Relazione del progetto di Sistemi Operativi Avanzati

# Multi-flow Device File

Simone Tiberi (M. 0299908)

(email: simone.tiberi.98@gmail.com)

12 marzo 2022

## 1 Specifica del progetto (traduzione)

La specifica richiede l'implementazione di un device driver Linux per la gestione di flussi di dati a due livelli di priorità. Attraverso una sessione aperta verso un dispositivo, un thread può leggere/scrivere dati in modo tale che:

- la consegna dei dati segua una politica **FIFO** (**F**irst-**I**n-**F**irst-**O**ut),
- non appena letti, i dati **scompaiano** dal flusso,
- la scrittura sul flusso ad alta priorità sia **sincrona**,
- la scrittura sul flusso a bassa priorità sia **asincrona**, basata su deferred work, mantenendo comunque la sincronia nella notifica dell'esito dell'operazione (in conformità all'interfaccia della `write`),
- la scrittura sia sempre eseguita in modo sincrono,
- il driver supporti al più **128** devices associati al corrispettivo minor number.

Il device driver deve implementare il supporto all'operazione `ioctl`, al fine di gestire le sessioni di I/O e permettere:

- di impostare il livello di priorità (`HIGH` or `LOW`) per le operazioni,
- di scegliere se effettuare letture e/o scritture in modo bloccante o meno,
- di impostare un timeout per regolare il risveglio in caso di richieste bloccanti.

Inoltre è richiesto di implementare un meccanismo per abilitare o disabilitare i dispositivi in termini di minor numbers, basato sui parametri del modulo. Nel caso in cui un dispositivo

sia disabilitato, ogni tentativo di apertura di nuova sessione deve fallire, ma devono comunque continuare ad essere gestite quelle aperte in precedenza.

Altri parametri addizionali esposti tramite VFS devono fornire un'immagine dello stato corrente dei device in termini di:

- abilitazione,
- numero di byte correntemente presenti nei due flussi,
- numero di thread correntemente in attesa nei due flussi.

## 2 Struttura del repository

La realizzazione della specifica è stata organizzata, all'interno del repository, nelle seguenti cartelle:

- nella radice è presente uno script `bash` per la creazione di nodi di I/O pilotabili con il driver sviluppato,
- in `KERNEL_MODULE/` è contenuto il codice effettivo del driver,
- in `USER_LIB/` è contenuta una libreria utente per facilitare l'interfacciamento con i nodi di I/O associati al driver sviluppato,
- in `SAMPLES/` sono presenti una semplice demo e un file contenente diversi test cases.

## 3 Driver per la gestione dei dispositivi multi-flusso

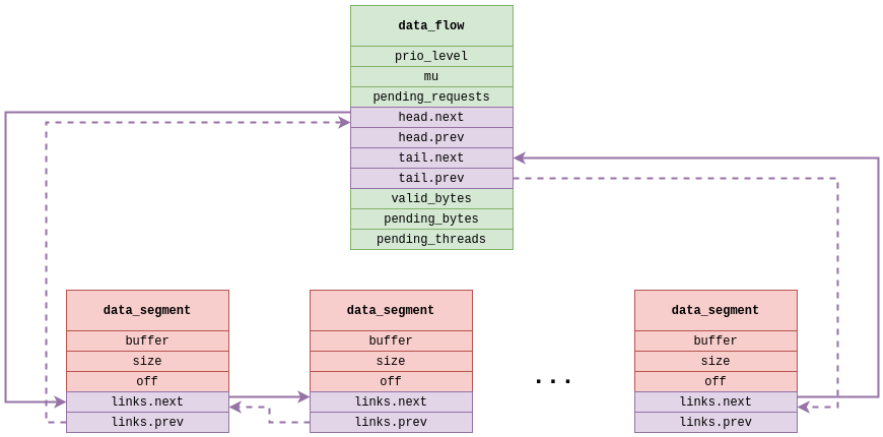


*Figura 1: Gestione dei dati all'interno dei vari devices*

In fig. 1 è riportato uno schema dell'architettura adottata, all'interno della soluzione proposta, per la gestione dei dati. Una delle strutture adottate, per la realizzazione dell'implementazione, è la `data_flow`, la quale rappresenta il singolo flusso di dati associato ad un device.

