MULTIFLOW DEVICE DRIVER

Matteo Chiacchia
0300177

1. INTRODUZIONE

Il documento seguente riporta le scelte progettuali e la documentazione relative all'implementazione del modulo kernel "*multiflow_driver.c*".

1.1 Specifica

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 un'esecuzione asincrona (basata sul *deferred work*) 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, in entrambi i flussi.
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 *threads* attualmente in attesa per i dati lungo i due flussi di priorità

2. STRUCT DI GESTIONE E PARAMETRI

2.1 Gestione del device

Per la gestione dei *device file* è stato implementata una struttura dati che ha lo scopo di salvare dati e metadati relativi ad ognuno dei *minor number* possibili. È stato quindi creato un array di strutture in cui l'elemento in posizione i-esima è utilizzato per la gestione del file relativo all'i-esimo *minor number*.

La *struct* è la seguente:

```
typedef struct _object_state{
    long available_bytes; //numero di bytes che il device file può ancora gestire
    Flow flows[FLOWS];    //flussi (bassa e alta priority) gestiti dal device file
}Object_state;
```

Dove:

- *long* ***available_bytes***: mantiene il numero di *bytes* logicamente disponibili nel device file. Nelle scritture asincrone, infatti, vengono riservati logicamente dei *bytes* che verranno poi effettivamente occupati nelle *deferred_write.* Nel nostro caso è stata scelta una capacità di ***1MB***.
- *Flow* ***flows[FLOWS]***: array di *struct _flow* per la gestione dei due (*FLOWS = 2*) flussi di priorità del *device file.*

Quest'ultima *struct* è rappresentata nel seguente modo:

```
typedef struct _flow{
    struct mutex operation_synchronizer; //lock di gestione dei thread concorrenti
    Object_content *obj_head;  //testa della lista collegata contenente i blocchi di bytes
    wait_queue_head_t wait_queue; //wait_queue in cui sono presenti i task in attesa del lock
}Flow;
```

Dove:

- *struct mutex* ***operation_synchronizer***: utilizzato per evitare che due o più *thread* concorrenti modifichino dati/informazioni della struttura dati condivisa. Nel momento in cui un *thread* deve leggere o scrivere, quindi, effettua

un'operazione di *lock* sul *mutex* per poi eseguire un *unlock* una volta completata l'operazione.

- *wait_queue_head_t* ***wait_queue***: lista dei *thread* in attesa di prendere il *lock.* Quando si ha un'operazione bloccante e il *mutex* non è disponibile, viene messo il *thread* in attesa all'interno di questa *wait_queue*.
- *Object_content* ***obj_head***: puntatore alla testa della struttura dati contenente i *bytes* scritti e relativi metadati.

Questa *struct* è rappresentata nel seguente modo:

```
typedef struct _object_content{
    int last_offset_read; //ultimo offset letto
    char *stream_content;  //bytes scritti
    struct _object_content *next; //puntatore al blocco successivo
}Object_content;
```

Dove:

- *int* ***last_offset_read***: rappresenta il byte successivo alll'ultimo *byte* acceduto durante le letture precedenti. Tutti i *bytes* precedenti non sono quindi più leggibili logicamente.
- *char* ***stream_content***: contenuto effettivo di una scrittura nel *device file.*
- *struct _object_content* ***next***: puntatore al blocco di *bytes* successivo.

Viene utilizzata *l'API* ***kzalloc*** per ottenere una memoria allocata settata a 0.
Si è scelto di eliminare dalla memoria fisica i *bytes* solamente nel momento in cui il blocco è stato letto tutto. Quando viene letta solo una parte del blocco, infatti, viene spostato l'offset di lettura ma i *bytes* vengono eliminati solo logicamente. Una volta che l'offset diventa maggiore della posizione dell'ultimo byte presente nel blocco si effettua la ***kfree*** eliminando questo dalla memoria fisica si fa puntare la testa della lista al blocco successivo.

