Progetto Sistemi Operativi Avanzati 2021/2022

Multiflow Device Driver

Danilo Dell'Orco 0300229

1 Sommario

2 Introduzione....3
2.1 Specifica: Multiflow Device....3
3 Strutture Dati e Parametri....4
3.1 Stream di Dati....4
3.2 Gestione del Dispositivo....4
3.3 Gestione della Sessione....6
3.4 Packed Work Struct....6
3.5 Parametri del Modulo....7
3.5.1 Lettura e Gestione dei parametri....7
3.6 File Operations....8
4 Operazioni di scrittura....9
4.1 Scrittura sul Data Stream (`write_on_stream`)....9
4.2 Ottenimento del lock in scrittura (`try_mutex_lock`)....11
4.2.1 Scrittura ad alta priorità....11
4.2.2 Scrittura a bassa priorità....12
5 Operazioni di Lettura....13
5.1 Scansione e Lettura dei Blocchi....13
5.1.1 Partial Block Reading....14
5.1.2 Full Block Reading....15
5.1.3 Tail Block Reading....16
5.2 Ottenimento del lock in lettura (`try_mutex_lock`)....17
6 Altre File Operations....18
6.1 Inizializzazione e cleanup del modulo....18
6.2 Apertura e chiusura di un device....18
6.3 Gestione della Sessione....19
7 Utilizzo del Modulo....20
7.1 Organizzazione della Repository....20
7.2 Montaggio e Rimozione del Modulo....20
7.3 User CLI....21
7.3.1 Operazioni sui device....22
7.3.2 Operazioni sulla sessione....22
7.3.3 Gestione dei dispositivi....23
7.3.4 Altri comandi....23

2 Introduzione

Lo scopo del documento è quello di descrivere e documentare l'implementazione del **Multiflow Device Driver**, motivando le scelte progettuali effettuate nello sviluppo del modulo e come poter interagire con esso.

2.1 Specifica

La specifica è relativa a un device driver 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 un' operazione 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 su delayed work) delle operazioni di scrittura, mantenendo comunque l'interfaccia in grado di notificare in modo sincrono il risultato. Le operazioni di lettura sono invece eseguite tutte 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.

Devono essere implementati anche alcuni parametri del modulo Linux ed alcune funzioni 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).

3 Strutture Dati e Parametri

In questa sezione descriviamo le strutture dati definite per rappresentare i principali componenti del modulo. Tali strutture verranno ampiamente utilizzate poi nelle operazioni effettive verso il dispositivo, ed i dettagli di come queste sono implementate verranno descritti in seguito.

3.1 Stream di Dati (`stream_block`)

Tramite l'utilizzo del device driver un thread può leggere o scrivere segmenti di dati. Per realizzare questo meccanismo, lo stream di dati è stato implementato come una **lista collegata** di `stream_block`, in cui ogni `stream_block` mantiene il contenuto di una specifica operazione di scrittura. La relativa struttura dati è definita come segue:

```
typedef struct _stream_block {
    int read_offset;
    char *stream_content;
    struct _stream_block *next;
    int id;
} stream_block;
```

- `read_offset`: tiene traccia della posizione dell'ultima lettura all'interno del singolo blocco. Questo perché potrebbero esserci letture che non consumano totalmente i dati di un blocco, ed una lettura successiva deve quindi continuare dal primo byte non letto in precedenza.
- `stream_content`: puntatore all'area di memoria che contiene gli effettivi dati scritti sul blocco.
- `next`: puntatore al blocco successivo dello stream.
- `id`: codice numerico che identifica il blocco nello stream. Non svolge nessuna funzione nelle operazioni di lettura e scrittura, ma risulta utile per visualizzare ed analizzare lo stato dei singoli blocchi.

3.2 Gestione del Dispositivo

Lo stato di ogni dispositivo è mantenuto tramite una struttura `object_state`, che contiene tutte le informazioni per poter operare sul device. In totale devono essere gestiti *128 dispositivi*, quindi si utilizza un array `objects` contenente *128* strutture `object_state`; `objects[N]` mantiene lo stato del dispositivo con minor number `N`.

La struttura è così definita:

```
typedef struct _object_state {
    long available_bytes;
    flow_state priority_flow[NUM_FLOWS];
} object_state;
```

- `available_bytes`: tiene traccia dei bytes liberi sul dispositivo. Questa viene inizializzata al valore `MAX_SIZE_BYTES`, e viene aggiornata a seguito di ogni scrittura e lettura su uno dei due flussi di priorità. Questo campo è fondamentale nelle *deferred write* poiché permette di verificare se c'è spazio disponibile sul device prima di schedulare l'operazione, e riservare dei bytes per tale scrittura, che non potrà fallire. Questo meccanismo verrà descritto in dettaglio successivamente.

- `priority_flow[NUM_FLOWS]`: mantiene i dati scritti sui flussi ad alta e bassa priorità, ed i relativi metadati per la gestione dei blocchi. Si mantengono due strutture distinte, in modo da poter operare in parallelo sui due flussi di priorità.

  - `priority_flow[0]`: flusso a bassa priorità.

  - `priority_flow[1]`: flusso ad alta priorità.

Le informazioni sullo stream sono rappresentate nella struttura `flow_state`, definita dai seguenti campi:

```
typedef struct _flow_state {
    struct mutex operation_synchronizer;
    stream_block *head;
    stream_block *tail;
    wait_queue_head_t wait_queue;
} flow_state;
```

- `operation_synchronizer`: mutex che permette di sincronizzare le operazioni di lettura e scrittura sul flusso. Il driver può essere utilizzato anche su sessioni di I/O differenti, ed è necessario quindi gestire la *concorrenza sul singolo device file* per garantire che un solo thread per volta possa operare in lettura/scrittura.

- `head`: puntatore al primo `stream_block` del flusso. La lettura inizia sempre dal blocco puntato da questo campo, seguendo la politica `FIFO`.

- `tail`: puntatore all'ultimo `stream_block` del flusso. Quando si effettua una scrittura viene appeso un nuovo blocco in coda allo stream, e tramite questo campo si velocizza tale operazione; infatti basterà settare il campo `next` del blocco puntato da `tail` all'indirizzo del nuovo blocco, senza dover scorrere l'intera *linked list.*

- `wait_queue`: Mantiene la *Wait Queue* del singolo flusso di priorità. Nelle operazioni bloccanti se il mutex non è disponibile il task viene messo in sleep e inserito in questa coda di attesa. La gestione delle operazioni bloccanti verrà descritta nei paragrafi successivi.

3.3 Gestione della Sessione

Il driver permette di modificare la sessione di I/O tramite l'utilizzo di `ioctl()`. Le informazioni della sessione vengono tenute in una struttura `session_state`, associata al campo `private_data` del file aperto. La struttura è definita come segue:

```
typedef struct _session_state {
    int blocking;
    int priority;
    int timeout;
} session_state;
```

- `blocking`: indica se la sessione utilizza operazioni bloccanti o non-bloccanti.
  - `BLOCKING` (0) / `NON_BLOCKING` (1)
- `priority`: indica se la sessione opera sul flusso ad alta o bassa priorità.
  - `LOW_PRIORITY` (0) / `HIGH_PRIORITY` (1)
- `timeout`: mantiene il timeout di l'attesa sul lock nelle operazioni bloccanti.