In essa sono presenti:

- il livello di priorità (`HIGH_PRIO` o `LOW_PRIO`) a cui il flusso è associato,
- il mutex per garantire l'atomicità nell'aggiornamento delle strutture dati necessarie alla gestione del flusso di dati,
- la wait queue dove i thread possono attendere nel caso in cui le loro richieste bloccanti non siano momentaneamente soddisfacibili,
- la testa e la coda di una lista collegata per l'effettiva memorizzazione dei dati, il cui meccanismo di collegamento è esemplificato in fig. 1,
- il numero di byte *validi*, ovvero scritti ma non ancora letti,
- il numero di byte *pendenti*, ovvero accettati come deferred work, ma non ancora effettivamente scritti nel flusso,
- il numero di thread in attesa della disponibilità di dati da leggere o di spazio utilizzabile in scrittura.

Un ulteriore struttura presente in fig. 1 è la `data_segment`, la quale rappresenta il chunk di dati scritto sul flusso da una singola operazione di scrittura. Per questo motivo al suo interno sono presenti:

- il buffer effettivo in cui memorizzare i dati scritti,
- la taglia complessiva dei dati scritti dall'operazione,
- l'offset a cui iniziare a leggere i dati,
- una coppia di pointers (`struct list_head`) per il collegamento con gli altri chunks.

Dalla specifica si evince come ogni device $i$ (con $0 \leq i \leq 127$) debba supportare due flussi di dati associati a priorità distinte e che la selezione del livello su cui effettuare le operazioni sia **per sessione**, per cui in fig. 2 è riportato lo schema della soluzione scelta per implementare questa meccanica.

In particolare in essa sono compaiono altre due nuove strutture:

- la `device_state`, in cui sono contenuti i metadati associati al dispositivo, quali:
  - il vettore dei due flussi di dati,
  - la work queue in cui accodare le scritture `LOW_PRIO` che debbono essere gestite in modo deferred;

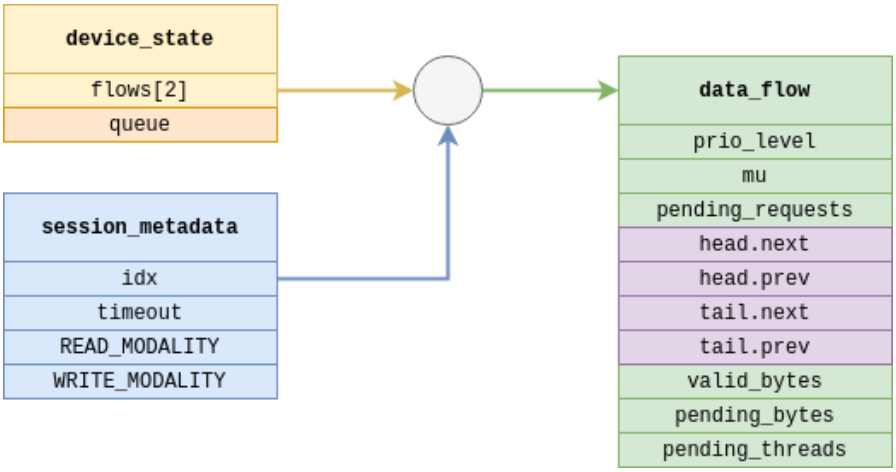


*Figura 2: Come reperire il flusso attualmente attivo a partire dalla sessione*

- la `session_metadata` che raccoglie al suo interno il set minimale di informazioni necessarie per la gestione della sessione aperta verso un device, ovvero:
  - l'indice necessario al fine di spiazzarsi all'interno dell'array di flussi contenuto nella struttura `device_state`,
  - il timeout da utilizzare nel caso in cui si deve attendere per dati da leggere o spazio utilizzabile in scrittura,
  - le modalità, bloccanti o meno, di lettura e scrittura.

È opportuno osservare che tutti i campi della struttura `session_metadata` vengono acceduti atomicamente, o perché definiti come `atomic[_long]_t`, o perché sono singoli bit.

In definitiva dunque l'iter per reperire il riferimento al flusso corrente adottato in tutta l'implementazione, attraverso la macro `get_active_flow`, è il seguente:

1. si accede all'array globale `struct device_state devs[MINORS]` utilizzando il minor number come indice. Questo viene reperito in modalità differenti a seconda della versione del kernel Linux in esercizio, sfruttando una macro apposita;
2. si preleva dalla sessione l'indice di priorità attualmente configurato e lo si usa per spiazzarsi all'interno dell'array `flows`.

