Relazione del progetto di Sistemi Operativi Avanzati AA 2021-2022

Caliandro Pierciro (matr 0299815)

26 marzo 2022

1 Introduzione

Il seguente documento riassume le scelte progettuali e la documentazione relative allo sviluppo del modulo kernel **multistream-driver.c**.
Di seguito, viene riportata la specifica di progetto dove sono indicate anche le funzionalità che il software deve fornire:

> Implementare un device driver che offra flussi di dati ad alta e bassa priorità. Tramite una sessione aperta ad un device file un thread può leggere e scrivere segmenti di dati. La consegna dei dati segue una politica di tipo "First-in-First-Out" in ciascuno dei due diversi flussi di dati.
> Dopo un'operazione di lettura, i dati letti vengono rimossi dal flusso. Inoltre, il flusso ad alta priorità deve offrire delle operazioni di scrittura sincrone mentre il flusso a bassa priorità deve offrire una esecuzione asincrona (basata su lavoro deferred) delle operazioni di scrittura, ma riuscendo comunque a mantenere l'interfaccia in grado di notificare il risultato in maniera sincrona.
> Le operazioni di lettura invece sono sempre eseguite in maniera sincrona.
> Il device offre il supporto a 128 device file corrispondenti allo stesso ammontare di minor numbers.
> Il driver deve inoltre implementare il supporto al servizio `ioctl(...)` per poter gestire la sessione di I/O come segue:
>
> - configurazione del livello di priorità (altro o basso) per le operazioni
> - operazioni di lettura e scrittura bloccanti o non bloccanti
> - configurazione di un timeout che regoli il risveglio delle operazioni bloccanti
>
> Devono inoltre essere implementati dei parametri del modulo e delle funzioni per poter abilitare o disabilitare il device file, in termini dello specifico minor number.
> Se quest'ultimo è disabilitato, ogni tentativo di apertura di una nuova sessione deve fallire (ma sessioni già aperte dovranno ancora poter essere gestite).
> Ulteriori parametri addizionali esposti tramite il VFS devono fornire una fotografia dello stato corrente del device file in merito alle seguenti informazioni:
>
> - abilitato o disabilitato
> - numero di bytes correntemente presenti nei due flussi di priorità
> - numero di thread correntemente in attesa per i dati lungo i due flussi

2 Gestione dei device file

Per la gestione dei device file, è stata utilizzata una struttura di dati per mantenere i dati ed i metadati relativamente ad ognuno dei possibili minor numbers supportati.
La struttura è la seguente:

```
1     typedef struct _object_state{
2     #ifdef SINGLE_SESSION_OBJECT
3       struct mutex object_busy;
4     #endif
5       struct mutex operation_synchronizer[2];
6       int valid_bytes[2];
```

```
7       int total_free_bytes[2];
8       object_content* list_heads[2];
9       wait_queue_head_t the_wq_head;
10    } object_state;
11
```

dove:

- `operation_synchronizer` è una array di due mutex, usati per garantire che non vi sia più di un thread alla volta che modifica delle informazioni per la struttura di dati condivisa. Viene quindi usato un mutex distinto per ognuno dei due flussi di priorità che in questo modo possono operare in parallelo;
- `valid_bytes` è un array che mantiene i byte correntemente validi per entrambe i flussi di priorità. Difatti, all'atto della lettura i byte vengono cancellati logicamente mediante l'uso dell'indice `read_offset` della struttura `object_content`
- `total_free_bytes` è un array che mantiene invece il numero di bytes effettivamente disponibili nel flusso. Questo perché è possibile che vengano "logicamente" riservati dei byte che saranno poi usati per le scritture deferred, come verrà esplicato in seguito.

Vi è poi il campo `list_heads` che è usato per mantenere le due teste delle liste collegate utilizzate per rappresentare il flusso di byte.
Difatti, ciascun flusso può contenere un numero di byte pari alla dimensione di una pagina di memoria, ovvero 4096 bytes, moltiplicata per il numero di pagine massime utilizzabili che viene limitato dalla define `MAX_PAGES`.
Vi è poi la struttura di dati utilizzata per rappresentare il singolo elemento della lista collegata dei dati del device, riportata in seguito

```
1   typedef struct _object_content{
2     int record_length;
3     int read_offset;
4     char *stream_content;
5     struct _object_content *next;
6   } object_content;
7
```

qui, vi sono i seguenti metadati:

- `record_length` che indica la taglia attuale del contenuto della pagina di memoria;
- `read_offset` che indica l'offset di lettura all'interno del record. Quindi, tutti i byte precedenti a questo offset di lettura non sono più leggibili logicamente;
- `next` puntatore all'elemento successivo della lista

vi è infine il campo `stream_content`, che mantiene l'effettivo contenuto del file.
Come detto in precedenza, per l'allocazione di tale campo si considera la dimensione di una pagina, usando quindi l'API kernel `__get_free_page` per ottenere una pagina di memoria dall'allocatore.
Sono state effettuate diverse scelte progettuali in merito a tale gestione del file, in particolare:

1) Utilizzare pagine invece di allocare ogni volta un record di taglia pari al numero di byte che l'utente ha richiesto di scrivere.
   In questo modo, si cerca di minimizzare l'overhead legato alla memorizzazione del record all'interno della lista: infatti, ogni nodo mantiene, oltre ai dati anche 16 byte di metadati,

che nel momento in cui l'utilizzo del driver prevede che vengano effettuate scritture di pochi byte ogni volta, diventano significativi rispetto alla dimensione dei dati.
Utilizzando invece memoria con la granularità di una pagina, l'overhead è minimale rispetto alla dimensione dei dati effettivamente utili;

2) Allocare direttamente una pagina intera ogni volta che viene richiesta una operazione di scrittura per cui non basta lo spazio nell'ultima pagina allocata.
In questo modo, anche se lo spazio non viene totalmente saturato, successive scritture avranno già la pagina allocata e non saranno necessarie altre interazioni con l'allocatore di memoria nel breve periodo;

3) La cancellazione dei byte avviene in primo luogo in maniera logica, spostando l'indice `read_offset` in avanti di tanti byte quanti ne vengono letti. L'eliminazione effettiva della pagina viene effettuata quando l'indice di lettura arriva alla dimensione massima della pagina (4KB), in questo modo si paga di meno il costo di eventuali eliminazioni dei nodi dalla lista, che avrebbero un costo algoritmico O(n).

3 Operazioni di lettura e scrittura

Le operazioni di lettura e scrittura sono rispettivamente la `dev_read` e `dev_write`.
Entrambe hanno un comportamento iniziale molto simile: difatti, per ciascuna vengono effettuati diversi controlli basati sulle informazioni della sessione di I/O, rappresentate dalla struttura di dati `io_sess_info`, riassunti nei seguenti passi ad alto livello:

- si tenta di acquisire un lock mediante l'API `mutex_try_lock`.
  Nel caso in cui il lock sia occupato, si verifica se l'operazione è bloccante ed in caso contrario si esce
- viene poi verificato rispettivamente:
  - per le operazioni di lettura, se vi sono dati da leggere ovvero `valid_bytes` per il flusso di priorità corrente per quella sessione > 0
  - per le scritture, invece, se è possibile scrivere bytes, quindi `total_free_bytes` per il flusso di priorità corrente per quella sessione $\neq 0$.
- nel caso in cui non fosse rispetta la condizione sui dati, si restituisce un valore di errore in quanto non è possibile completare l'operazione
- infine, si esegue l'operazioni di lettura o scrittura, andando a limitare il numero di byte che viene richiesto di leggere o scrivere al numero massimo disponibile, come avviene nelle classiche system call di `read` e `write`.

Sia nella funzione di lettura che di scrittura, la copia dei dati da o verso l'utente avviene avvalendosi in prima istanza di un buffer temporaneo.
Questo poiché le API usate per finalizzare la copia di tali dati sono `copy_to_user` e `copy_from_user`, che nel caso in cui, ad esempio, una pagina di memoria user space fosse mmappata ma non materializzata porterebbero il thread a dormire in quanto andrebbe risolto un page fault.
Siccome tale copia deve avvenire sulla struttura di dati condivisa, avendo preso il lock si bloccherebbe l'interno flusso fino alla risoluzione del fault, quindi la strategia prevede:

- per le scritture, di effettuare la `copy_from_user` all'interno di un buffer temporaneo.Successivamente, acquisire il lock e copiare i dati nel flusso condiviso mediante `memcpy`

- per le letture, acquisire il lock e copiare i dati dal flusso condiviso in un buffer temporaneo mediante `memcpy`. Inoltre, ogni volta che viene letto l'intero contenuto di una pagina, si incrementa una variabile locale che indica il numero di pagine che andranno deallocate e, di conseguenza, il numero di nodi rimossi, operazione che avverrà alla fine della funzione di lettura.
  Infine, viene rilasciato il lock e si effettua la copia nel buffer utente mediante la `copy_to_user`

In entrambe i casi, sia della `read` che della `write`, il valore di ritorno è il numero di byte che sono stati letti o scritti in caso di successo oppure un valore di errore.

