Relazione Progetto Sistemi Operativi Avanzati Anno 2021/22

Marco Calavaro 0295233

Tabella dei contenuti

1 Traccia del progetto

Multi-flow device file

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)

2 Organizzazione del codice

Il codice è stato organizzato in 3 blocchi logici, per poterne semplificare la lettura, ognuno di essi è formato da un file `.c` e da un header file `.h` e sono rispettivamente:

- Driver-core: blocco che sviluppa le funzionalità principali del driver tra cui le operazioni che esso deve eseguire e le istruzioni da eseguire per l'istallazione del modulo.
- Deferred-work: blocco che sviluppa il lavoro da eseguire in modo differito necessario per la scrittura sul dispositivo con priorità bassa.
- klist: blocco che sviluppa il codice del flusso di dati su cui poi viene costruito il device file.

3 Descrizione del divice file

Il driver, seguendo le specifiche di progetto, deve poter gestire un numero prefissato di device (128), la loro rappresentazione è realizzata dalla `struct device_state` che ne mantiene i metadati.

```
typedef struct _device_state{
    int id;
    wait_queue_head_t waitq[PRIORITY_NUM];
    struct workqueue_struct* workq; //only for low priority
    klist* data_flow[PRIORITY_NUM];
} device_state;
```

Codice della struct device_state

Descrizione della struttura:

- `id`: variabile che mantiene il minor number del device
- `waitq`: utilizzata per mettere in attesa le operazioni bloccanti
- `workq`: utilizzata per la priorità bassa per schedulare lavoro differito
- `data_flow`: mantiene l'indirizzo dei due flussi di dati, che sono realizzari dalla struct klist

3.1 Klist

Per poter realizzare i due flussi di dati, si è scelto di sviluppare una struttura dati di tipo *lista collegata*. Essa non distingue tra le priorità, ma sarà poi compito delle operazioni del driver garantire il corretto accesso e funzionamento del divice.

La struttura dati `struct klist` mantiene i metadati di una lista:

```
typedef struct _klist
{
    struct mutex op_mtx;
    klist_elem* head;
    klist_elem* tail;
    unsigned long len;
    unsigned long reserved;

}klist;
```

Descrizione della struttura:

- `op_mtx`: mutex utilizzato per sincronizzare le operazioni sulla struttura di dati condivisa
- `head`: puntatore all'elemento in testa alla lista
- `tail`: puntatore all'elemento di coda della lista
- `len`: variabile che mantiene la lunghezza attuale della lista
- `reserved`: variabile utilizzata per limitare la lunghezza della lista

Ciascun elemento della lista viene rappresentato dalla `struct klist_elem`:

```
typedef struct _klist_elem{
    char* buffer;
    int last_read;
    int size;
    struct _klist_elem* next;
}klist_elem;
```

Descrizione della struttura:

- `buffer`: puntatore alla zona di memoria che mantiene i dati dell'elemento
- `last_read`: indice dell'ultimo elemento che è stato letto, inizialmente 0
- `size`: grandezza del buffer
- `next`: prossimo elemento della lista

Operazioni della klist

Di seguito sarenno brevemente descritte le operazioni che vengono invocate per la gestione della lista.

Operazioni di allocazione:

```
klist* klist_alloc(void)
klist_elem* klist_elem_alloc(char*,int,gfp_t)
```

Tali operazioni vengono invocate per allocare rispettivamente la `klist` e un nuovo elemento di essa.

Operazioni di free:

```
void klist_free(klist*)
void klist_elem_free(klist_elem*)
```

Operazioni duali a quelle di allocazione, vengono utilizzate per liberare lo spazio della relativa `klist`, elemento di essa. Nello specifico la `klist_free` sfrutta internamente la funzione `remove_head` che libera l'elemento in testa alla coda.

Operazioni per riservare spazio:

```
bool reserve_space(klist*,unsigned long)
void free_reserved_space(klist*,unsigned long)
```

Queste sono le operazioni per poter gestire dello spazio riservato della lista, esse permettono quindi di avere un controllo sulle dimensioni massime che ciascuna lista può raggiungere.

Operazioni per I/O:

```
int klist_put(klist*,char*,unsigned int,gfp_t)
int klist_get(klist*,char*,unsigned int)
```

Le seguenti operazioni permettono di inserire caratteri o leggerli dalla lista, internamente esse gestiscono gli `klist_element`, nello specifico la put genera un nuovo elemento ogni qual volta viene invocata inserendo tale elemento in coda, mentre la get parte dall'elemento di testa e lo rimuove se viene letto ciascun carattere del suo buffer.

4 Descrizione del comportamento del driver

Nel seguente capitolo saranno descritte le operazioni che il driver deve esporre per la gestione del device:

- Lettura e scrittura dal dispositivo
- Controllo delle impostazioni di I/O
- Apertura e rilascio del dispositivo

A ciascun dispositivo viene associata una `struct session_state` che rappresenta le informazioni di gestione della sessione di I/O, essa viene iniziallizata nella funzione `dev_open` secondo dei parametri di default. La sessione viene gestita tramite le operazioni di `dev_ioctl`.

La struct in questione:

```
typedef struct _session_state{
    int priority;
```

```
    bool blocking;
    unsigned long timeout;
} session_state;
```

Descrizione della struttura:

- `priority`: indica la priorità corrente della sessione, assume i volori `HIGH_PR = 0`, `LOW_PR=1`.
- `blocking`: indica se le operazioni devo essere effettuate in modo bloccante.
- `timeout`: indica il tempo di attesa massimo per le operazioni bloccanti.