## 3.1 File operations

### 3.1.1 Apertura della sessione (`open`)

La `mfdf_open`, ovvero l'implementazione all'interno del driver dell'operazione di apertura è molto semplice e si limita a:

- verificare se il dispositivo è correntemente abilitato, ed in caso contrario restituire il codice d'errore `EAGAIN` al chiamante,
- allocare, mediante `SLAB` allocator, una struttura `session_metadata` descritta poc'anzi e collegarla alla `struct file` sfruttando l'apposito pointer generico `private_data`,
- inizializzare i campi della sessione a valori di default (e.g. flusso attivo: `LOW_PRIO`).

### 3.1.2 Rilascio della sessione (`release`)

La `mfdf_release`, ovvero l'implementazione all'interno del driver dell'operazione di rilascio, essendo duale dell'apertura, si limita a liberare il buffer allocato per la memorizzazione dei metadati relativi alla sessione.

### 3.1.3 Scrittura (`write`)

Nella `mfdf_write`, ovvero l'implementazione all'interno del driver dell'operazione di scrittura, per prima cosa si *pre-alloca* la struttura `data_segment` in cui scrivere i dati richiesti, per minimizzare la dimensione della sezione critica. È opportuno osservare che in funzione della modalità di scrittura selezionata nella sessione vengono impostati i flag per la richiesta di memoria (`GFP_ATOMIC` o `GFP_KERNEL`).

Si passa poi ad eseguire un blocco di codice responsabile di mandare eventualmente *a dormire* i thread che effettuano richieste bloccanti non soddisfacibili. L'iter seguito è il seguente:

1. si incrementa atomicamente il numero di thread pendenti,
2. si passa allo stato `TASK_INTERRUPTIBLE` tramite la `wait_event_interruptible_timeout`. Il controllo per il risveglio viene effettuato invocando una funzione che:
   (a) tenta di prendere il lock,
   (b) effettua il controllo,
   (c) in caso positivo ritorna `TRUE` **senza rilasciare il lock**[1] altrimenti rilascia il lock e torna `FALSE`
3. si decrementa atomicamente il numero di thread pendenti,
4. eventualmente si ritorna al chiamante restituendo come codici d'errore:
   - `ETIME` nel caso in cui la richiesta non è stata soddisfatta nel timeout settato,
   - `EINTR` nel caso in cui il thread è stato colpito da segnale.

Dopodiché si effettua realmente la scrittura:

[1] Questo garantisce che una volta ripreso il controllo, in caso di successo dell'API nessuno può aver modificato lo stato del flusso.

- nel caso `HIGH_PRIO`, semplicemente invocando la funzione responsabile di effettuare il collegamento delle strutture dati come mostrato in fig. 1,
- nel caso `LOW_PRIO` schedulando il deferred work. A tal proposito in fig. 3 è riportata la relazione che sussiste fra le strutture in gioco.

A livello implementativo, è stato scelto di utilizzare una work queue privata *single threaded* per garantire l'ordinamento FIFO. Questo perché la linearizzazione temporale delle scritture del buffer avviene prendendo come punto di riferimento l'istante di inserimento in coda (i.e. non è possibile che se il thread $T_1$ inserisce in coda una scrittura **prima** del thread $T_2$, questa venga effettivamente scritta sul flusso **prima** della precedente).

L'allocazione delle code *single threaded* è stata realizzata in modo differente a seconda della versione del kernel Linux correntemente in esercizio:

- tramite la `alloc_ordered_workqueue` dalla versione `2.6.36`, ovvero con l'avvento delle **CMWQ** (**C**oncurrency **M**anaged **W**ork**Q**ueue),
- tramite la `create_single_threaded_workqueue` viceversa