2.2 Gestione della sessione di I/O

Per il controllo della sessione di ***I/O*** è stata utilizzata una *struct* che ha il compito di salvare il tipo di sessione che si sta attualmente utilizzando.
Essa è definita nel seguente modo:

```
typedef struct _session{
    int priority; //prorità delle operazioni (alta o bassa)
    int blocking;  //operazioni (write or read) bloccanti o non bloccanti
    unsigned long timeout; //timeout che regola l'attivazione delle blocking operations
}Session;
```

I campi possono avere i seguenti valori:

- ***Priority***
  - *LOW_PRIORITY = 0*
  - *HIGH_PRIORITY = 1*
- ***Blocking***
  - *BLOCKING = 0*
  - *NON_BLOCKING = 1*
- ***Timeout***
  - Valore numerico rappresentato in millisecondi

In fase di inizializzazione è stato scelto di avere una sessione del tipo:

- *session->priority =* ***HIGH_PRIORITY***
- *session->blocking =* ***NON_BLOCKING***
- *session->timeout =* ***0***

I valori dei parametri possono essere modificati tramite la *system call* **`ioctl`**, come vedremo in seguito.

2.3 Gestione delle *deferred write*

Le *deferred write* vengono effettuate nel flusso a bassa priorità. L'idea, infatti, è di eseguire l'operazione di *write* in maniera asincrona riuscendo però a notificare il risultato in maniera sincrona.
Per l'effettiva implementazione della scrittura asincrona è stato scelto di utilizzare le *work queues* che vengono salvate all'interno di una struttura dati che mantiene anche altre informazioni.

La struttura è del seguente tipo:

```
typedef struct _packed_work_struct{
    const char *data; // Puntatore al buffer kernel temporaneo dove salvare i dati da scrivere.
    size_t len;  // Quantità di dati da scrivere, corrisponde alla lunghezza del buffer 'data'.
    Object_content *new_content; // Puntatore al blocco vuoto per la scrittura successiva.
    int minor; // Minor number del driver su cui si sta operando.
    struct work_struct work; // Struttura di deferred work
}packed_work_struct;
```

- *const char* ***data***: buffer *kernel* in cui vengono spostati i *bytes* scritti dall'utente. Una volta che la scrittura può essere effettuata, questi *bytes* vengono scritti all'interno del *device file*.
- *size_t* ***len***: quantità di bytes da scrivere nel *device file*. Questo campo è utilizzato anche per occupare logicamente la memoria prima che la scrittura venga effettivamente compiuta.
- *Object_content* ***new_content***: blocco vuoto allocato per appenderlo in seguito alla lista collegata dei blocchi di scritture.
- *int* ***minor***: *minor number* del *device file* con cui è stata aperta la sessione di *I/O*
- *struct work_struct* ***work***: viene salvato il lavoro da eseguire.

Il lavoro da effettuare viene inizializzato tramite ***__INIT_WORK*** ed eseguito tramite ***schedule_work.***

2.4 Parametri del modulo

Sono stati dichiarati ***5*** parametri per il modulo software, i quali sono esposti nel *VFS* all'interno della *directory* ***/sys/module/multiflow_driver/parameters***.
Questi sono degli array di *int,* dichiarati tramite ***module_param_array***, e hanno una dimensione uguale al numero di *device files* gestibile dal *device* (*MINORS*= 128).
I parametri sono i seguenti:

- ***enabled_device[MINORS]***: array in cui ad ogni posizione è specificato se il *device file* è abilitato o meno (*DISABLED = 0, ENABLED = 1*). Se non dovesse esserlo, ogni tentativo di apertura di una sessione non andrà a buon fine.
- ***hp_bytes[MINORS]***: array in cui ad ogni posizione è specificato il numero di *bytes* attualmente presenti nel flusso ad alta priorità.
- ***lp_bytes[MINORS]***: array in cui ad ogni posizione è specificato il numero di *bytes* attualmente presenti nel flusso a bassa priorità
- ***hp_threads[MINORS]***: array in cui ad ogni posizione è specificato il numero di *threads* ad alta priorità attualmente in attesa.
- ***lp_threads[MINORS]***: array in cui ad ogni posizione è specificato il numero di *threads* a bassa priorità attualmente in attesa.

Ogni volta che avviene una scrittura nell'i-esimo *device file* viene aumentato il valore di ***hp_bytes[i]*** o di ***lp_bytes[i]*** del numero di bytes scritti dall'utente.