3.4 Packed Work Struct

Sul flusso a bassa priorità le scritture avvengono in maniera asincrona, utilizzando le *work queues* come meccanismo di *deferred work*. Qui viene utilizzata una struttura `packed_work_struct`, che contiene la `work_struct` del lavoro schedulato e altre informazioni necessarie per poi eseguire l'effettiva scrittura. Quando il demone di sistema esegue la *deferred write* si può risalire dalla `work_struct` alla `packed_work_struct` tramite `container_of()`.

```
typedef struct _packed_work_struct {
    const char *data;
    int minor;
    size_t len;
    session_state *session;
    struct work_struct work;
} packed_work_struct;
```

- `data`: puntatore ad un buffer kernel temporaneo contenente i dati che verranno successivamente scritti sul dispositivo.
- `minor`: minor number del dispositivo su cui si sta operando.
- `len`: quantità di byte da scrivere sullo stream.
- `session`: puntatore alla sessione verso il device file.
- `work`: mantiene il lavoro di scrittura che verrà eseguito successivamente. Questo viene inizializzato tramite `__INIT_WORK()` e schedulato tramite `schedule_work()`.

3.5 Parametri del Modulo

Il modulo deve esporre diversi parametri tramite VFS per mantenere lo stato di un dispositivo. Tali parametri sono stati dichiarati usando la macro `module_param_array`, in modo da definire un array di `unsigned long` con numero di elementi pari al numero di dispositivi controllati dal driver. In particolare gli array definiti sono i seguenti:

- `device_enabling`: specifica nell'elemento *i-esimo* se il device con minor `i` è abilitato oppure disabilitato.
  - `device_enabling[i]=0`: il dispositivo i-esimo è disabilitato e non è possibile aprire delle sessioni verso il rispettivo device file.
  - `device_enabling[i]=1`: il dispositivo i-esimo è abilitato ed è possibile aprire delle sessioni verso il rispettivo device file.
- `total_bytes_low`, `total_bytes_high`: specificano nell'elemento i-esimo il numero di bytes disponibili per la lettura sui due flussi di priorità. Il valore di questi parametri vengono incrementati e decrementati a seguito di ogni operazione di `read` e `write`.
- `waiting_threads_low`, `waiting_threads_high`: specificano nell'elemento i-esimo il numero di thread che sono in attesa per leggere o scrivere dati sui due flussi di priorità. Il valore di questi parametri viene incrementato di una unità ogni volta che un thread si mette in attesa del lock, e viene decrementato ogni volta che un thread acquisisce il lock o scade il suo timeout di attesa.

3.5.1 Lettura e Gestione dei parametri

Tutti i parametri sono accessibili tramite un apposito *pseudofile* nella directory `/sys/module/multistream-driver/parameters`. Il parametro `device_enabling` viene creato con i permessi di *lettura e scrittura* (`0660`) in quanto deve essere possibile abilitare o disabilitare un dispositivo cambiando il valore dell'apposita entry.

Invece i parametri `total_bytes` e `waiting_threads` sono creati con i soli permessi di lettura (`0440`), in quanto è possibile leggere tali valori, ma non deve essere consentita la modifica manuale. Infatti questi parametri vengono aggiornati automaticamente a seguito delle operazioni di lettura e scrittura, e manipolandoli diventerebbero di fatto inconsistenti.

Per abilitare o disabilitare il dispositivo si opera direttamente in scrittura sul file `/sys/module/multistream-driver/parameters/device_enabling`. Questo contiene 128 valori binari separati da una virgola, ognuno associato allo stato di un dispositivo. Tramite la CLI si può cambiare lo stato del singolo dispositivo: si utilizzano `getline` e `fputs` per modificare il byte in posizione `2*minor`, relativo al device che si vuole abilitare/disabilitare.

In alternativa si potrebbe implementare questo meccanismo tramite `ioctl` per comunicare con il modulo e modificare direttamente i valori nell'array `device_enabling`. Tuttavia, poiché tali parametri sono esposti nel VFS, si è deciso di sfruttare l'interazione diretta con i relativi file, in modo da poter gestire e visualizzare i dispositivi anche da terminale.

3.6 File Operations

La struttura `file_operations` definisce tutti i puntatori alle varie funzioni che realizzano effettivamente il *multiflow device driver*. In particolare le funzioni definite sono le seguenti:

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .write = dev_write,
    .read = dev_read,
    .open = dev_open,
    .release = dev_release,
    .unlocked_ioctl = dev_ioctl
};
```

4 Operazioni di scrittura

Le operazioni di scrittura sono implementate nella `dev_write`, che viene invocata a livello kernel quando a livello user si utilizza la system call `write()`. Prima di procedere con la scrittura, si effettua un controllo sullo spazio disponibile nel dispositivo; in particolare si verifica se il numero di byte da scrivere (`len`) è inferiore ai byte liberi sul dispositivo, mantenuti nel campo `object_state->available_bytes`.

Se lo *spazio libero non è sufficiente* per contenere la scrittura, questa di fatto non avviene e si notifica l'errore all'utente. Se invece il dispositivo ha *abbastanza bytes liberi* si continua con la `dev_write`, che ha un comportamento differente a seconda del flusso di priorità su cui si sta operando.

- **Scrittura ad alta priorità**: la scrittura avviene in maniera **sincrona**. Viene chiamata la funzione `write_on_stream`, che si occupa di scrivere fisicamente i byte sul flusso. Se la scrittura ha successo si aggiornano i valori `available_bytes` e `total_bytes_high` del dispositivo.
- **Scrittura a bassa priorità**: la scrittura avviene in maniera **asincrona** tramite *deferred work*. In questo caso viene chiamata la funzione `write_low` che si occupa di schedulare l'effettiva `write_on_stream`. E' stato già verificato che c'è spazio disponibile sul dispositivo, per cui si notifica immediatamente all'utente il risultato della scrittura, che non può fallire nell'ottenere il lock.

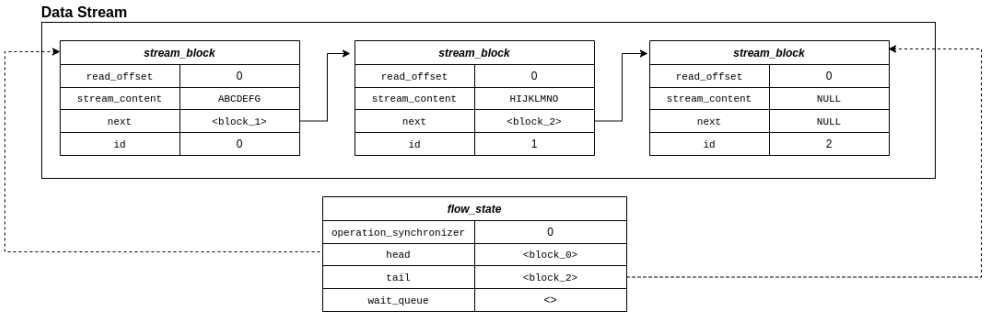
4.1 Scrittura sul Data Stream (`write_on_stream`)

La funzione `write_on_stream` viene utilizzata per entrambi i flussi di priorità, ed implementa la scrittura effettiva dei dati sullo stream. L'idea di base è quella di costruire un *Data Stream* tramite una *lista collegata* di oggetti `stream_block`. Ogni blocco punta al blocco successivo tramite il campo `next`, e per ogni operazione di scrittura viene creato un nuovo blocco dati, che viene messo in coda allo stream.

Per tenere traccia di quali blocchi compongono il flusso si utilizza la struttura `flow_state`, che indica in particolare quali sono il *primo* e l'*ultimo* blocco dello stream. Per costruzione l'ultimo blocco, puntato dal campo `tail`, ha sempre `stream_content=NULL`, ed è il blocco designato a *contenere la successiva operazione di scrittura*. Dunque l'operazione di scrittura consiste nel copiare i byte inseriti dall'utente nel campo `stream_content` dell'ultimo blocco.

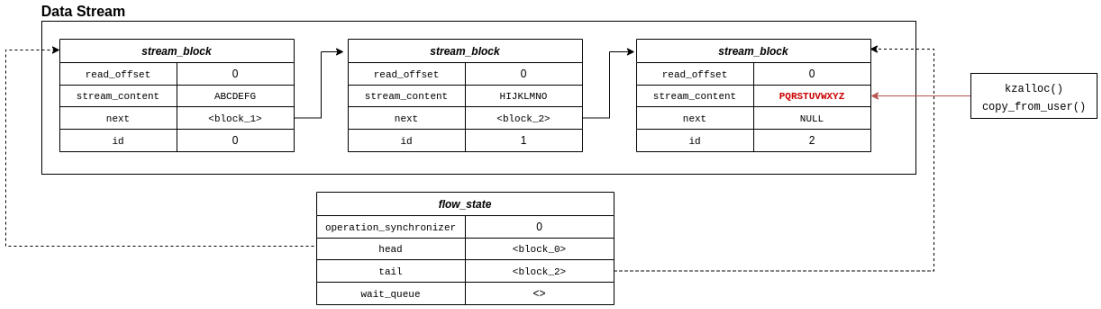
Prima di scrivere i byte è necessario ottenere il lock sul dispositivo. Per questo a partire dall'`object_state` del dispositivo si accede al `flow_state` dello stream, e poi si cerca di ottenere il lock su `flow_state->operation_synchronizer`. A tale scopo si utilizza la funzione `try_mutex_lock`, che internamente utilizza *logiche di locking differenti* a seconda del tipo di operazione (*lettura/scrittura*, *low/high*, *bloccante/non-bloccante*), e ritorna `LOCK_ACQUIRED` o `LOCK_NOT_ACQUIRED`. Le metodologie specifiche usate per prendere il lock verranno trattate durante l'analisi dei singoli tipi di operazione.

Se il processo ottiene il lock con successo, si prosegue con la scrittura sul flusso. Per descrivere nel dettaglio le operazioni effettuate analizziamo l'evoluzione della struttura `flow_state` e dei vari `stream_block`. Consideriamo la seguente situazione iniziale, in cui sono già state effettuate *due operazioni di scrittura* verso il dispositivo.



- Il primo blocco (`block_0`) è puntato dal campo `head` di `flow_state`, e punta tramite `stream_content` ai dati scritti nella prima operazione di scrittura.
- Il primo blocco (`block_1`) è puntato dal campo `next` del blocco precedente, e punta tramite `stream_content` ai dati scritti nella seconda scrittura.
- L'ultimo blocco (`block_2`) è puntato dal campo `tail` di `flow_state`. Il suo campo `stream_content` è `NULL` e verrà popolato in una successiva operazione di scrittura.

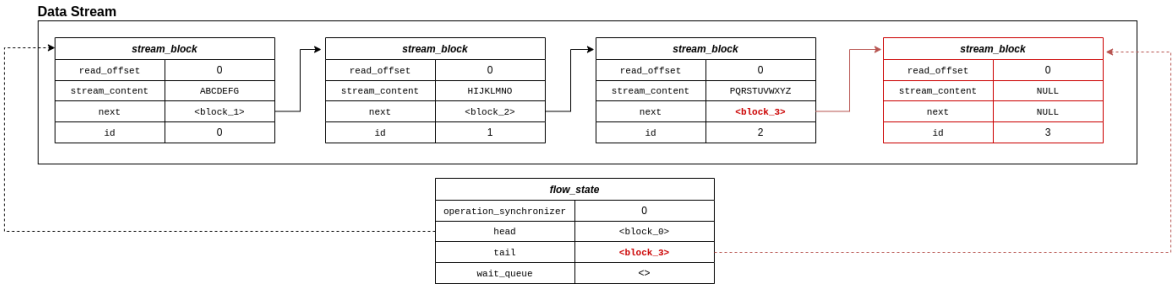
Quando viene effettuata una nuova operazione di `write` sullo stream si utilizza la `kzalloc` per allocare un buffer `data_buff` nel kernel. Tramite `copy_from_user` vengono copiati i dati scritti dall'utente su questo buffer, ed infine si assegna il campo `stream_content` dell'ultimo `stream_block` proprio a `data_buff`. Di fatto vengono quindi copiati i bytes da scrivere sul blocco in coda allo stream.



Infine viene si deve predisporre il flusso dati per poter ricevere le successive operazioni di scrittura. Sempre tramite `kzalloc` si alloca un nuovo `stream_block` con contenuto vuoto, a cui verranno associati i bytes della struttura successiva, e vengono poi aggiornati i vari puntatori che tengono traccia dello stream:

- Il campo `next` dell'ultimo blocco scritto punta al *nuovo blocco vuoto*.
- Il campo `head` del `flow_state` rimane *invariato*
- Il campo `tail` del `flow_state` punta al nuovo blocco vuoto appena aggiunto.

Vediamo che la situazione, oltre l'aggiunta di un nuovo blocco, è identica a quella descritta inizialmente, e pertanto il Data Stream è pronto per ricevere una nuova operazione di scrittura.



4.2 Ottenimento del lock in scrittura (`try_mutex_lock`)

Quando un thread esegue la `write_on_stream`, prima di procedere con la scrittura deve ottenere il lock sul dispositivo in modo da poter operare in maniera isolata sul device. Per fare questo si utilizza la funzione `try_mutex_lock`, che internamente utilizza logiche differenti a seconda del *flusso* su cui si scrive (alta o bassa priorità), e del *tipo* di operazione (bloccante o non-bloccante).

4.2.1 Scrittura ad alta priorità

Se la scrittura è sul flusso ad alta priorità, allora le operazioni devono avvenire in maniera **sincrona**, e per questo `try_mutex_lock` cerca di acquisire il lock tramite `mutex_trylock`; questa funzione ritorna `0` se il lock è disponibile e viene acquisito correttamente, altrimenti un numero che indica l'errore. Se l'operazione ha successo si prosegue quindi con la `write_on_stream`, altrimenti si deve valutare se l'operazione è *bloccante* o *non-bloccante*.

Operazione non-bloccante: il lock non è stato acquisito e la scrittura deve fallire. La `try_mutex_lock` ritorna quindi l'errore `LOCK_NOT_ACQUIRED` al chiamante `write_on_stream`, che non prosegue con la scrittura e notifica l'errore.

Operazione bloccante: il lock non è stato acquisito e quindi il task viene messo nella W*ait Queue* del flusso, mantenuta nel campo `flow_state->wait_queue`. Il processo deve restare in *sleep* per un tempo massimo specificato da `session_state->timeout`, per cui si utilizza la `wait_event_timeout` a tale scopo. Questa funzione prende come parametro il puntatore alla waitqueue, una funzione da eseguire quando la coda viene risvegliata, ed i *jiffies* relativi al timeout. Come funzione si specifica la `mutex_trylock` in modo tale che quando viene risvegliata la waitqueue ogni thread in sleep cercherà di ottenere il lock. Se un thread riesce ad ottenere il lock ritorna al chiamante e procede con la scrittura, altrimenti resta in sleep fino allo scadere del timeout. Al termine del timer il processo cerca nuovamente di ottenere il lock, e se l'operazione fallisce viene ritornato `LOCK_NOT_ACQUIRED` alla `write_on_stream`. Quindi se un processo riesce ad ottenere il lock prima dello scadere del timeout si prosegue con la scrittura, altrimenti questa non avviene e si restituisce un errore all'utente. Ogni volta che un thread rilascia il lock chiama `wake_up()` sulla *WaitQueue*, attivando di fatto tutti i thread in attesa, che proveranno a prendere il lock sul dispositivo.

4.2.2 Scrittura a bassa priorità

Sul flusso `LOW_PRIORITY` le operazioni di `write` devono avvenire in maniera asincrona. Per questo nelle scritture a bassa priorità non viene chiamata direttamente la `write_on_stream`, ma una funzione intermedia `write_low`, che si occupa di schedulare la scrittura secondo il meccanismo di **deferred work**.

Nonostante la scrittura può avvenire in un secondo momento, l'utente deve comunque essere notificando in maniera sincrona rispetto all'esito dell'operazione. Per fare ciò si verifica subito se c'è spazio sufficiente nel dispositivo, prima ancora di chiamare la `write_low`. Se la scrittura può avvenire vengono subito aggiornati i campi `available_bytes` e `total_bytes_high`.

- A prescindere di quando si andrà a scrivere effettivamente sul dispositivo, vengono *riservati logicamente* i byte necessari per la scrittura deferred.
- In questo modo le operazioni di `read`/`write` successive terranno conto dello spazio realmente disponibile sul dispositivo, anche se la scrittura avverrà in un secondo momento.
- Questo implica dire che una **scrittura deferred non può fallire**, in quanto il risultato viene mostrato all'utente prima ancora di scrivere sullo stream di dati.

Nella `write_low` viene allocata una struttura `packed_work_struct`, che permette di risalire alle informazioni sulla scrittura partendo dalla `work_struct` che verrà schedulata.

Tramite `kzalloc` viene allocato un buffer kernel temporaneo, che viene associato al campo `packed_work->data`. All'interno di questo buffer vengono copiati i byte utente che dovranno poi essere scritti sullo stream, utilizzando la `copy_from_user` .

Successivamente si utilizza la macro `__INIT_WORK` per inizializzare la `work_struct` (mantenuta nel campo `work` della struttura packed). Come secondo parametro di `__INIT_WORK` viene specificato l'indirizzo della funzione `write_deferred`. Infine, viene schedulato il lavoro appena creato, tramite la `schedule_work()`.

