocati dinamicamente nell'utilizzo del modulo.