Viceversa, nelle operazioni di lettura il valore viene diminuito del numero di bytes consumati dall'utente.
In ***hp_threads[i]*** e ***lp_threads[i]*** la entry viene incrementata di 1 ogni volta che un *thread* si mette in attesa di ottenere il *lock* e decrementata di uno nel momento in cui un *thread* lo ottiene.

3. OPERAZIONI DI SCRITTURA E LETTURA

Le operazioni di scrittura e lettura del *device* sono rispettivamente ***dev_write*** e ***dev_read***. In entrambi le funzioni vengono effettuati dei controlli per verificare se l'operazione può essere effettivamente eseguita:

- In ***dev_write*** si verifica che il numero di *bytes* scritto dall'utente è maggiore del numero di *bytes* ancora disponibili all'interno del *device file*.
- In ***dev_read***, invece, il controllo viene eseguito all'interno nella funzione di appoggio `read_bytes` e consiste nel verificare che sono presenti *bytes* da leggere.
- In entrambi le funzioni si tenta, come verrà spiegato in seguito, di acquisire il ***lock*** (tramite l'*API* ***mutex_trylock***).

Nel caso una di queste condizioni non fosse verificata viene ritornato un errore in quanto l'operazione non può essere effettuata.

3.1 Operazioni di scrittura

Le operazioni di scrittura sono diverse a seconda della priorità attuale.
Nel caso di ***HIGH PRIORITY*** viene invocata la funzione ***hp_write*** che opera nel seguente modo.

- Effettua un ***try_lock*** e nel caso non riesca ad acquisire il lock ritorna un errore.
- Copia i *bytes* scritti dall'utente all'interno di un buffer tramite ***copy_from_user*** e salva l'indirizzo di memoria del buffer all'interno del campo *stream_content* dell'ultimo blocco della struttura dati condivisa.
- Appende un blocco *empty* al blocco precedentemente scritto, aggiorna i parametri del modulo e ritorna il numero di *bytes* effettivamente scritti.

Nel caso di ***LOW_PRIORITY***, invece, le operazioni sono più articolate. Viene chiamata la funzione ***`write_work_schedule`***. Essa ha il compito di inizializzare la *packed_work_struct* descritta in precedenza e mettere l'operazione all'interno di una *work_queue.*
Le operazioni effettuate sono le seguenti:

- Si effettua un ***try_lock*** e nel caso non si riesca ad acquisire il *lock* si ritorna un errore.
- Si inizializza un'istanza della *packed_work_struct* e un *buffer kernel* temporaneo di appoggio. Si copiano i *bytes* scritti dall'utente all'interno del *buffer* (tramite ***`copy_from_user`***) e si aggiornano i parametri del modulo.
    - Viene diminuito il valore di bytes disponibili all'interno del *device file.*
    - Si aumenta il valore del parametro ***lp_bytes[i]*** del numero di *bytes* che saranno scritti.
- Si inizializza il lavoro da effettuare tramite ***`__INIT_WORK`***, lo esegue tramite ***`schedule_work`*** e ritorna il numero di bytes da scrivere in modo tale da poterli occupare logicamente ed essere in grado di notificare l'utente in maniera sincrona.

È importante considerare il fatto che le scritture *LOW_PRIORITY* sono effettuate in maniera asincrona, quindi una volta che l'utente è stato notificato la scrittura **non** può fallire. Per questo motivo viene riservato spazio all'interno del *device file,* in modo tale da esser certi che il *thread deferred,* quando sarà schedulato, avrà a disposizione abbastanza *bytes* su cui scrivere. Si nota che, riservando spazio per le scritture *deferred,* eventuali scritture successive potrebbero fallire a causa della mancanza di memoria fisica, quando in realtà la memoria è stata occupata solo a livello logico.