3.1 Operazioni bloccanti

Nel caso di operazioni bloccanti di lettura o scrittura, il thread può andare in sleep su una apposita wait event queue, la cui testa della coda è definita nel campo `the_wq_head` della struttura `object_state`, che è un array contenete le teste delle due wait queue, una per le operazioni a bassa priorità ed una per quelle ad alta priorità.
La funzione che si occupa del porre il thread in sleep è la `do_sleep_wqe`: tale funzione prende fra i parametri il valore su cui verrà fatto il controllo della condizione ed il motivo per cui si va in wait: difatti, un thread può richiedere di essere posto nella wait queue per 3 motivi:

1. il mutex non è disponibile;
2. non vi sono dati da leggere nel device file;
3. non c'è più spazio per scrivere byte nel device file

Il risveglio sarà di un singolo thread che verificherà la condizione e non di tutti, in quanto se anche venissero svegliati tutti, alla fine uno solo riuscirebbe ad acquisire il lock con successo.
Per il risveglio, è possibile utilizzare l'API `wake_up_interruptible`: tale API risveglia un singolo thread che si trova in wait in stato esclusivo, più eventualmente tutti i thread presenti all'interno della coda che son invece in stato di wait non esclusivo.
Per porsi nella wait queue, vi è l'API `wait_event_interruptible_timeout` che permette di uscire o dopo che la condizione è stata verificata o dopo lo scadere del timeout, che viene passato come parametro della funzione `do_sleep_wqe`.
Tale API per default porta il thread a dormire in stato non esclusivo, quindi la scelta è stata quella di ridefinire tale macro, così come anche la macro chiamata al sua interno come segue:

```
1    #define __wait_event_interruptible_timeout_exclusive(wq_head,
     condition, timeout)       \
2          ___wait_event(wq_head, ___wait_cond_timeout(condition),
                    \
3                   TASK_INTERRUPTIBLE, 1, timeout,
             \
4                   __ret = schedule_timeout(__ret))
5
6
7    #define wait_event_interruptible_timeout_exclusive(wq_head,
     condition, timeout)       \
8    ({                                                          \
9           long __ret = timeout;
        \
10          might_sleep();
     \
11          if (!___wait_cond_timeout(condition))                \
```

```
12                __ret = __wait_event_interruptible_timeout_exclusive(
    wq_head,         \
13                                  condition, timeout);
        \
14              __ret;
    \
15        })
16
17
```

La macro ridefinita è
`___wait_event`, a cui viene passato come parametro valore del 4° parametro 1, invece dello 0 che portava a dormire in stato non exclusive.
Siccome le operazioni di `read` e `write` permettono la scrittura e lettura anche di una quantitativo di bytes minore di quelli realmente richiesti, l'unico evento che può portare un thread a dormire è la non disponibilità del driver, ovvero il caso in cui il mutex che regola le operazioni sulla struttura di dati condivisa non possa essere preso.
Inoltre, siccome la wait è in modalità interrompibile, quando il thread esce verifica anche l'eventualità che il risveglio sia stato causato da un segnale.

3.2 Gestione delle scritture deferred

Sul flusso a bassa priorità le scritture devono avvenire in modalità deferred, riuscendo però a mantenere l'interfaccia del driver in grado di notificare il risultato in maniera sincrona.
Per questo motivo, le operazioni di scrittura effettuate dalle work queue non possono mai fallire, quindi è stato messo in atto quanto segue:

- tutti i controlli relativamente alla possibilità di prendere il lock e di poter scrivere i dati all'interno del device vengono effettuati prima di schedulare il lavoro deferred
- in base a quanti dati verranno scritti dalla work queue, viene sottratta la quantità corrispondente al campo `total_free_bytes`: in questo modo, se anche il lavoro della work queue non viene immediatamente schedulato, successive chiamate alla write nel caso in cui non vi sia spazio sufficiente falliranno, e quando verrò poi schedulato il thread deferred sarà sicuro di poter scrivere tali bytes.