4.1 Operazioni di lettura e scrittura

Le operazioni di lettura e scrittura ricevono nel campo `private_data` del `file` la struct `session_state` per poter recuperare le informazioni necessarie al loro comportamento che differisce sia in basa alla priorità, sia in base al caso bloccante. Una volta distinto i vari casi basterà invocare la put o get dalla relativa `klist`.

Tali operazioni sfruttano le funzioni `copy_from_user` e `copy_to_user` per poter scambiare i dati tra lo spazio kernel e l'utente.

Operazione di scrittura:

Tale operazione non può essere bloccante, ma potrebbe non andare a scrivere nuovi dati nella relativa *klist* se non è possibile riservare spazio nella lista. Il check viene effettuato tramite l'invocazione della funzione `reserve_space`, la logica del driver prevede che deve essere possibile poter scrivere il nuovo blocco di dati per intero e non un parziale di esso.

Una volta riservato lo spazio bisogna distinguere tra le priorità:

- Caso HIGH priority: si esegue direttamente la put dei nuovi dati nella lista ad alta priorità del device.
- Caso LOW priority: Si prepara il lavoro che verrà eseguito in modo differito.

Operazione di lettura:

Tale operazione può essere bloccante o meno, nel caso la sessione sia bloccante bisognerà mettere in attesa il thread fino a quando nella relativa coda non sono presenti dati o scade il timeout.

Per quanto riguarda le priorità la scrittura non differisce nel suo coportamento, ma solo la lista che va a leggere, tramite la funzione *get* della lista. Una volta recuperati i dati viene liberato lo spazio riservato tramite la funzione `free_reserved_space`.

Scrittura deferred

Il flusso a bassa priorità gestisce le scritture in modo deferred, mantenendo la risposta del driver sincrona. Per questo prima di schedulare il lavoro deferred, viene riservato dello spazio nella relativa lista.

Il lavoro deferred viene gestito tramite `work queue`, esso viene impacchettato nella `struct packed_work`, la funzione che viene invocata è `actual_work` la quale compie un lavoro simile alla scrittura sul flusso di alta priorità.

```
typedef struct _packed_work{
        char* buffer;
        int len;
        device_state* device;
        struct work_struct the_work;
} packed_work;
```

Gestione delle operazioni bloccanti

Per poter gestire il lavoro bloccante in caso di mancanza di dati da leggere il thread viene messo in sleap su una `wait queue`.

Quando saranno disponibili dei dati, tramite l'API `wake_up_interruptible` sarà risvegliato uno dei thread in attesa.

4.2 IO Control

Viene sfruttata la system call `ioctl` per poter gestire la sesione del dispositivo, i possibili comandi sono definiti dalle seguenti macro:

```
#define SET_HIGH_PR _IO(MAGIC_NUMBER,0x00)
#define SET_LOW_PR _IO(MAGIC_NUMBER,0x01)
#define SET_OP_BLOCK _IO(MAGIC_NUMBER,0x02)
#define SET_OP_NONBLOCK _IO(MAGIC_NUMBER,0x03)
#define SET_TIMEOUT_BLOCK _IOW(MAGIC_NUMBER,0x04,int32_t*)
```

In base al comando che viene utilizzato è quindi possibile cambiare:

- La priorità della sessione
- Cambiare le operazioni in bloccanti o non bloccanti
- Cambiare il valore del timeout per operazioni bloccanti

4.3 Apertura e rilascio del device

Per poter aprire e rilasciare il dispositivo si utilizzano rispettivamente le funzioni `dev_open` e `dev_release`, la open prepara la `struct session_state` inizializzandola con alcuni valori di default:

```
#define DEFAULT_PR HIGH_PR
#define MAX_TIMEOUT ULONG_MAX
```

4.4 Operazioni di init e clenup del modulo

Viene utilazzata la funzione `init_module` per inizializzare le strutture dati per il funzionamento del driver:

- Viene inizializzato un arrei che mantiene per ciascun device lo stato ABILITATO, DISABILITATO.
- Vengono inizializzate le `struct device_state` .

> - Viene assegnato il ***major number*** del driver.

La funzione `cleanup_module` si occupa invece di effettuare la free delle varie *struct* del driver.

5 Parametri kernel

Il progetto richiedeva che il modulo esportasse 5 parametri nel VFS nella directory relativa al modulo `/sys/modules/project-module/parameters` definiti tramite la funzione `module_param_array`. Di seguito viene descritto il loro funzionamento:

- `devices_state`: gestisce lo stato del device che può essere `DISABLED = 0` oppure `ENABLE = 1`, questo file ha permessi di lettura e scrittura.
- `waiting_threads_high[low]`: ogni entry del file indica il numero di thread che sono stati messi in attesa su ciascun flusso.
- `high[low]_prio_data`: ogni entry rappresenta il numero di byte del corrispondente device sul flusso di priorità high[low]

Questi ultimi parametri hanno permessi di sola lettura. (0440)

6 Test

Per poter testare i dispositivi è stato creato del codice client che permette di creare e interagire con i dispositivi.

Nello specifico vi sono:

> - Due script per la creazione e rimozione di dispositivi, rispettivamente `add_dev.sh` e `rm_dev.sh`.
> - Codice c per interaggire con il dispositivo.

6.1 Client code

Per poter compilare il codice del client basterà utilizzare il comando `make all` che creerà il file `client`, tale script deve essere lanciato con i permessi di root e richiede come parametro il percorso del dispositivo.

All'apertura viene mostrato un menù con 8 opzioni che permettono:

> - Scrivere e leggere dal dispositivo.
> - Cambiare la priorità del dispositivo.
> - Cambiare la tipologia delle operazioni (Bloccante o non).
> - Impostare un nuovo valore del timer