Il *deferred work* da eseguire è stato implementato tramite la funzione ***delayed_write***, eseguita dal *kworker daemon* nel momento in cui viene schedulato. Essa opera nel seguente modo:

- Riprende l'indirizzo della *packed_work_struct* iniziale (dove salvati tutti i dati relativi alla scrittura) tramite l'*API* ***`container_of`***.
- Si mette in attesa di prendere il *lock* e rimane in questo stato fino al momento in cui non viene schedulata. Allo stesso tempo viene aumentato il parametro ***lp_threads[minor]*** spiegato in precedenza.

```
__sync_fetch_and_add(&lp_threads[minor], 1);
mutex_lock(&(flow->operation_synchronizer));
__sync_fetch_and_add(&lp_threads[minor], -1);
```

- Una volta schedulata aggiunge i *bytes* scritti dall'utente all'ultimo blocco della lista collegata, per poi appendere un blocco *empty* a questo seguendo un ragionamento analogo al caso *HIGH_PRIORITY.*

3.2 Operazioni di lettura

Le operazioni di lettura vengono svolte nello stesso modo in entrambi i flussi di priorità.
L'idea è cercare di acquisire il *lock* e, in caso positivo, se dovessero essere presenti *bytes* da consumare, si entra in un ciclo che opera nel seguente modo.

- Ad ogni iterazione si verifica se la quantità di *bytes* ancora da leggere (***bytes_to_read***) è **minore, uguale o maggiore** di quella dei *bytes* ancora da consumare nel primo blocco della lista.
- Nel **primo** e **secondo** caso si copiano i *bytes* da leggere in un buffer utente, si aumenta il valore di ***last_offset_read*** della quantità letta e, nel caso questo dovesse superare il valore della lunghezza dello *stream_content* del blocco, si elimina definitivamente quest'ultimo e si definisce il blocco successivo come *head* della lista.
- Nel **terzo** caso, invece, si copiano nel buffer utente tutti i *bytes* ancora da consumare del primo blocco della lista, si aumenta il valore del numero di *bytes* letti (***bytes_read***) e analogamente si diminuisce dello stesso valore il numero di *bytes* ancora da leggere (***bytes_to_read***). Infine si elimina questo blocco, si definisce il successivo come *head* della lista e si comincia una nuova l'iterazione.

3.3 Operazioni bloccanti

La gestione delle operazioni bloccanti è effettuata nel momento in cui il *thread* tenta di prendere il ***lock***. Se l'operazione è **non bloccante** e l'*API* ***mutex_trylock*** fallisce, il *thread* ritorna un messaggio d'errore non potendo mettersi in attesa.
Viceversa, se l'operazione dovesse essere **bloccante** e *mutex_trylock* dovesse fallire, il *thread* entra in attesa per un tempo massimo specificato da ***session->timeout***.
Si è utilizzata l'*API* ***wait_event_timeout*** che mette il processo in *sleep* fino al momento in cui la condizione (in questo caso l'ottenimento del *lock*) è verificata.

Questa condizione è controllata ogni volta che la *waitqueue* del flusso attuale è attivata da una ***wake_up***.

- Se il valore di ritorno di ***wait_event_timeout*** è 1, allora il *thread* è riuscito a ottenere il *lock* prima dello scadere del *timeout* e può eseguire le varie operazioni come spiegato precedentemente.
- Se il valore di ritorno di ***wait_event_timeout*** è 0, allora il *timeout* è scaduto e viene ritornato un messaggio d'errore, non essendo il *thread* in grado di svolgere l'operazione soddisfando le richieste dell'utente.

4. CONTROLLO DELLA SESSIONE DI I/O

Per quanto riguarda il controllo e la gestione della sessione di *I/O* è stata utilizzata la *system call* ***ioctl***. Tramite questa è possibile cambiare i valori della sessione attuale. Si utilizza il secondo parametro della *system call* per decidere il tipo di cambiamento da eseguire:

- Cambiare il tipo di *priority*: ***CHANGE_PRIORITY_IOCTL (3)***
    - *Priority* scelta a seconda del valore del terzo parametro:
        - 0 indica di cambiare la priorità a *LOW_PRIORITY*
        - 1 indica di cambiare la priorità a *HIGH_PRIORITY*

- Cambiare il tipo di *blocking*: ***CHANGE_BLOCKING_IOCTL (4)***
    - *Blocking* scelto a seconda del valore del terzo parametro
        - 0 indica che le operazioni dovranno essere *NON_BLOCKING*
        - Un valore maggiore di 0 indica che le operazioni dovranno essere BLOCKING e il valore del *timeout* impostato al valore considerato.

