Progetto Sistemi Operativi Avanzati A.A. 2021/2022

Multi Flow Device Driver

Jacopo Fabi 0293870

1 Specification

La specifica è relativa ad un driver Linux che implementa flussi di dati ad alta e bassa priorità. Attraverso una sessione aperta verso il device file, un thread può leggere e scrivere segmenti dati. La consegna dei dati segue una policy First-in-First-out lungo ciascuno dei due diversi flussi di dati (bassa e alta priorità).

Dopo le operazioni di lettura, i dati letti scompaiono dal flusso. Inoltre, il flusso dati ad alta priorità deve offrire operazioni di scrittura sincrone mentre il flusso dati a bassa priorità deve offrire un'esecuzione asincrona (basata su delayed work) delle operazioni di scrittura, pur mantenendo l'interfaccia in grado di notificare in modo sincrono l'esito. Le operazioni di lettura sono tutte eseguite in maniera sincrona.

Il device driver dovrebbe supportare 128 devices corrispondenti alla stessa quantità di minor number.

Il device driver dovrebbe implementare il supporto per il servizio ioctl(..) in modo tale da gestire la sessione di I/O come segue:

- Setup del livello di priorità (alto o basso) per le operazioni;
- Operazioni di lettura e scrittura bloccanti vs non bloccanti;
- Setup del timeout che regola il risveglio delle operazioni bloccanti.

Alcuni parametri e funzioni del modulo Linux dovrebbero essere implementati in modo tale da poter abilitare o disabilitare un device file, in termini di specifico minor number. Se è disabilitato, qualsiasi tentativo di aprire una sessione dovrebbe fallire (ma sessioni già aperte verranno comunque gestite).

Ulteriori parametri esposti via VFS dovrebbero fornire un'immagine dello stato corrente del device in accordo alle seguenti informazioni:

- Abilitato o disabilitato;
- Numero di bytes correntemente presenti nei due flussi (alta e bassa priorità);
- Numero di threads correntemente in attesa di dati lungo i due flussi (alta e bassa priorità).

2 Data Structures

Di seguito vengono descritte le strutture dati implementate per rappresentare i diversi oggetti necessari alla realizzazione del modulo.

2.1 Data Segment

La struttura `data_segment_t` rappresenta un segmento di dati, ogni segmento viene creato durante una scrittura per cui mantiene i dati di quella singola operazione:

```
typedef struct data_segment {
    struct list_head entry;
    char *content;
    int byte_read;
    int size;
} data_segment_t;
```

- `entry`: oggetto `list_head` che rappresenterà un entry di una linked list.
- `content`: puntatore al contenuto del segmento di dati.
- `byte_read`: numero di bytes letti fino all'istante di tempo `t`, necessario per la gestione delle operazioni di lettura.
- `size`: dimensione del contenuto del segmento di dati.

2.2 Device Manager

Ogni dispositivo viene gestito dalla struttura `device_manager_t`, oggetto di I/O che mantiene tutte le informazioni necessarie per operare sul dispositivo ad esso associato:

```
typedef struct device_manager {
    struct workqueue_struct *workqueue;
    flow_manager_t *flow[FLOWS];
} device_manager_t;
```

- `workqueue`: puntatore alla workqueue associata al flusso di dati a bassa priorità.
  - Implementata come *singlethread workqueue*, in questo modo i deferred work sono processati su un singolo worker thread e non è quindi necessario gestire la concorrenza, eseguendo le scritture nell'ordine in cui sono state schedulate.
- `flow[FLOWS]`: puntatore ad un array di due oggetti di tipo `flow_manager_t`, ognuno dei quali si occupa della gestione di un singolo flusso del dispositivo.

Il driver supporta fino a 128 dispositivi, per questo motivo in fase di inizializzazione si riserva un'area di memoria per la variabile globale *devices*, array di 128 oggetti di tipo `device_manager_t`, così da poter gestire tutti i dispositivi richiesti.

2.3 Flow Manager

La struttura `flow_manager_t` mantiene tutte le informazioni necessarie per operare su un singolo flusso di priorità di un dispositivo:

```
typedef struct flow_manager {
    struct list_head head;
    struct mutex op_mutex;
    wait_queue_head_t waitqueue;
} flow_manager_t;
```

- `head`: oggetto di tipo `list_head` standalone che rappresenta il *flusso di dati come* una *linked list*, testa della lista che punta al primo (`head→next`) e all'ultimo segmento (`head→prev`) di dati.
- `op_mutex`: mutex per la sincronizzazione delle operazioni di lettura e scrittura sul flusso di dati, il driver supporta sessioni di I/O concorrenti sullo stesso device file ed è quindi necessario dover gestire la concorrenza.
- `waitqueue`: waitqueue del singolo flusso di priorità, necessaria per operazioni bloccanti per cui un thread viene messo in sleep e inserito nella coda di attesa fino a quando non si verificano le condizioni per riprendere l'esecuzione.

Un oggetto di tipo `flow_manager_t` gestisce un flusso di priorità, per questo motivo implementa diverse funzioni per inizializzare segmenti di dati, leggere/scrivere segmenti di dati dal/sul flusso, rilasciare le risorse allocate quando non più necessarie:

```
void init_flow_manager(flow_manager_t *);
void init_data_segment(data_segment_t *, char *, int);
void write_data_segment(flow_manager_t *, data_segment_t *);
void read_from_flow(flow_manager_t *, char *, int);
void free_data_segment(data_segment_t *);
void free_flow(flow_manager_t *);
```

2.3.1 Init Flow Manager

La funzione `init_flow_manager()` inizializza gli oggetti per gestire il flusso di dati, ovvero la testa della lista, un mutex e una waitqueue.

2.3.2 Init Data Segment

La funzione `init_data_segment()` inizializza un segmento di dati specificando il puntatore al buffer che contiene i dati e la dimensione del contenuto.