Quando il **kworker daemon** prende il controllo va ad eseguire la `write_deferred`. Qui partendo dalla `work_struct` si risale tramite `container_of` alla `packed_work_struct` in cui è contenuta. A questo punto si recuperano tutti i metadati riguardo la scrittura richiesta, e questi vengono passati come parametri di input alla `write_on_stream`.

Come già detto, la scrittura *low priority* non può fallire, e per questo nella funzione `try_mutex_lock` si utilizza in questo caso `mutex_lock` anziché `mutex try_lock`. Così facendo il demone incaricato della scrittura resta in attesa finché non riesce ad ottenere il lock sul dispositivo. Solo quando il lock viene acquisito si ritorna `LOCK_ACQUIRED` alla `write_on_stream`, e si procede con la scrittura di un nuovo blocco descritta in precedenza.

5 Operazioni di Lettura

L'operazione di lettura è implementata nella funzione `dev_read`, che viene invocata quando si effettua la system call `read()` a livello applicativo. La lettura avviene andando a *leggere progressivamente* i blocchi dello stream, fino a raggiungere la quantità di bytes che si vogliono leggere.

Come da specifica, quando un byte viene letto si rimuove questo dallo stream. A tale scopo il contenuto letto viene prima *rimosso logicamente*, spostando in avanti il `read_offset` del blocco, e poi *rimosso fisicamente* quando vengono completamente letti tutti i bytes di un blocco.

I blocchi vengono scansionati all'interno di un ciclo `while(1)` seguendo la politica FIFO: la lettura inizia sempre dalla testa dello stream (`flow_state->head)`, e prosegue sul blocco successivo (`stream_block->next`) fino a quando non vengono letti il numero di bytes desiderati, o non ci sono più dati da leggere nel data stream.

Per tenere traccia del progresso di lettura si utilizzano due valori interi:

- `to_read`: mantiene il numero di bytes che devono essere ancora letti dallo stream; viene decrementato ogni volta che vengono letti dati da un blocco.
- `bytes_read`: mantiene quanti byte sono stati letti dallo stream, e viene incrementato ogni volta che vengono letti dati da un blocco.

I bytes letti da un blocco vengono copiati progressivamente in un buffer utente tramite `copy_to_user`. In particolare si utilizza `bytes_read` come offset sul buffer destinazione in modo da restituire all'utente un unico risultato anche se la lettura coinvolge blocchi differenti.

5.1 Scansione e Lettura dei Blocchi

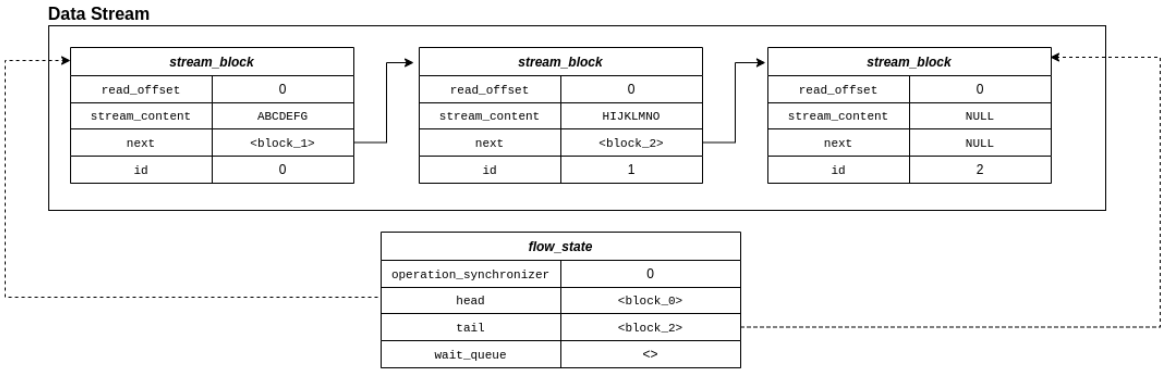
Durante ogni iterazione del `while` si considera un singolo blocco (indicato con `current_block`), e la lettura su tale blocco inizia sempre a partire dal `read_offset` corrente, tenendo conto dunque delle letture precedenti. A questo punto possono esserci tre scenari differenti:

- **Lettura blocco parziale**: la lettura richiede un numero di bytes inferiore a quelli logicamente presenti nel blocco attuale.
- **Lettura blocco completa**: la lettura richiede un numero di bytes pari o superiore alla taglia del blocco attuale.
- **Lettura del blocco in coda**: la lettura richiede un numero di bytes che scorre tutti i blocchi fino a giungere all'ultimo blocco (puntato da `flow_state->tail)`.

5.1.1 Partial Block Reading

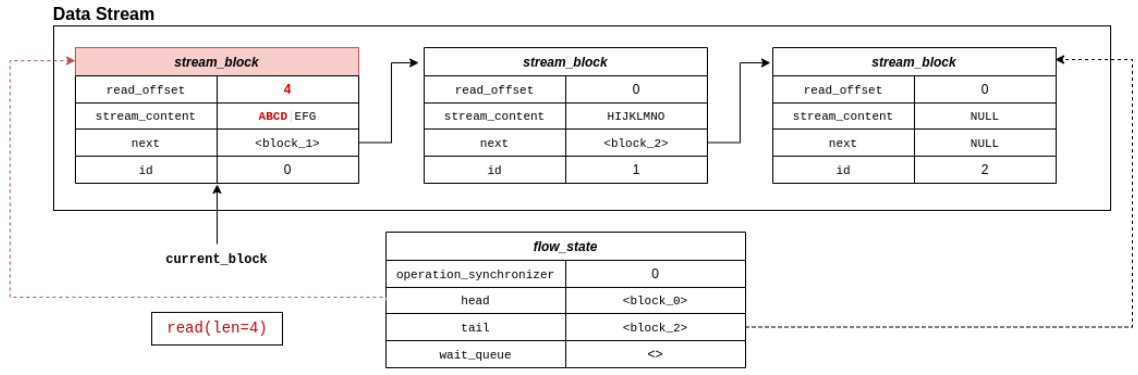
Se `to_read` è inferiore ai bytes logicamente presenti nel blocco allora, dopo aver copiato i dati letti nel buffer utente, si sposta soltanto il `read_offset` del blocco in avanti, senza rimuovere fisicamente i bytes che sono stati letti.

Consideriamo ad esempio lo stream descritto in precedenza per la `write`.



Supponiamo venga richiesta la lettura di `to_read=4 bytes`. La lettura inizia sempre dal blocco `head`, che in questo caso contiene *7 bytes* di dati. In questo caso si ha una lettura parziale del blocco, si parte quindi dal `read_offset` (*=0*) e si leggono i successivi *4 bytes*.

Vengono restituiti all'utente i bytes `ABCD`, che vengono rimossi solo logicamente dallo stream spostando in avanti `read_offset` proprio di *4 bytes*.



Poiché il *blocco 0* ha ancora dati da leggere, le successive operazioni di `read` avverranno sempre a partire dal *blocco 0*, ma con `read_offset` iniziale pari a *4*; di fatto verranno considerati quindi soltanto i bytes a partire dal quinto, ignorando `ABCD` che è sono già stati consumati da qualcuno.

5.1.2 Full Block Reading

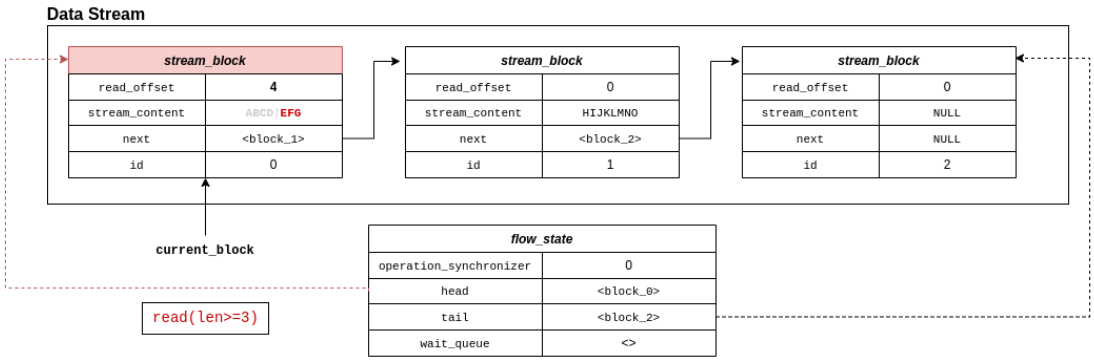
Se `to_read` è uguale o superiore ai *bytes logicamente presenti nel blocco*, allora verranno letti tutti i bytes rimanenti nel blocco. Ciò significa che dopo aver copiato i dati sul buffer utente si procede con la *rimozione fisica* del blocco dallo stream. Questo avviene in tre passaggi:

1. Si sposta in avanti la testa dello stream, facendo puntare `flow_state->head` al blocco successivo a quello letto.

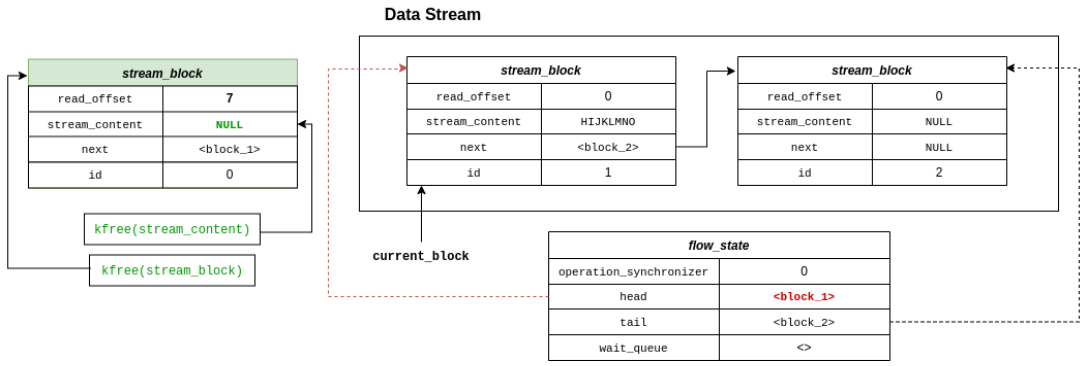
2. Viene liberato il buffer kernel che manteneva i dati del blocco (puntato dal campo `stream_content`).

3. Viene deallocata l'area di memoria che mantiene la struttura `stream_block`, completando di fatto la rimozione fisica dei dati dati dallo stream.

Vediamo schematicamente questo meccanismo, partendo dallo scenario descritto sopra. In questo caso supponiamo venga richiesta la lettura di 3 o più bytes. La lettura inizia dall'*offset 4*, e pertanto verranno consumati i rimanenti *3 bytes* `EFG` del blocco.



A questo punto il contenuto del blocco è stato letto completamente, e si può procedere con la rimozione fisica dei dati. Si scollega il blocco appena letto dal data stream, facendo puntare `flow_state->head` al blocco successivo `block_1`. Poi tramite `kfree` si libera il buffer puntato da `stream_content`, ed infine sempre tramite `kfree` si rimuove l'intero `block_0` dalla memoria.

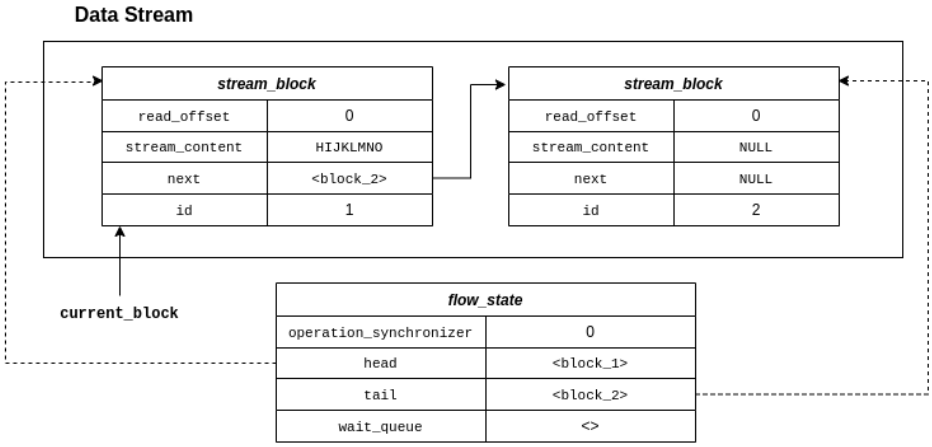


5.1.3 Tail Block Reading

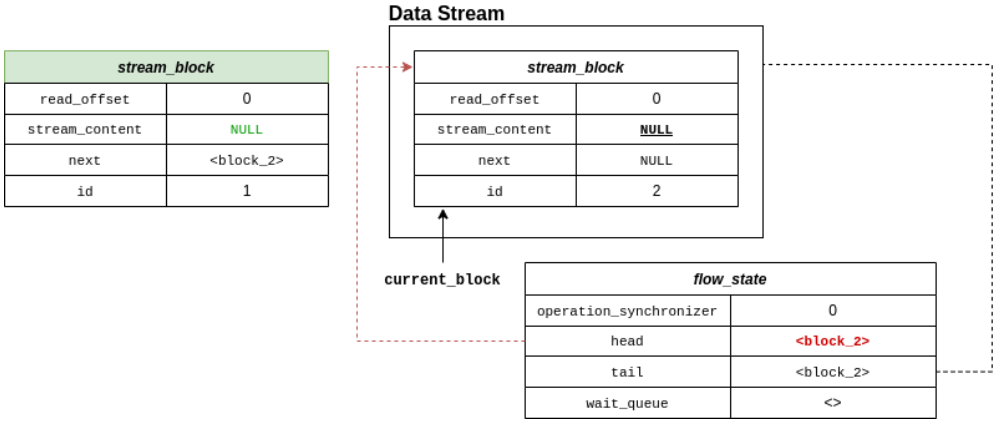
Se l'utente richiede la lettura di più byte rispetto a quelli complessivamente disponibili sul flusso, all'interno del ciclo `while` verranno consumati tutti i blocchi del data stream, di fatto andando quindi a leggere tutti i dati disponibili.

Per terminare la lettura in questo caso non si effettua un controllo sui bytes disponibili, ma si effettua un semplice controllo sul valore di `current_block->stream_content`. In particolare la `read` arriverà necessariamente a considerare l'ultimo blocco come `current_block`; questo blocco è puntato da `flow_state->tail`, e sappiamo per costruzione che ha sempre `stream_content=NULL`. Ciò vuol dire che non si prosegue con la scansione di blocchi successivi, terminando così la lettura.

Vediamo anche quest'ultima situazione sullo schema precedente.



Consideriamo una lettura di *100 bytes*, che partirà dal *blocco 1*. Verranno letti tutti gli *8 bytes* disponibili, quindi viene rimosso fisicamente il blocco, si aggiorna `to_read=100-8=92` e si passa a leggere il blocco successivo. Qui devono essere letti *92 bytes*, ma si verifica che `current_block->stream_content` è `NULL`: sono stati dunque letti tutti i bytes presenti sul flusso e l'operazione di lettura termina con successo.



5.2 Ottenimento del lock in lettura (`try_mutex_lock`)

L'operazione di lettura *non è idempotente*, in quanto va a consumare (logicamente o fisicamente) i dati presenti sullo stream. Ciò implica l'utilizzo di un meccanismo di locking anche in lettura, per garantire la consistenza tra le operazioni effettuate.

Per questo motivo all'interno della `dev_read`, prima di entrare nel ciclo `while` e scansionare i vari blocchi, si utilizza la funzione `try_mutex_lock` già descritta in precedenza. Anche in questo caso si ha un comportamento differente a seconda di come è impostata la sessione.

- **Lettura non-bloccante**: si utilizza la `mutex_trylock` in quanto il thread non dovrà attendere se il dispositivo è occupato. Quindi se un altro processo ha il lock sul dispositivo la `try_mutex_lock` ritorna `LOCK_NOT_ACQUIRED` alla `dev_read`, e di fatto la lettura non viene eseguita.
- **Lettura bloccante**: si utilizza lo stesso approccio descritto per la `write`. Se il lock non viene acquisito si mette il task nell'apposita *Wait Queue* del flusso. Il thread resterà in *sleep* finché la coda non viene risvegliata o fino allo scadere del timeout specificato.

A differenza della scrittura, l'operazione di lettura è *sincrona* sia sul flusso ad alta priorità che su quello a bassa priorità. Quindi la `try_mutex_lock` internamente utilizza sempre `mutex_trylock`, e di fatto potrà fallire anche sul flusso a bassa priorità se non ci sono dati da leggere o se non si riesce ad acquisire il lock.

6 Altre File Operations

Analizziamo ora l'implementazione delle altre file operations, che vanno a realizzare le seguenti funzionalità:

- *Inizializzazione del Modulo*
- *Cleanup del modulo*
- *Apertura di un device*
- *Chiusura di un device*
- *Gestione della Sessione di I/O*
  - Modifica della priorità
  - Modifica del tipo di operazioni
  - Impostazione del timeout

6.1 Inizializzazione del modulo

Quando si monta il modulo viene invocata la `init_module`. In questa funzione viene inizializzato lo stato di tutti i 128 dispositivi, andando a settare i campi della struttura `object_state` e dei relativi `flow_state`. Inoltre tutti i dispositivi vengono abilitati di default settando `device_enabling[i]=1`.

Successivamente si registra il *Char Device* tramite `__register_chrdev`. Si specifica `0` come major number, per cui questo verrà scelto dinamicamente dal sistema. Il Major assegnato al device viene stampato tramite `printk` sul buffer del kernel, in modo che l'utente possa poi recuperarlo per andare a lavorare sui dispositivi tramite CLI.