Ogni tipo di operazione eseguita viene salvata all'interno del relativo campo della *struct* ***session*** in modo tale da mantenere queste informazioni salvate.

N.B: Quando un device file è aperto viene instaurata una sessione di default:

- *HIGH PRIORITY*
- *NON BLOCKING*
- *TIMEOUT = 0*

5. INIT E CLEANUP DEL MODULO

Per il montaggio e smontaggio del modulo vengono invocate rispettivamente le funzioni ***init_module*** e ***cleanup_module.***

5.1 init_module

Nella *init_module* vengono inizializzate tutte le strutture dati necessarie alla gestione del modulo. In particolare, viene allocato spazio per l'inizializzazione della lista collegata rappresentante il *device file* in memoria. Viene, quindi, allocata la testa della lista in modo da avere sempre il primo blocco diverso da *NULL* ed evitare la gestione di questo ulteriore caso.
Si inizializzano anche il *mutex* e la *wait_queue* per ogni flusso di ogni *minor number*, rispettivamente tramite le *API* ***mutex_init*** e ***init_waitqueue_head***.
Viene infine registrato il *device driver* mediante la ***__register_chrdev*** a cui viene assegnato un *major number* dal Sistema Operativo.

5.2 cleanup_module

Nella *cleanup_module* viene liberata tutta la memoria allocata durante l'inizializzazione del modulo. Vengono, quindi, eseguite iterativamente delle chiamate a ***kfree*** per rimuovere dalla memoria i blocchi (e il contenuto presente) della lista collegata rappresentante il *device file*.
Finita l'iterazione, il modulo viene de-registrato tramite ***unregister_chrdev***.

6. INSTALLAZIONE RIMOZIONE E UTILIZZO DEL MODULO

Sono presenti due directory

- ***/driver*** dedicata all'implementazione del modulo. I file presenti sono
  - ***multiflow_driver.c***: codice del modulo.
  - ***lock_functions.h***: contiene le funzioni di appoggio per la gestione delle operazioni bloccanti e non.

    - ***read_write_functions.h***: contiene le funzioni di appoggio per la gestione delle operazioni di lettura e scrittura.
    - ***structs.h***: contiene le strutture dati utilizzate e analizzate precedentemente.
    - ***values.h***: contiene le definizioni delle *MACRO* e dei parametri del modulo analizzati precedentemente.
    - ***module.sh***: script *bash* per compilare e montare il modulo.
- ***/user*** dedicata all'implementazione di un programma utente per interfacciarsi con il modulo.
    - ***user_test.c***: codice utente che permette di interfacciarsi con il modulo tramite riga di comando.

I comandi da eseguire per il montaggio e smontaggio del modulo sono i seguenti (si presuppone che i comandi vengano chiamati avendo i privilegi di ***root***):

- ***`make all`***: genera il *kernel object* ***multiflow_driver.ko***
- ***`insmod multiflow_driver.ko`***: monta il *kernel object* all'interno del *kernel*
- ***`rmmod multiflow_driver.ko`***: smonta il modulo dal *kernel*
- ***`make clean`***: elimina i file generati durante la compilazione

Per verificare quale sia il *major number* assegnato al modulo si può utilizzare il comando ***`dmesg`***.
È stato implementato il codice utente *user_test.c* che fornisce la possibilità di eseguire le seguenti operazioni:

1. Scrivere sul *device.*
2. Leggere dal *device*.
3. Cambiare il tipo di sessione.
    - *Priority, blocking* e valore di *timeout.*
4. Leggere i parametri (*paragrafo 2.4*) del *device file* con cui si è instaurata una sessione.
5. Cambiare l'abilitazione di un *device file* con uno specifico *minor number* specificato a *runtime.*
6. Chiudere la sessione e terminare il programma.

È importante sottolineare che, una volta lanciato il programma, vengono creati 128 nodi con *minor numbers* {0, 1, 2, …, 127} tramite il comando ***`mknod`***. In seguito verrà

instaurata una sessione con il nodo avente il *minor number* specificato durante il lancio del programma.
Il comando da utilizzare è, infatti, il seguente:

- ***sudo ./user [device_path] [Major_number] [Minor_number]***