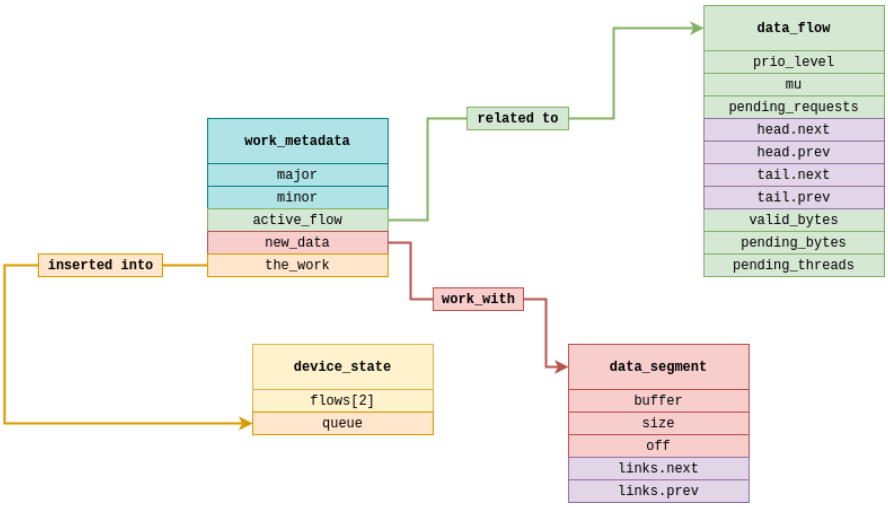


*Figura 3: Relazione delle strutture coinvolte nel meccanismo del deferred work*

Relativamente al deferred work, è opportuno analizzare l'uso del campo `pending_bytes`. L'idea è quella di garantire che il valore restituito all'utente in modo sincrono sia effettivamente pari al numero di bytes scritti nel flusso in modo deferred.

Per garantire ciò ogni qualvolta si verifica l'effettiva disponibilità di spazio scrivibile, nel caso `LOW_PRIO` si tiene conto anche dei byte che sono stati già presi in carico, ma non sono ancora stati scritti, ovvero i `pending_bytes`.

Infine, prima di ritornare al chiamante, si invoca una `wake_up_interruptible` per risvegliare i thread in attesa. La scelta implementativa è stata quella di adottare una coda singola **per**

**flusso**, in cui dunque confluiscono sia lettori che scrittori. Analizzando i possibili scenari è facile accorgersi che nella maggior parte dei casi in coda o vi sono soltanto readers o soltanto writers.

L'unica alternativa a ciò, si verifica in casi *transienti* come la timeline di seguito riportata:

1. il flusso è saturo,
2. arriva una richiesta di scrittura bloccante da parte del thread $X$ che lo porta nello stato di `TASK_INTERRUPTIBLE`,
3. arriva una richiesta di lettura da parte del thread $Y$ di un numero di byte pari alla taglia massima del flusso di dati,
4. prima che lo scrittore venga risvegliato, arriva un'ulteriore richiesta di lettura da parte del thread $Z$, che lo porta nello stato di `TASK_INTERRUPTIBLE`.

#### 3.1.4 Lettura (`read`)

Nella `mfdf_read`, ovvero l'implementazione all'interno del driver dell'operazione di lettura, per prima cosa si entra all'interno del blocco di codice responsabile eventualmente di *mandare a dormire* il thread richiedente, del tutto equivalente a quello descritto per la `mfdf_write`.

Si invoca poi una funzione responsabile di realizzare l'effettiva lettura dei dati dalla lista di strutture `data_segment` collegate al flusso.

L'idea in questo caso è quella di leggere a partire da `head->next`, ovvero il primo `data_segment` associato al flusso, fintantoché non si arriva alla `tail` e, ad ogni iterazione:

1. computare i byte effettivamente leggibili come $\min\{\Delta_{tot}, \Delta_{segment}\}$, dove:
   - $\Delta_{tot}$ corrisponde alla differenza tra quanto è stato richiesto di leggere e quanto è stato già letto,
   - $\Delta_{segment}$ corrisponde alla differenza tra la taglia complessiva dei dati contenuti in *segment* e quanti byte sono stati già letti da esso per via di letture precedenti;
2. copiare i byte all'interno del buffer specificato dall'utente, aggiornando il numero totale di byte letti e l'offset del segmento corrente,
3. nel caso in cui siano stati letti tutti i byte nel segmento, procedere a scollegarlo dalla lista, facendo dunque puntare la testa all'elemento successivo.

Infine si risvegliano i thread in coda prima di restituire il numero di byte letti al chiamante.

#### 3.1.5 Configurazione della sessione (`ioctl`)

La `mfdf_ioctl` è caratterizzata da un prototipo condizionale all'interno dell'implementazione. Questo perché dalla versione `2.6.35` la `ioctl` è stata rimossa in favore di due alternative:

- ~~`compat_ioctl`~~ per la compatibilità con i programmi utente a 32 bit,
- `unlocked_ioctl` che, a differenza della *vecchia* `ioctl`, **non prende il BKL** (**B**ig **K**ernel **L**ock).