6.2 Cleanup del modulo

Quando il modulo viene rimosso viene chiamata la `cleanup_module`. Questa funzione va a liberare tutte le aree di memoria allocate per contenere i dati dello stream, utilizzando iterativamente `kfree()` sulle due linked list di `stream_block`. Vengono poi deallocate anche tutte le strutture `flow_state` e `object_state` utilizzate.

Infine si va a rimuovere il device driver tramite `unregister_chrdev` e si notifica tramite `printk` l'avvenuta rimozione.

6.3 Apertura e chiusura di un device

Quando lato utente si apre un dispositivo, lato kernel viene invocata la `dev_open`. In questa funzione si sfrutta la macro `get_minor` per ottenere il minor number del dispositivo aperto a partire dal puntatore all'oggetto `file`.

Successivamente se il minor è valido (compreso tra *0* e *127*), si verifica se il file è abilitato o disabilitato, controllando il valore di `device_enabling[Minor]`. Se questo vale `1` allora il dispositivo è abilitato e si prosegue con l'apertura del device, altrimenti viene ritornato un errore.

Tramite `kzalloc` si alloca la struttura `session_state` che mantiene la sessione verso il dispositivo aperto, e si inizializza questa come `HIGH_PRIORITY` e `NON_BLOCKING`, con `TIMEOUT` pari a `0 ms`. Tali parametri possono essere poi modificati tramite gli appositi comandi della CLI che operano sulla sessione.

A questo punto è stata correttamente aperta una sessione di I/O verso il device file, e si notifica tramite `printk` il successo dell'operazione.

Quando invece viene chiuso un dispositivo precedentemente aperto, deve essere chiusa anche la sessione di I/O verso quel device. Lato kernel viene invocata la `dev_release`, che tramite `kfree` va a liberare l'area di memoria precedentemente allocata per la struttura `session_state`.

6.4 Gestione della Sessione

Per modificare i parametri della sessione si utilizza lato utente l'API `ioctl()`. Si specifica come secondo parametro un comando intero, che discrimina l'operazione specifica da effettuare sulla sessione. Il terzo parametro invece viene passato con valore diverso da 0 soltanto per le operazioni di *setup del timeout*, indicandone il relativo valore in millisecondi.

Lato kernel viene invocata la `dev_ioctl`, ed in base al comando specificato viene settato l'apposito campo nella struttura `session_state`.

- **Gestione della priorità**, si modifica il campo `session->priority`.
  - `SET_LOW_PRIORITY` (`3`): Imposta la sessione per operare sul flusso a bassa priorità.
  - `SET_HIGH_PRIORITY` (`4`): Imposta la sessione per operare sul flusso ad alta priorità.
- **Gestione del tipo di operazioni**, si modifica il campo `session->blocking`.
  - `SET_BLOCKING_OP` (`5`): Specifica che le operazioni successive saranno bloccanti.
  - `SET_NON_BLOCKING_OP` (`6`): Specifica che le operazioni successive saranno non-bloccanti.
- **Impostazione del timeout**, si modifica il campo `session->timeout`.
  - `SET_TIMEOUT` (`7`): Modifica il timeout di attesa sul lock, impostandolo al valore passato come terzo parametro a `ioctl()`.
- **Gestione dello stato dei dispositivi**, permette di abilitare o disabilitare un dispositivo tramite `ioctl` invece che scrivendo direttamente sul file. Come già detto questo meccanismo non è attualmente implementato, ma i codici 8 e 9 sono comunque riservati.
  - `ENABLE_DEV` (`8`) : Abilita un dispositivo
  - `DISABLE_DEV` (`9`): Disabilita un dispositivo

7 Utilizzo del Modulo

In questa sezione viene documentata la struttura del progetto, come installare il modulo nel sistema, e come interagire con esso tramite la CLI fornita.

7.1 Organizzazione della Repository

La directory principale del progetto è `soa-project`, che mantiene al suo interno due directory `driver` e `user`.

- **`driver/`**: contiene il codice `multiflow_driver.c` del modulo e lo script `reinstall_module.sh`, che permette di compilare ed installare rapidamente il modulo.
- **`user/`**: contiene il codice `user_cli.c` e l'eseguibile `user_cli` che implementa una semplice CLI per interagire con i dispositivi del driver.

7.2 Montaggio e Rimozione del Modulo

Per installare il modulo si può eseguire lo script `driver/reinstall_module.sh`. Nello specifico questo va a smontare eventuale versioni precedenti del modulo, compila l'ultima versione, e poi la installa nel sistema.

In alternativa si possono comunque eseguire manualmente i comandi necessari, quindi `make all` per la compilazione, e `insmod multiflow_driver.ko` per l'installazione.

Quando il modulo viene montato con successo sul buffer del kernel viene stampato il **major number** assegnatogli. Questo può essere quindi recuperato dall'utente tramite il comando `dmesg`.