Per l'implementazione del lavoro deferred è stato scelto di utilizzare le **work queue**: siccome all'interno della funzione di scrittura è necessario ri-acquisire il lock, nella funzione di scrittura passata come argomento della work queue viene usata l'API `mutex_lock()`, che può portare il thread in sleep nel caso in cui il lock non sia disponibile.
Le work queue sono in grado per costruzione di riconoscere l'evenienza per cui il lavoro sia bloccante.
Il parametro passato alla funzione della work queue, la `do_wq_write`, è l'indirizzo della struttura di dati riportata in seguito:

```
1  typedef struct _packed_write_data_wq{
2    void *buffer;
3    char *data;
4    int minor;
5    size_t len;
6    struct work_struct the_work;
7  } packed_data_wq;
8
```

nel primo campo viene salvato l'indirizzo della struttura di dati stessa, in modo da poter accedere agli altri campi mediante la macro `container_of`.

3.3 Funzione `write_data`

La funzione viene utilizzata per scrivere effettivamente i byte, per entrambe i flussi corrispondenti ai due livello di priorità.
All'interno di tale funzione, viene inizialmente effettuato un calcolo di quante pagine sarà necessario allocare per poter scrivere i byte richiesti dall'utente, in modo da evitare di dover allocare "on-the-fly" mentre si effettua la scrittura.
Successivamente, la scrittura avviene all'interno di un while, dove ogni volta viene decrementata dalla taglia di byte da scrivere quanto è stato scritto nell'iterazione. Questo poiché è possibile che una scrittura cada a cavallo di più di una pagina di memoria, quindi nel caso in cui questo avvenga, si passa a scrivere sulla pagina successiva.
La copia dei dati avviene in due modi diversi in base alla priorità:

- alta: viene effettuata mediante l'API `copy_from_user`, in quanto il buffer passato è quello dell'utente che ha invocato la `write`
- priorità bassa: l'API non è utilizzabile, in quanto è stata usata in precedenza per effettuare una copia locale del buffer utente in un buffer kernel, viene quindi eseguita un iterazioni ciclica per copiare i byte nella struttura di dati apposita.

4 Controllo della sessione di I/O

Per quanto riguarda il controllo della sessione di I/O, è stata utilizzata la system call `ioctl`. La scelta è stata quella di passare come secondo parametro della funzione il comando da gestire, che è in realtà un campo della
`enum ctl_ops{SET_PRIO=1, SET_BLOCKING=3, SET_OPENCLOSE=4}` e come 3° parametro l'effettivo valore da impostare.
Di seguito, vengono riassunti i possibili casi di utilizzo della `ioctl` per la gestione della singola sessione di I/O:

- `SET_PRIO`: usato per cambiare la priorità del flusso di dati. Per il 3° parametro:
  - 0 indica l'utilizzo del flusso a bassa priorità
  - 1 indica invece l'uso del flusso ad alta priorità
- `SET_BLOCKING`: rende lo operazioni bloccanti o non bloccanti. Per il 3° parametro:
  - 0 indica che le operazioni non sono bloccanti
  - un valore diverso da 0 che le operazioni sono bloccanti e viene utilizzato come valore del timeout nella funzione di wait.
- `SET_OPENCLOSE`: configura l'apertura o la chiusura del device file in termini del minor number specifico. Per il 3° parametro:
  - 0 indica che il device è aperto
  - 1 che il device è chiuso.
    Viene quindi portata ad 1 la entry corrispondente al minor number del device file su cui è stata invoca la funzione, tale entry è quella dell'array `enable_disable_array`,

che è dichiarato come parametro del modulo.
Nella system call `dev_open` verrà quindi verificato, controllando tale array, se è permesso aprire una nuova sessione di I/O verso il device file.

Per far si che le informazioni vengano cambiate solo per la specifica sessione di I/O, è stato sfruttato il campo `private_data` della struttura di dati `file`, passando come valore la struttura riassunta in seguito:

```
1  typedef struct _io_sess_info{
2    int priority;
3    int is_blocking;
4    long timeout;
5  } io_sess_info;
6
```

Tale struttura viene allocata nella `dev_open` e deallocata nella `dev_close`, inoltre viene consultata dentro tutte le funzioni del modulo per verificare le informazioni relative alla sessione di I/O.