A livello di prototipo, la versione `unlocked` e quella tradizionale differiscono per la presenza di un pointer all'`inode` nella firma, che comunque non viene utilizzato. Per questo motivo l'uso del costrutto `#ifdef...#endif` è limitato alla sola firma.

A livello implementativo, la funzione presenta uno switch-case in cui, in base al `cmd` passato, vengono impostati i diversi campi della struttura `session_metadata`, previa aver sanitizzato l'`arg` passato.

Per evitare di utilizzare come `cmd` codici *well-known* è stata adottata la macro `_IOW`, definita all'interno dell'header file `linux/ioctl.h`, la quale genera l'operational code a partire da:

- un *magic number* unico in tutto il kernel,
- un codice effettivo unico all'interno del driver.

## 3.2 Parametri del modulo

All'interno del VFS `sys` sono esposte diverse informazioni associate all'operatività del driver:

- il valore del major number, utile per lo script di generazione di nodi e per la fase di testing (read only),
- l'array dei flag di abilitazione dei devices (read/write),
- il numero di byte standing nei due flussi per ogni dispositivo (read only),
- il numero di byte standing threads nei due flussi per ogni dispositivo (read only).

In particolare, per evitare che il major number venga modificato in maniera illegittima (e.g. cambiando manualmente i privilegi associati al file), anziché adottare la macro `module_param` è stata utilizzata la versione con il callback esplicito, andando a porre come operazione `set NULL`.

Per quanto riguarda invece gli ultimi due punti dell'elenco precedente, la strategia adottata è stata differente. Anziché estrarre dalle strutture dati i campi ed utilizzare array globali, è stato preferito utilizzare `kobjects` e `kobj_attribute`, per esporre informazioni intrinsecamente read only e mantenere dunque i vari metadati raggruppati in modo semanticamente significativo.

Per cui è stato creato un nuovo kernel object `mfdf` all'interno della directory `/sys/kernel`, a cui sono stati associati due attributi, ovvero `standing_bytes` e `standing_threads`.

In definitiva dunque in tabella 1 è riportata una sintesi di *dove* e *come* consultare le informazioni esposte tramite `sys`.

| Informazione | Pseudofile | Permessi |
|---|---|---|
| Abilitazione dispositivi | `/sys/module/mfdf/parameters/enable` | 0660 |
| Major ottenuto dalla registrazione | `/sys/module/mfdf/parameters/major` | 0440 |
| Byte standing per dispositivo | `/sys/kernel/mfdf/standing_bytes` | 0440 |
| Thread standing per dispositivo | `/sys/kernel/mfdf/standing_threads` | 0440 |

*Tabella 1: Informazioni esposte in* `sys`

### 3.3 Installazione del modulo

Per installare il modulo kernel è possibile utilizzare le regole specificate all'interno del `Makefile`:

- `$ make` compila il modulo, utilizzando la regola `make modules` di `kbuild`,
- `# make install` monta il modulo, tramite `insmod`, ed installa l'header file necessario per l'utilizzo dell'`ioctl` *user-level* in `/usr/local/include/mfdf`,
- `$ make clean` rimuove l'output della compilazione dalla directory locale,
- `# make clean-all` in più al semplice `clean`:
  - smonta il modulo, se questo è effettivamente montato, sfruttando un costrutto condizionale per evitare errori,
  - rimuove l'header file `/usr/local/include/mfdf/ioctl.h`.

## 4 Libreria utente

La libreria utente è costituita da una serie di ridefinizioni, tramite direttive `#define`, di alcune funzioni tipiche delle librerie `fcntl.h` e `unistd.h` tipiche per la gestione di oggetti di I/O (e.g. `open`, `write`, ...).

L'unica funzione effettivamente realizzata *ex-novo* è un wrapper della write, per poter formattare il *cosa* scrivere secondo una regola `printf`-like.

Relativamente al Makefile presente nella directory, tramite la regola:

- `$ make` è possibile compilare la libreria utente e generare:
  - il file oggetto (`user.o`),
  - l'archivio utilizzabile dal linker (`libmfdfuser.a`);
- `$ make doc` è possibile convertire il file markdown contenente il manuale della libreria utente nel formato delle manpages;
- `# make install` è possibile:

  - installare l'header file, contenente le direttive `#define` ed il prototipo della funzione `printf`-like sopra citate, all'interno della directory `/usr/local/include/mfdf`,
  - installare la libreria statica `libmfdfuser.a` all'interno della directory `/usr/local/lib/mfdf`,
  - comprimere, utilizzando `gunzip`, la manpage generata dalla regola `doc`,
  - installare la manpage compressa all'interno della directory `/usr/local/share/man/man0p`;
- `$ make clean` è possibile rimuovere dalla directory locale ogni output prodotto dalle regole descritte in precedenza;
- `# make clean-all` è possibile rimuovere anche gli output installati nelle directory globali.