```
[ago27 18:35] MULTI-FLOW DEV: ------------------------------------ CLEAN ----------------------------------------
[  +0,000005] MULTI-FLOW DEV: Unregistering the device, releasing pending resources.
[  +0,001131] MULTI-FLOW DEV: Data stream memory & device metadata released.
[  +0,000002] MULTI-FLOW DEV: The device with major number 504 has been unregistered.
[  +0,000001] MULTI-FLOW DEV: ------------------------------------------------------------------------------
[  +1,648054] MULTI-FLOW DEV: ------------------------------------ INIT -----------------------------------------
[  +0,000006] MULTI-FLOW DEV: Initializing Object State.
[  +0,000021] MULTI-FLOW DEV: Object State correctly Initialized.
[  +0,000006] MULTI-FLOW DEV: New device registered, it is assigned major number 503
[  +0,000002] MULTI-FLOW DEV: ------------------------------------------------------------------------------
```

Per rimuovere il modulo si può utilizzare il comando `rmmod multiflow_driver`, mentre tramite `make clean` si possono rimuovere dalla directory `soa-project/driver` tutti i file generati in fase di compilazione.

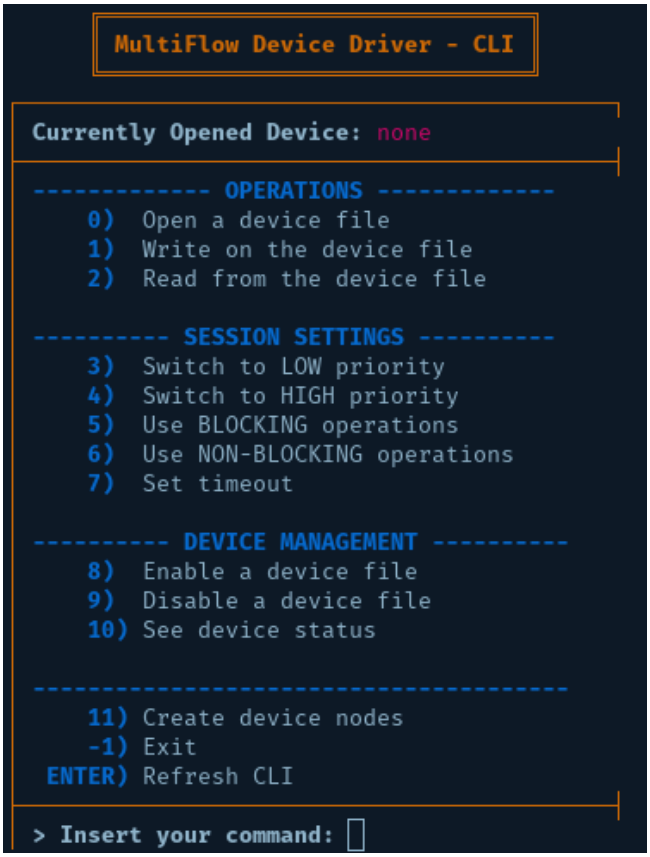
7.3 User CLI

Lanciando tramite `sudo` il programma `user/user_cli` è possibile interagire con i *multiflow devices* tramite il driver appena installato. Il programma accetta due argomenti da riga di comando:

- `major` (`argv[1]`): Major number del device installato, ottenuto tramite `dmesg`.
- `device_path` (`argv[2]`): Percorso nel VFS in cui verranno creati i *device files*. Se l'utente non specifica questo parametro viene utilizzato il path di default, ovvero `/dev/mflow-dev`.

Prima di operare con il Char Device è necessario creare i device file che rappresentano i dispositivi sul VFS. Questo può essere fatto:

- Manualmente tramite `mknod dev/nome_device MAJOR MINOR`
- Utilizzando il comando `11` *(Create device nodes)* da `user_cli`, che genera automaticamente *128* file relativi ai dispositivi che devono essere gestiti.

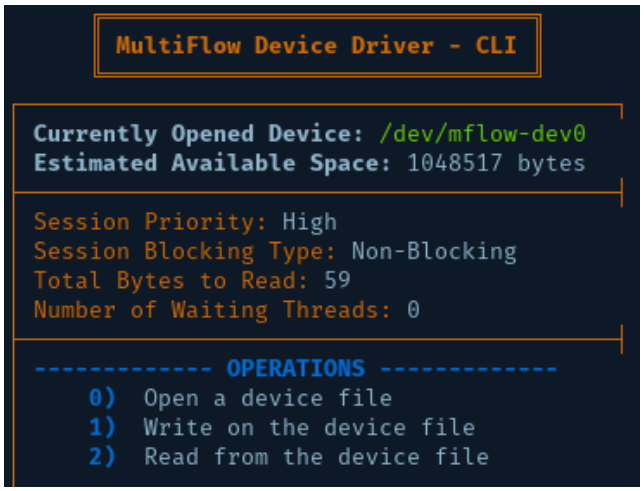


Vediamo nel dettaglio quali sono le operazioni offerte dal programma.

7.3.1 Operazioni sui device

La CLI offre le operazioni basilari per operare con un dispositivo.

- **Open a device file** (**0**): Chiede all'utente di inserire un minor number `N` e viene aperto il relativo file `/dev/mflow-devN`. Dopo aver aperto un file vengono mostrate nell'header della CLI le informazioni principali su quel device.
- **Write on the device file** (**1**): Richiede all'utente di inserire i dati da scrivere, ed effettua la scrittura sul file aperto tramite la system call `write()`.
- **Read from the device file** (**2**): Richiede all'utente la quantità di bytes che vuole leggere, ed effettua la lettura dal file aperto tramite la system call `read()`.



7.3.2 Operazioni sulla sessione

Tramite CLI è possibile modificare alcuni parametri della sessione, che vanno a definire il comportamento delle operazioni di `write`/`read`. Tutti questi comandi fanno utilizzo di `ioctl` per interagire con il modulo e modificare lo stato della sessione.

- **Switch to LOW/HIGH priority** (**3**/**4**): Modifica il parametro `priority` della sessione, cambiando quindi il flusso dati da `HIGH` a `LOW` o viceversa.
- **Use BLOCKING/NON-BLOCKING operations** (**5**/**6**): Viene modificato il parametro `blocking` della sessione, passando quindi da operazioni non-bloccanti a bloccanti e viceversa.
- **Set timeout** (**7**): Modifica il parametro `timeout` della sessione, impostando quindi il tempo di attesa per il lock nelle operazioni bloccanti.

7.3.3 Gestione dei dispositivi

Tramite VFS vengono esposti diversi parametri che rappresentano lo stato del dispositivo, che possono essere letti o manipolati direttamente dalla CLI.

- **Enable/Disable a device file** (**8**/**9**): Richiede un minor number all'utente e abilita o disabilita il dispositivo associato a quel minor. Per fare ciò scrive il valore `0` o `1` nel file `/sys/module/multiflow_driver/parameters/device_enabling`, nella posizione specifica associata al dispositivo.
- **See device status** (**10**): Visualizza tutte le informazioni sullo stato di un dispositivo, specificato dall'utente tramite il minor number. Si accede quindi in lettura ai parametri del modulo `/sys/module/multiflow_driver/parameters/`.

7.3.4 Altri comandi

La CLI oltre a quelli già descritti presenta altri tre comandi:

- **Create device nodes** (**11**): Genera 128 file nel path di default, o nel path specificato dall'utente tramite secondo argomento. A tale scopo si utilizza un semplice ciclo `for` dove viene chiamata più volte `mknod`. I file generati hanno:
  - Tutti lo stesso major number, indicato tramite il primo argomento dall'utente.
  - Minor numbers progressivi da *0* a *127*.
- **Refresh CLI** (**ENTER** o **12**): Aggiorna le informazioni mostrate nell'header della CLI, utile se più processi hanno sessioni aperte verso lo stesso device. Ad esempio si può visualizzare il nuovo spazio disponibile su un client differente da quello che ha effettuato l'ultima operazione.
- **Exit** (**-1**): Chiude il device file attualmente aperto, e termina il programma.