2.3.3 Write Data Segment

La funzione `write_data_segment()` aggiunge un nuovo segmento al flusso sfruttando la chiamata a `list_add_tail()`, che aggiunge un entry prima della testa della lista:

```
void write_data_segment(flow_manager_t *flow, data_segment_t *segment) {
list_add_tail(&(segment->list), &(flow→head));
}
```

2.3.4 Read From Flow

Per come avviene la scrittura sul flusso, si costruisce una *coda FIFO* dove i segmenti sono linkati tra loro nel preciso ordine di inserimento: ogni segmento punta a quello inserito immediatamente prima (`entry→prev`) e dopo (`entry→next`) di lui.

La linked list, quindi, per costruzione, ci permette di recuperare qualsiasi numero di segmenti a partire dal solo oggetto `list_head` standalone:

- Il campo `head→next` punta al `list_head` del primo segmento, il cui campo `entry→next` punterà al `list_head` del secondo segmento e così via.
- Ogni segmento può essere ottenuto tramite la funzione `list_entry()`, che riceve in input un entry della lista e ritorna l'oggetto `data_segment_t` che la incapsula.

La funzione `read_from_flow()` legge da uno o più segmenti di dati del flusso in base alla dimensione specificata, riportandoli sul buffer fornito in input.

Inizialmente viene recuperata la testa della lista per ottenere il primo segmento di dati dalla prima entry (`head→next`), e si inizializza a 0 il numero di bytes letti (`byte_read`), necessario per gestire eventuali letture parziali su un segmento:

```
void read_from_flow(flow_manager_t *flow, char *read_content, int len) {
    struct list_head *head, *cur, *old;
    data_segment_t *cur_seg;
    int byte_read;
    head = &(flow->head);
    cur = head→next;    //entry is list_head name in data_segment structure
    cur_seg = list_entry(cur, data_segment_t, entry);
    byte_read = 0;
```

A questo punto si entra in un ciclo `while()` solo se il primo segmento ha dati sufficienti, quindi se il numero di bytes richiesti (`len`) è maggiore o uguale alla taglia del segmento:

```
    while (len - byte_read >= cur_seg->size - cur_seg->byte_read) {
        memcpy(read_content + byte_read, cur_seg->content + cur_seg->byte_read,
               cur_seg->size - cur_seg->byte_read);
        byte_read += cur_seg->size - cur_seg→byte_read;
        old = cur;
        cur = cur→next;
        free_data_segment(cur_seg);
        list_del(old);
        if (cur == head) break;
        cur_seg = list_entry(cur, data_segment_t, entry);
    }
```

- Si effettua la copia dei bytes da leggere dal segmento sul buffer fornito in input e si aggiorna la variabile `byte_read`.
- Si salva il puntatore all'entry in analisi (`old`) e si aggiorna il puntatore `cur` per passare all'entry successiva (`cur→next`).

- Si rilascia il segmento di dati consumato e si elimina la corrispondente entry dalla linked list tramite la chiamata a `list_del()`.
- Se la successiva entry è il `list_head` standalone il ciclo termina, altrimenti si recupera il segmento associato e si procede con l'iterazione successiva essendoci ancora dati disponibili per la lettura.
    - Dalla seconda iterazione il ciclo tiene conto del numero di bytes consumati (`byte_read`) e di quelli letti sul segmento corrente (`cur_seg→byte_read`).
    - In questo modo è possibile gestire eventuali letture parziali su un singolo segmento di dati del flusso.

Quindi, nel ciclo si consuma un segmento alla volta in maniera iterativa finché non ci sono più dati, oppure si esce se i bytes ancora da leggere (`len - byte_read`) sono di meno di quelli disponibili su un certo segmento (`cur_seg→size - cur_seg→byte_read`).

Per gestire questa situazione di lettura parziale, al termine del ciclo è stato aggiunto un controllo per cui, se l'entry non è la testa della lista, e quindi ci sono dati da leggere, si recuperano effettivamente tutti i bytes richiesti (`len - byte_read`):

```
    if (cur != head) {
        cur_seg = list_entry(cur, data_segment_t, entry);
        memcpy(read_content + byte_read, cur_seg->content + cur_seg->byte_read,
               len - byte_read);
        cur_seg->byte_read += len - byte_read;
        byte_read += len - byte_read;
    }
}
```

2.3.5 Free Data Segment

La funzione `free_data_segment(data_segment_t *)` rilascia l'area di memoria occupata dal buffer puntato dal segmento che mantiene il suo contenuto, per poi liberare l'area effettivamente legat al segmento nella sua interezza.

2.3.6 Free Flow

La funzione `free_flow(flow_manager_t *)` rilascia le aree di memoria occupate dai diversi segmenti di dati che compongono il flusso, dal mutex e infine dal gestore stesso. Tramite il metodo `list_for_each()` si itera sulla linked list, ad ogni passo si elimina l'entry e si rilascia il segmento corrispondente, recuperato sfruttando `list_entry()`:

```
old = NULL;
list_for_each(cur, head) {
    if (old) list_del(old);
    cur_seg = list_entry(cur, data_segment_t, entry);
    free_data_segment(cur_seg);
    old = cur;
}
```

2.4 Session

Per interagire con un dispositivo è necessario aprire una sessione verso il device file, per cui parliamo di un oggetto che avrà un'unica istanza per ogni thread in esecuzione. Come da specifica, inoltre, abbiamo due livelli di priorità e due tipologie di operazioni, con le operazioni bloccanti che consentono di configurare un certo timeout di attesa.

La struttura `session_t`, quindi, mantiene le informazioni associate ad una certa sessione di I/O, ed è associata al campo `private_data` del file quando questo viene aperto:

```
typedef struct session {
    short priority;
    gfp_t flags;
    unsigned long timeout;
} session_t;
```

- `priority`: rappresenta la priorità della sessione, quindi se si va ad operare sul flusso a bassa o ad alta priorità.
- `flags`: rappresenta la tipologia di operazioni della sessione, quindi se si va ad operare in maniera bloccante (`GFP_KERNEL`) o non bloccante (`GFP_ATOMIC`).
- `timeout`: rappresenta il timeout di attesa nelle operazioni bloccanti.