5 Parametri del modulo

Vi sono 5 parametri per il modulo software, esposti nel VFS all'interno della directory `/sys/modules/multistream-driver/parameters`.
Tutti i parametri sono degli array di `unsigned long`, dichiarati mediante `module_param_array` e di taglia pari al numero di minor numbers pilotabili col driver.
Vengono illustrati in seguito, con il relativo utilizzo:

- `enable_disable_array`: ciascuna entry ha valore 0 se è possibile aprire delle sessioni verso il device file, 1 altrimenti. È possibile lavorare con la granularità della singola entry mediante l'utilizzo della `ioctl`, come illustrato sopra;

- `high_data_count` e low_data_count: rispettivamente ogni entry rappresenta il numero di bytes attualmente presenti nei due flussi di priorità, per ciascun minor number.
  Nelle funzioni di `read` e `write` vengono ogni volta decrementati ed incrementati

- `high_wait_data` e `low_wait_data`: ogni entry riporta il numero di thread che sono attualmente in attesa per leggere o scrivere dei dati su ambo i flussi.
  Anche in questo caso, la entry viene incrementata di 1 quando un thread viene messo in sleep sulla wait queue e decrementata di uno al ritorno da tale sleep.

Fatta eccezione per l'array `enable_disable_array`, tutti gli altri parametri del modulo vengono creati con i permessi di accesso al relativo pseudo-file pari a "0440", in quanto non deve essere permesso di poter modificare il valore all'interno di essi dall'esterno del modulo.

6 Init e cleanup del modulo

Le funzioni di `init_module` e `cleanup_module` vengono invocate una sola volta, rispettivamente all'atto del montaggio e dello smontaggio del modulo dal kernel.

6.1 `init_module`

Nella `init_module` viene effettuata l'inizializzazione delle strutture di dati utilizzare dal modulo, in particolare della lista collegata che rappresenterà, per ognuno dei minor possibili, il device file in memoria.
Qui infatti, viene allocata la testa della lista, che per default sarà sempre diversa da `NULL`. Questo permette di non dover mai controllare all'interno del kernel, nel caso di inserimenti e rimozioni dalla lista, se la testa sia pari a `NULL`.
Vengono poi allocate le prime pagine per i flussi ad alta e bassa priorità, infine viene registrato il device driver mediante la `__register_chrdev`, facendosi assegnare per default un major number libero dal Sistema Operativo.

6.2 `cleanup_module`

Nella funzioni di cleanup, viene resa all'allocatore tutta la memoria allocata nella `init_module` ed inoltre viene verificato anche se è necessario deallocare altri nodi della lista, magari allocati da chiamate di write.
Vengono quindi restituite la memoria occupata dalle pagine di memoria del device mediante la `free_page`, per poi deallocare il nodo mediante la `kfree`.
Infine, viene de-registrato il driver mediante la `unregister_chrdev`.

7 Installazione, rimozione del modulo ed utilizzo

Tutti i comandi riportati in seguito presuppongono la possibilità di utilizzare una shell con privilegi di root.
Il codice del modulo è presente nel file `src/driver/multistream-driver.c`, le strutture di dati usate sono invece riportate nel file `src/driver/structs/structs.h`.
Per la compilazione del modulo, viene fornito un apposito `Makefile`, che permette di invocare due comandi:

- mediante `make all` viene generato il kernel object `multistream-driver.ko`, che può essere inserito all'interno del kernel mediante l'utilizzo del programma `insmod`
- con il comando `rmmod`, invece, è possibile smontare il modulo dal kernel
- con il comando `make clean` è possibile rimuovere tutti i file generati dalla compilazione.

Dopo aver installato il modulo, è stato fatto auditing mediante `printk`, sfruttando il buffer kernel `dmesg` (accessibile con l'omonimo comando da shell) per poter verificare quale major number è stato assegnato al driver. Per l'utilizzo del modulo, sono stati prodotti due software:

- il programma `src/node_creations/nodes.c` permette di creare dei nodi nel sistema mediante l'utilizzo della system call `mknod`. Tale software può essere compilato con il `Makefile` fornito e richiede come parametri il nome del file da creare, il major number ed il numero di minor number corrispettivi, difatti il file verrà creato con un nome che è quello passato a cui viene appeso il minor number progressivamente.
- il programma `src/user/user.c` permette invece di compilare e lanciare un programma utente.
  Mediante un'interfaccia utente molto semplice, è possibile interagire con il device driver per utilizzare tutte le funzionalità offerte da esso.