2.5 File Operations

La struttura `file_operations` mantiene i puntatori alle funzioni definite dal driver, che eseguono diverse operazioni su un dispositivo: ogni campo della struttura corrisponde all'indirizzo di una certa funzione definita dal driver per gestire un'operazione richiesta.

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = device_open,
    .release = device_release,
    .unlocked_ioctl = device_ioctl,
    .write = device_write,
    .read = device_read
};
```

2.6 Asynchronous Task

Come già anticipato, le scritture sul flusso a bassa priorità vengono eseguite in maniera asincrona sfruttando le *work queues* per realizzare il cosiddetto *deferred work:* il codice viene schedulato per essere eseguito in un secondo momento.

A tale scopo è stata implementata la struttura `async_task`, caratterizzata dal job da schedulare e da altre informazioni per completare la scrittura e notificare l'esito al client:

```
typedef struct async_task {
    struct delayed_work del_work;
    data_segment_t *to_write;
    session_t *session;
    int minor;
} async_task_t;
```

- `del_work`: struttura di tipo `delayed_work` che schedula un job su una workqueue per essere eseguito almeno dopo un certo intervallo di tempo prefissato.
    - A differenza di un oggetto di tipo `work_struct`, permette di specificare un attesa minima prima dell'esecuzione del job, ritardando l'aggiunta in coda.
- `to_write`: puntatore al segmento di dati da scrivere.
- `session`: puntatore ad una struttura di tipo `session_t` utilizzata per salvare le informazioni relative alla sessione verso il device file.
- `minor`: minor number associato al dispositivo su cui avviene la scrittura.

3 Module Parameters

Il driver deve esporre diversi parametri tramite VFS per fornire una vista dello stato corrente di un certo dispositivo, oltre che per abilitare o disabilitare un device file:

- `enabled`: array di `bool` dove l'*i-esimo* elemento indica se il *device i* è abilitato oppure disabilitato.
    - Un dispositivo disabilitato impedisce l'apertura di sessioni verso il rispettivo device file, permettendo comunque a sessioni già aperte di poter lavorare.
- `bytes_in_buffer`: array di `long` dove l'elemento *i-esimo* indica il numero di bytes leggibili dal *device i* sul flusso a bassa priorità, mentre l'elemento *128+i* indica il numero di bytes leggibili dal *device i* sul flusso ad alta priorità.
    - I valori sono aggiornati ogni volta che vengono completate letture e scritture.
- `threads_in_wait`: array di `long` dove l'elemento *i-esimo* indica il numero di threads in attesa sul *device i* sul flusso a bassa priorità, mentre l'elemento *128+i* indica il numero di threads in attesa sul *device i* sul flusso ad alta priorità.
    - I valori sono aggiornati ogni volta che un thread si mette in attesa di un lock oppure riesce ad acquisirlo.

Per entrambi i parametri `bytes_in_buffer` e `threads_in_wait` si utilizza una singola array per i due livelli di priorità così da ottenere un vantaggio in termini di prestazioni. Evitiamo l'uso di due array distante per i due flussi, e quindi il controllo da codice su quale utilizzare in base alla priorità, sfruttando una semplice operazione aritmetica:

- `(PRIORITY * MINOR_NUMBER) + i;`
- `Priority low = 0` accediamo ai campi da 0 a 127 per priorità bassa;
- `Priority high = 1` accediamo ai campi da 128 a 255 per priorità alta.

3.1 Permissions

Ogni parametro del modulo è riportato su un corrispondente file nella directory *sys/module/multi_flow_device_driver/parameters*, e sono creati con permessi differenti per gestirli in maniera corretta come richiesto dalla specifica:

- `enabled` viene creato con i permessi di lettura e scrittura abilitati per l'utente ed il gruppo proprietario (`S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP`) potendo abilitare o disabilitare i dispositivi.
- `bytes_in_buffer` e `threads_in_wait` vengono creati con i permessi di lettura per l'utente ed il gruppo proprietario (`S_IRUSR | S_IRGRP`), questo perché i valori possono essere letti ma la modifica diretta non è consentita venendo aggiornati in modo automatico dal driver tramite operazioni di lettura e scrittura.

3.2 Interactions

Per abilitare o disabilitare un device file, è possibile sfruttare l'interazione diretta con il parametro esposto nel VFS.

Un dispositivo può essere *abilitato/disabilitato* tramite lo script `enable_device.bash`:

- Si copia il file `/sys/module/multi_flow_device_driver/parameters/enabled`, caratterizzato da 128 valori binari Y/N, sul file temporaneo `/tmp/to_delete`.
- Si splitta il file temporaneo tramite il comando *awk* e viene settato a Y/N il valore che corrisponde al minor number specificato come primo parametro.
- Il file originale viene sostituito e quello temporaneo rimosso.

Un dispositivo può essere *interrogato* in due modi differenti:

- Tramite lo script `query_device.bash`, si specifica il minor number del dispositivo e sfruttando il comando *cut* i files dei diversi parametri vengono tokenizzati per poter recuperare i valori del dispositivo di interesse.
- Tramite il comando `make`, in questo caso è possibile accedere ai parametri nella loro interezza non potendo specificare il minor number, avendo quindi una vista globale dello stato dei diversi dispositivi.
  - I comandi a disposizione sono `show-devices`, s`how-hp-bytes`, `show-lp-bytes`, `show-hp-threads`, `show-lp-threads`.
  - Si utilizza il comando *cat* per leggere i files dei parametri, recuperando per i bytes leggibili e i threads in attesa la sola metà di array relativa alla priorità di interesse.

4 Module Operations

Di seguito vengono descritte le funzioni di inizializzazione e rilascio del modulo, oltre alle operazioni implementate dal driver per operare sui diversi dispositivi.

4.1 Initialization

Quando viene montato il modulo si invoca la funzione `init_module()`, al cui interno viene registrato un *char device driver* sfruttando la chiamata a `__register_chrdev()`:

- Il major number viene impostato a 0 così che l'allocazione venga gestita in maniera automatica dal sistema.
- Il minor number di partenza viene impostato a 0 mentre il totale dei dispositivi a 128, così da poter gestire tanti device files come richiesto dalla specifica.

A questo punto, viene effettuato il setup delle strutture di tipo `device_manager_t`, ognuna delle quali si occupa, come già visto, della gestione di un singolo dispositivo:

```
for (i = 0; i < MINOR_NUMBER; i++) {
    devices[i].workqueue = create_singlethread_workqueue("work-queue-" + i);
    devices[i].flow[LOW_PRIORITY] = kmalloc(sizeof(flow_manager_t), GFP_KERNEL);
    devices[i].flow[HIGH_PRIORITY] = kmalloc(sizeof(flow_manager_t), GFP_KERNEL);
    init_flow_manager(devices[i].flow[LOW_PRIORITY]);
    init_flow_manager(devices[i].flow[HIGH_PRIORITY]);
}
```

- Istanziamo una *singlethread workqueue* per ogni dispositivo per schedulare i task sul flusso a bassa priorità.
- Istanziamo ed inizializziamo due oggetti `flow_manager_t` per ogni dispositivo, ognuno dei quali si occupa della gestione di un singolo flusso di priorità.

Alla fine, in caso di errori di allocazione avviene il rilascio di tutte le aree di memoria allocate e la rimozione del driver, altrimenti il major number assegnato viene stampato sul buffer del kernel così da poterlo recuperare in due modi diversi:

- Tramite il comando `dmesg` che visualizza il buffer del kernel su stdout.
- Tramite il comando `cut` tokenizzando la riga in *proc/devices* che fa riferimento al driver (`cat /proc/devices | grep multi-flow | cut -d " " -f 1` come nello script `create_devices.bash`).

4.2 Cleanup

Quando viene rimosso il modulo si invoca la funzione `cleanup_module()`, al cui interno viene rimosso il driver sfruttando la chiamata a `unregister_chrdev()` e rilasciate tutte le aree di memoria allocate per gestire i dispositivi e salvare i segmenti di dati.

4.3 Device Open

Lato utente, l'apertura di un dispositivo equivale alla system call `open()`, che si traduce a livello kernel nella creazione di un oggetto di tipo `file` per rappresentare il dispositivo aperto, e nell'invocazione della funzione `device_open()`.

Un oggetto di tipo `file` mantiene le informazioni necessarie a lavorare con la sessione aperta verso il device, ed è passato dal kernel a tutte le funzioni che operano su di esso.

Nella `device_open()`, il driver recupera il minor number associato al dispositivo aperto sfruttando la macro `get_minor`, che utilizza il puntatore all'oggetto `file`.

Se il minor number fornito alla macro è valido, si verifica se il device è abilitato tramite il controllo del valore corrispondente nell'array `enabled`, solo a questo punto l'apertura del dispositivo può proseguire, altrimenti fallisce e viene restituito un errore.

Il driver alloca ed inizializza un oggetto di tipo `session_t`, che rappresenta la precisa configurazione della sessione, associata poi al campo `private_data` offerto dal kernel per puntare a dati allocati utili così da preservare lo stato tra le diverse system calls:

```
static int device_open(struct inode *inode, struct file *filp) {
    session_t *session;
    int minor = get_minor(filp);
    if (minor < 0 && minor >= MINOR_NUMBER) return -ENODEV;
    if (!enabled[minor]) return -EBUSY;
    session = kmalloc(sizeof(session_t), GFP_KERNEL);
    if (session == NULL) {
        pr_info("Failure on session_t allocation\n");
        return -1;
    }
    session->priority = HIGH_PRIORITY;
    session->flags = GFP_KERNEL;
    session->timeout = MAX_SECONDS;
    filp->private_data = session;
    pr_info("Session opened for minor: %d\n", minor);
    return 0;
}
```

La sessione di I/O verso il dispositivo è stata aperta correttamente, la configurazione iniziale è predisposta per eseguire le operazioni sul flusso ad alta priorità e in maniera bloccante, con un timeout di default pari a `MAX_SECONDS`: il client può poi modificare la configurazione della sessione tramite la funzione `ioctl()`, che permette di manipolare i parametri invocando a livello kernel la `device_ioctl()`.

4.4 Device Release

Lato utente, la chiusura di un dispositivo equivale alla system call `close()`, che si traduce a livello kernel nell'invocazione della funzione `device_release()`: il driver

rilascia l'area di memoria allocata per la struttura `session_t` e ripristina a `NULL` il campo `private_data`, è il kernel che poi si occupa del rilascio della struttura `file`.

4.5 Device I/O Control

Lato utente, è possibile modificare i parametri di un sessione tramite la system call `ioctl()`, che si traduce a livello kernel nell'invocazione della funzione `device_ioctl()`.

La funzione `ioctl()` riceve in input il file descriptor relativo alla sessione di I/O, un comando intero che identifica l'operazione da apportare sulla sessione, ed un terzo parametro opzionale per operazioni che richiedono il setup di un certo timeout.

La funzione `device_ioctl()` è implementata come uno *switch-case* che va a modificare un certo campo della struttura `session_t` in base al comando specificato, ovviamente per fare questo è necessario costruire un mapping tra i comandi lato utente e lato kernel:

- `PRIORITY`, si modifica il campo `session→priority`
  - `set_high_priority(3)`: lato kernel corrisponde al caso `TO_HIGH_PRIORITY`, per cui si imposta la sessione per lavorare sul flusso ad alta priorità.
  - `set_low_priority(4)`: lato kernel corrisponde al caso `TO_LOW_PRIORITY`, per cui si imposta la sessione per lavorare sul flusso a bassa priorità.
- `OPERATION`, si modifica il campo `session→flags`
  - `set_blocking_operations(5)`: lato kernel corrisponde al caso `BLOCKING`, per cui si imposta la sessione per effettuare le operazioni in maniera bloccante.
  - `set_unblocking_operations(6)`: lato kernel corrisponde al caso `UNBLOCKING`, per cui si imposta la sessione per effettuare le operazioni in maniera non bloccante.
- `TIMEOUT`, si modifica il campo `session→timeout`
  - `set_timeout(7, value)`: lato kernel corrisponde al caso `TIMEOUT`, per cui si imposta nella sessione il timeout al valore inserito per le operazioni bloccanti.
- `DEVICE STATE`, si modifica l'array `enabled` così da poter abilitare o disabilitare un dispositivo tramite `ioctl()` senza dover modificare direttamente il parametro esposto sul VFS
  - `enable_device(8)`: lato kernel corrisponde al caso `ENABLE`, per cui si imposta a true il valore nell'array relativo al minor number del dispositivo.
  - `disable_device(9)`: lato kernel corrisponde al caso `DISABLE`, per cui si imposta a false il valore nell'array relativo al minor number del dispositivo.

4.6 Init Operation

La funzione `init_operation()` è stata progettata con l'obiettivo di inizializzare una operazione di tipo bloccante o non su un certo dispositivo, ed è quindi utilizzata sia all'interno dell'operazione di lettura, `device_read()`, che di scrittura, `device_write()`.

Il metodo sfrutta la macro `is_blocking(flags)`, per cui si distinguono due diversi flussi di esecuzione in base al tipo di operazione offerta dalla sessione:

- `Operazione bloccante`, se una risorsa non può essere ottenuta, il thread viene sospeso finché la condizione non si verifica.
    - Per le *scritture*, il thread viene sospeso finché non è in grado di ottenere il lock sul mutex e lo spazio per scrivere non è sufficiente.
    - Per le *letture*, il thread viene sospeso finché non è in grado di ottenere il lock sul mutex e c'è almeno un bytes disponibile sul flusso.
- `Operazione non bloccante`, il thread viene rimosso se l'operazione non può essere eseguita appena richiesta.
    - Per le *scritture*, il thread viene immediatamente rimosso nel caso in cui non riesce ad acquisire il lock o non c'è spazio sufficiente per scrivere.
    - Per le *letture*, il thread viene immediatamente rimosso nel caso in cui non riesce ad acquisire il lock o non c'è neanche un bytes disponibile sul flusso.

Il metodo ritorna il valore 1 in caso di successo per notificare la corretta inizializzazione, ovvero lock acquisito e condizione di lettura/scrittura verificata, in modo tale da poter procedere con l'effettiva operazione.

4.6.1 Blocking

Tramite la macro `inc_thread_in_wait()` si incrementa il numero di threads in attesa su un flusso di priorità di un certo dispositivo, aggiornando il corrispondente valore nell'array `threads_in_wait`.

Tramite la macro `wait_event_interruptible_exclusive_timeout()` il thread viene sospeso e inserito nella coda corrispondente finché la condizione specificata non si verifica, oppure scade il timeout settato per l'operazione bloccante:

- La condizione viene controllata ogni volta che si sveglia la waitqueue, ed utilizza la macro `lock_and_awake()` in modo diverso per lettura e scrittura.

Tramite la macro `dec_thread_in_wait()` si decrementa il numero di threads in attesa su un flusso di priorità di un certo dispositivo aggiornando il corrispondente valore nell'array `threads_in_wait`.

L'inizializzazione va a buon fine solamente se la condizione specificata nell'attesa si verifica alla scadenza del timeout o prima, altrimenti l'operazione non può proseguire.

Custom Wait

Quando un thread viene messo in attesa si aggiunge un entry nella waitqueue, nel caso di attesa non esclusiva verrà inserita in testa altrimenti alla fine della coda:

- Al risveglio della waitqueue il kernel sveglia tutti i threads in attesa non esclusiva ed uno soltanto, in maniera ordinata, di quelli in attesa esclusiva.
- Non ha senso far svegliare contemporaneamente tutti i threads in attesa sapendo che solo uno potrà acquisire il lock per procedere, soprattutto in presenza di un numero molto alto di threads che porta al fenomeno del *thundering herd*.

L'idea è quindi quella di mettere tutti i threads in attesa esclusiva, così che il kernel svegli sempre e solo il primo thread in coda, rispettando l'ordine di arrivo in waitqueue.

La macro `wait_event_interruptible_exclusive_timeout()` è implementata allo stesso modo della macro già esistente `wait_event_interruptible_timeout()`, ma è stato necessario definirla per abilitare l'attesa in modalità esclusiva:

```
#define wait_event_interruptible_exclusive_timeout(wq_head, condition, timeout) ({
    long __ret = timeout;
    might_sleep();
    if (!___wait_cond_timeout(condition)) __ret =
         __wait_event_interruptible_exclusive_timeout(wq_head, condition, timeout);
    __ret; })
```

Per schedulare i tasks nelle corrispondenti code in maniera esclusiva, la *custom wait* invoca al suo interno la macro `__wait_event_interruptible_exclusive_timeout()` al posto di quella già esistente `__wait_event_exclusive_timeout()`:

```
#define __wait_event_interruptible_exclusive_timeout(wq_head, condition, timeout)
    ___wait_event(wq_head, ___wait_cond_timeout(condition), TASK_INTERRUPTIBLE, 1,
                  timeout, __ret = schedule_timeout(__ret))
```

- La nuova macro invoca anch'essa `__wait_event()`, con la differenza che il parametro di input `exclusive` è settato ad 1 anziché a 0, inserendo quindi il tthraed nella waitqueue con il flag `WQ_FLAG_EXCLUSIVE`.

Lock and Awake

La macro `lock_and_awake()` è stata implementata per essere usata come condizione di `wait_event_interruptible_exclusive_timeout()`, in particolare restituisce un esito positivo se si riesce ad acquisire il lock e la condizione di lettura/scrittura è verificata:

```
#define lock_and_awake(condition, mutex) ({
    int __ret = 0;
    if (mutex_trylock(mutex)) {
        if (condition) __ret = 1;
        else mutex_unlock(mutex); }
    __ret; })
```

4.6.2 Non-blocking

Tramite `mutex_trylock()` si cerca di acquisire il lock sul mutex, se non è possibile viene immediatamente ritornato un errore, altrimenti si procede con il controllo della condizione di lettura/scrittura utilizzando:

- La macro `is_empty(priority, minor)` per verificare se ci sono dati disponibili per la lettura sul flusso di dati.
- La macro `is_free(priority, minor)` per verificare se c'è spazio disponibile per la scrittura sul flusso di dati.

Se anche la condizione è verificata l'operazione può proseguire, altrimenti si risveglia la waitqueue e si rilascia il lock così che un altro thread possa riprendere l'esecuzione.

4.7 Device Write

Lato utente, l'operazione di scrittura su un dispositivo equivale alla system call `write()`, che si traduce a livello kernel nell'invocazione della funzione `device_write()`.

Come già detto, per procedere con una scrittura è necessario ottenere il lock sul mutex associato al device e che ci sia effettivamente spazio libero per aggiungere dati al flusso.

Inizialmente, il modulo verifica se il numero di bytes da scrivere (`len`) non è valido, in questo caso la scrittura termina immediatamente, altrimenti si procede allocando le diverse aree di memoria necessarie per il completamento dell'operazione:

```
if (len <= 0) return 0;
tmp_buf = kmalloc(len, session->flags);
if (tmp_buf == NULL) {
    pr_info("Failure on char* allocation\n");
    return -1;
}
byte_not_copied = copy_from_user(tmp_buf, buff, len);
to_write = kmalloc(sizeof(data_segment_t), session->flags);
if (to_write == NULL) {
    pr_info("Failure on data_segment_t allocation\n");
    return -1;
}
res = init_operation(flow, session, minor, "write");
if (!res) goto free_area;
```

- Si alloca un buffer temporaneo per copiare a livello kernel i bytes da scrivere a partire da un buffer a livello user, sfruttando la funzione `copy_from_user()`.
- Si alloca un segmento di dati che dovrà essere aggiunto al flusso.
- Si invoca la funzione `init_operation()` per inizializzare l'operazione bloccante o non, che va a buon fine solo se si acquisisce il lock e c'è spazio per scrivere, altrimenti si salta ad una label per rilasciare le risorse e annullare la scrittura.

Il modulo prosegue effettuando dei controlli per impostare il numero di bytes da scrivere (`len`) al valore corretto, tenendo conto di eventuali bytes non copiati dal livello utente e dello spazio a disposizione sul flusso di dati:

```
if (len - byte_not_copied > free_space(session->priority, minor))
          len = free_space(session->priority, minor);
else len = len - byte_not_copied;
init_data_segment(to_write, tmp_buf, len);
pr_info("Start effective write.\n");
```

- Impostato tramite la macro `free_space(priority, minor)` nel caso in cui il valore `len - byte_not_copied` è più grande dello spazio disponibile sul flusso.

- Impostato a `len - byte_not_copied` se lo spazio disponibile può accogliere tutti i bytes portati dal livello utente al livello kernel.
- Si procede inizializzando il segmento di dati `to_write`, che punterà al buffer `tmp_buf` che mantiene i dati da scrivere per una dimensione pari a `len`.

A questo punto, può avere inizio l'effettiva operazione di scrittura, che avviene in modo sincrono o asincrono in base alla priorità della specifica sessione legata all'operazione.

4.7.1 High Priority

La scrittura sul flusso ad alta priorità viene realizzata aggiungendo un segmento alla fine del flusso tramite il metodo `write_data_segment()`, a questo punto si aggiorna tramite la macro `add_to_buffer()` il corrispondente valore nell'array `bytes_in_buffer`, per poi risvegliare la waitqueue e rilasciare il lock in favore di un altro thread:

```
if (session->priority == HIGH_PRIORITY) {
    pr_info("The selected operation is required at high priority.\n");
    write_data_segment(flow, to_write);
    add_to_buffer(HIGH_PRIORITY, minor, len);
    wake_up_interruptible(&(flow->waitqueue));
    mutex_unlock(&(device->flow[session->priority]->op_mutex));
    pr_info("Operation completed, bytes writed to the device
             at high priority: %s\n", tmp_buf);
}
```

4.7.2 Low Priority

La scrittura a bassa priorità deve seguire un'esecuzione asincrona basata sul lavoro ritardato (*delayed work*), a tale scopo si implementa il meccanismo del `deferred work`: permette di schedulare del codice per essere eseguito dopo un certo tempo.

Il modulo, a questo punto, alloca una struttura di tipo `async_task_t`, implementata per mantenere il job da schedulare e le informazioni per completare la scrittura differita:

```
else {
    pr_info("The selected operation is required at low priority.\n");
    task = kmalloc(sizeof(async_task_t), session→flags);
    if (task == NULL) {
        pr_info("Failure on async_task_t allocation\n");
        return -1; }
    task->to_write = to_write;
    task->session = session;
    task->minor = minor;
```

- Il job è un oggetto di tipo `delayed_work`, che a differenza di un `work_struct` consente di specificare un tempo minimo di attesa prima di schedularlo.
- Si associa al job l'indirizzo del segmento da scrivere, il minor del dispositivo su cui effettuare l'operazione e le informazioni sulla sessione verso il device file.

Per definizione, un deferred work *non può fallire*, per questo deve essere schedulato solo se tutte le strutture necessarie all'esecuzione vengono allocate correttamente: nel nostro caso, il `data_segment` da scrivere e il buffer temporaneo che mantiene i dati usato per inizializzare il segmento, già allocati in precedenza.

Si può quindi procedere inizializzando il job tramite `INIT_DELAYED_WORK()`, per poi schedularlo tramite la chiamata `queue_delayed_work()`, che permette di specificare una workqueue diversa da quella di default utilizzata a livello kernel:

```
    pr_info("Insert thread in the workqueue...\n");
    INIT_DELAYED_WORK(&(task->del_work), (void *)async_write);
    add_to_buffer(LOW_PRIORITY, minor, len);
    queue_delayed_work(device->workqueue, &(task->del_work),
          msecs_to_jiffies(5000));   //job inserted in queue after 5sec delay
```

- In questo modo, possiamo sfruttare una *singlethread workqueue* per le scritture sul flusso a bassa priorità, avendo la garanzia che le scritture vengono eseguite nell'ordine in cui sono state schedulate sulla singola coda associata al flusso.

Prima di schedulare il job si aggiorna il numero di bytes presenti sul flusso in maniera sincrona così da riservare logicamente lo spazio necessario per la scrittura, i dati saranno poi scritti effettivamente quando il job entra in esecuzione, job che non può fallire.

Async Write

La funzione `INIT_DELAYED_WORK()` inizializza un job, in particolare il secondo parametro rappresenta l'indirizzo della funzione che verrà eseguita dal job quando entrerà effettivamente in esecuzione.

Tramite `container_of()` si recupera l'indirizzo della struttura `async_task_t` associata al job in esecuzione, sfruttando quindi l'indirizzo del relativo campo `delayed_work` passato in input alla funzione di scrittura asincrona:

```
void async_write(struct delayed_work *data) {
    async_task_t *task = container_of((void*)data, async_task_t, del_work);
    device_manager_t *device = devices + task->minor;
    flow_manager_t *flow = device->flow[LOW_PRIORITY];
```

Recuperate tutte le informazioni necessarie, la scrittura avviene come un'esecuzione sincrona, con la differenza che è necessario acquisire il lock prima di poter procedere:

```
inc_thread_in_wait(task->session->priority, task->minor);
mutex_lock(&(flow->op_mutex));
dec_thread_in_wait(task->session->priority, task->minor);
write_data_segment(device->flow[LOW_PRIORITY], task->to_write);
pr_info("Operation completed, bytes writed to the device
         at low priority: %s\n", task->to_write->content);
kfree(task);
mutex_unlock(&(flow->op_mutex));
wake_up_interruptible(&(device->flow[LOW_PRIORITY]->waitqueue));
```

4.7.3 Result

In entrambi i casi di scrittura sincrona e asincrona, l'operazione di `write()` restituisce all'utente il numero di bytes scritti: ciò che cambia, per un'esecuzione asincrona, è che i dati saranno effettivamente scritti quando il job schedulato entrerà in esecuzione, ma poiché un deferred work non può fallire, è corretto che l'utente riceva subito la risposta.

4.8 Device Read

Lato utente, l'operazione di lettura da un dispositivo equivale alla system call `read()`, che si traduce a livello kernel nell'invocazione della funzione `device_read()`.

Come già detto, per procedere con una lettura è necessario ottenere il lock sul mutex associato al device e che ci siano effettivamente dati da leggere dal flusso.

Inizialmente, il modulo verifica se il numero di bytes da leggere (`len`) non è valido, in questo caso la lettura termina immediatamente, altrimenti si procede con l'allocazione di un buffer temporaneo che ospiterà i dati letti dal flusso:

```
if (len <= 0) return 0;
tmp_buf = kmalloc(len, session->flags);
if (tmp_buf == NULL) {
    pr_info("Failure on char* allocation\n");
    return -1;
}
res = init_operation(flow, session, minor, "read");
if (!res) goto free_area;
if(len > byte_to_read(session->priority,minor))
        len = byte_to_read(session->priority, minor);
pr_info("Start effective read.\n");
```

- Si invoca la funzione `init_operation()` per inizializzare l'operazione bloccante o non, che va a buon fine solo se si acquisisce il lock e ci sono dati da leggere, altrimenti si salta ad una label per rilasciare le risorse e annullare la lettura.
- Si controlla se è possibile leggere dal flusso il numero di bytes richiesti (`len`), altrimenti si aggiorna il valore così da leggere solamente quelli disponibili.

A questo punto, può avere inizio l'effettiva operazione di lettura, che avviene in maniera sincrona su tutti e due i flussi di priorità, per cui si distingue un solo comportamento.

La lettura dal flusso viene realizzata tramite il metodo `read_from_flow()`, a questo punto si aggiorna tramite la macro `sub_to_buffer()` il corrispondente valore nell'array `bytes_in_buffer`, per poi risvegliare la coda e rilasciare il lock:

```
read_from_flow(flow, tmp_buf, len);
sub_to_buffer(session->priority,minor,len);
wake_up_interruptible(&(flow->waitqueue));
mutex_unlock(&(flow->op_mutex));
res = copy_to_user(buff,tmp_buf,len);
valid = len - res;
final = kmalloc(valid, session->flags);
if (final == NULL) {
    pr_info("Failure on char* allocation\n");
    return -1;
}
memcpy(final, tmp_buf, valid);
pr_info("Operation completed, bytes readed from the device: %zu, %s\n", valid, final);
```

- Si procede tramite la funzione `copy_to_user()` per copiare a livello utente i bytes letti a partire da un buffer a livello kernel.
- Si tiene conto dei bytes non recuperati dal livello kernel per allocare un buffer che ospiterà i dati letti, per poi effettuare realmente la copia in memoria.
- Si ritorna all'utente il numero di bytes realmente letti dal livello kernel (`valid`).

5 Module Usage

Il progetto è organizzato in tre directory differenti:

- `driver`, contiene il codice del modulo.
- `user`, contiene il codice dell'applicazione utente per interagire con i dispositivi gestiti dal driver.
- `scripts`, contiene degli scripts per la creazione e l'abilitazione dei dispositivi, oltre che per interrogare un certo device di cui si vogliono conoscere i parametri.

5.1 Installation

L'installazione del modulo viene realizzata dallo script `driver/install.bash`, al suo interno si effettua la compilazione ed il montaggio del modulo, gestendo anche il caso in cui il modulo sia già presente: viene rimosso insieme ai devices e nuovamente compilato per essere re-installato.

In alternativa, è comunque possibile effettuare la compilazione e l'installazione tramite i comandi manuali `make all` e `insmod multi_flow_device_driver.ko`, rimuovendo poi con `make clean` tutti i files prodotti dalla compilazione del modulo.

Quando l'installazione del modulo va a buon fine, l'esito viene riportato sul buffer del kernel e il driver viene registrato in `/proc/devices`, per cui il major number assegnato al driver può essere recuperato tramite:

- Il comando `dmesg`.
- `cat /proc/devices | grep multi_flow_device_driver | cut -d " " -f 1`.

La rimozione del modulo viene effettuata tramite `rmmod multi_flow_device_driver.ko`.

5.2 Devices Creation

La creazione dei dispositivi viene realizzata dallo script `scripts/create_devices.bash`, si specifica come parametro il numero di device files da creare e se questo è valido, cioè compreso tra 1 e 128, si creano i files in maniera iterativa sfruttando il comando `mknod`:

```
base="/dev/multi_flow_device_"
major=$(cat /proc/devices | grep multi-flow | cut -d " " -f 1)
for (( minor=0; minor<$1; minor++ ))
do
    device="$base$minor"
    sudo mknod -m 666 $device c $major $minor
done
```

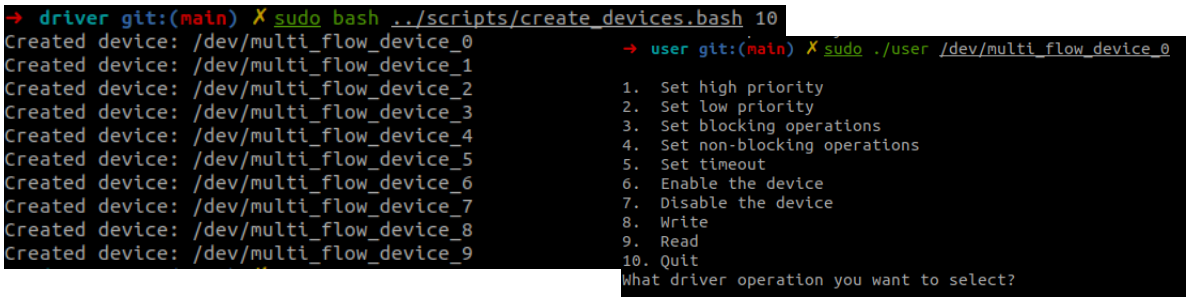
- Il path dei dispositivi creati è dato dal percorso `/dev/multi_flow_device_*`, al nome viene affiancato il corrispondente minor number.

Per rimuovere tutti i dispositivi, si fa uso del comando `rm dev/multi_flow_device_*`.

5.3 User CLI

L'interfaccia a riga di comando offerta all'utente viene eseguita tramite il programma `user/user`, che permette ad un client di interagire con dispositivi multi-flusso sfruttando il driver installato.

Il programma richiede in input un solo argomento da riga di comando, che corrisponde al percorso sul VFS di un certo device file esistente con il quale si vuole interagire:



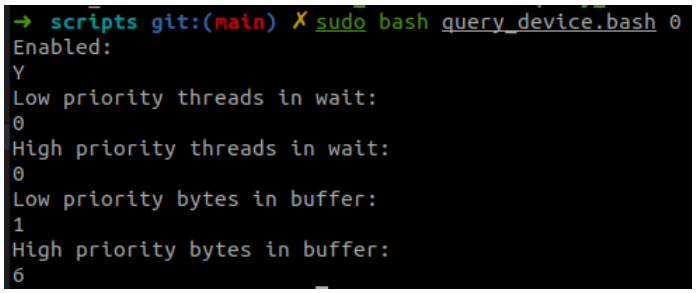
Se il percorso fornito in input è valido, viene aperta una sessione verso il device file, solo a questo punto la CLI mette a disposizione dell'utente le diverse operazioni possibili su un certo dispositivo, come presentato nella specifica:

- `SWITCH PRIORITY TYPE`, modifica il parametro `priority` della sessione per poter lavorare sui due diversi flussi di dati disponibili.
- `SWITCH OPERATIONS TYPE`, modifica il parametro `flags` della sessione per poter lavorare sui flussi di dati tramite operazioni bloccanti o meno.
- `SET TIMEOUT`, modifica il parametro `timeout` della sessione per impostare il tempo massimo di attesa per prendere il lock sul flusso nelle operazioni bloccanti.
- `SWITCH DEVICE STATUS`, modifica il parametro `enabled` del modulo per abilitare o disabilitare il dispositivo in uso.
- `WRITE`, effettua la scrittura dei dati inseriti dall'utente su un preciso flusso legato al dispositivo in uso tramite la syscall `write()`.
    - Nella scrittura asincrona, una volta schedulato un deferred work non può fallire, motivo per cui il client riceverà subito la risposta, mentre i bytes effettivi saranno scritti quando il job entrerà in esecuzione.
- `READ`, effettua la lettura del numero di bytes specificati dall'utente da un preciso flusso legato al dispositivo in uso tramite la syscall `read()`.
- `QUIT`, effettua la chiusura della sessione verso il dispositivo in uso tramite la syscall `close()` e termina l'esecuzione del programma.

I comandi che permettono la modifica di parametri della sessione e del modulo utilizzano invece l'API di `ioctl`, offerta proprio per supportare operazioni non definite dal driver.

5.3.1 Device Query

Per interrogare un dispositivo e recuperare tutte le informazioni sul suo stato corrente, un utente può utilizzare lo script `scripts/query_device.bash` specificando il minor number del device file:



In alternativa, è possibile avere una vista globale sullo stato di tutti i dispositivi utilizzando i comandi definiti all'interno del *Makefile* del modulo:

- `make show-devices`
- `make show-hp-bytes`
- `make show-lp-bytes`
- `make show-hp-threads`
- `make show-lp-threads`