In definitiva dunque, per maggiori informazioni relativamente all'utilizzo della libreria è possibile:

- installarla, utilizzando il comando `$ make && make doc && sudo make install`,
- consultare la pagina di manuale utilizzando il comando `$ man mfdf_user.h`

Un esempio di utilizzo della libreria utente è riportato nel listato 1, compilabile utilizzando come flags:

- `-L/usr/local/lib/mfdf` per aggiungere la directory a quelle ispezionate di default per individuare le libreria da linkare,
- `-lmfdfuser` per linkare effettivamente la libreria.

# 5 Testing del modulo

Per testare il corretto funzionamento del modulo è stato utilizzato il file `SAMPLES/src/test-cases.c`. L'idea è stata quella di definire una struttura che rappresentasse il caso di test, ovvero la `struct test_case` i cui campi sono:

- il nome del test, utile per l'attività di auditing,
- il function pointer della routine di test, del tipo `void T(int, int)`.

Per cui all'interno del `main` non si fa altro che:

1. reperire il minor number, leggendolo tramite funzione apposita da `/sys`,
2. invocare, per ogni entry dell'array `test_cases` (list: 2), la funzione `do_test` responsabile:
   - dell'inizializzazione dell'ambiente,
   - del cleanup dell'ambiente,

- della stampa dei risultati.

In compilazione, è possibile abilitare o meno la modalità `VERBOSE` al fine di mostrare, per ciascun caso di test, *expected* e *actual values*, mediante il comando `$ make MORE_FLAGS=-DVERBOSE`.

# A Codici citati all'interno del report

```c
#include <stdio.h>      // for printf
#include <string.h>     // for memset
#include <unistd.h>     // for unlink
#include <errno.h>      // for errno
#include <mfdf/user.h> // for mfdf_printf/read

#include "common.h"     // for init_test_environment and get_major_number

#define DEMO_DEV "/dev/demo-mfdf"

int main()
{
        char buff[4096];
        int ret, fd, major;

        memset(buff, 0x0, 4096);
        // get major number from /sys
        major = get_major_number();


        // creates a node that can be driven by the driver
        // and opens it via mfdf_open
        if((fd = init_test_environment(DEMO_DEV, major, 0)) == -1) {
                fprintf(stderr, "Opening error (errcode: %d)\n", errno);
                return 1;
        }

        // 'printf-like' writes a presentation string
        // to the previously selected stream (default: LOW)
        if (mfdf_printf(fd, "Hi there, I'm using a multi flow device file with file descriptor no. %d", fd) == -1) {
                fprintf(stderr, "Write error (errcode: %d)\n", errno);
                return 1;
        }

        // it reads what is written on the same flow in which
        // the writing took place previously
        if ((ret = mfdf_read(fd, buff, 4096)) == -1) {
                fprintf(stderr, "Read error (errcode: %d)\n", errno);
                return 1;
        }

        printf("This is what I read: \"%s\" (len: %d)\n", buff, ret);

        // delete the node used for the demo
        unlink(DEMO_DEV);
        return 0;
}
```

*Listato 1: File: demo.c*

```
/* This array MUST be NULL terminated */
static struct test_case test_cases[] = {
        {"Blocking read with no data (LOW)",         test_blocking_read_no_data_low},
        {"Blocking read with no data (HIGH)",        test_blocking_read_no_data_high},
        {"Non-blocking read with no data (LOW)",     test_non_blocking_read_no_data_low},
        {"Non-blocking read with no data (HIGH)",    test_non_blocking_read_no_data_high},
        {"Blocking write with no space (LOW)",       test_blocking_write_no_space_low},
        {"Blocking write with no space (HIGH)",      test_blocking_write_no_space_high},
        {"Non-blocking write with no space (LOW)",   test_non_blocking_write_no_space_low},
        {"Non-blocking write with no space (HIGH)", test_non_blocking_write_no_space_high},
        {"Write less byte than read ones (LOW)",     test_write_less_read_more_low},
        {"Write less byte than read ones (HIGH)",    test_write_less_read_more_high},
        {"Standing bytes (LOW)",                     test_standing_bytes_low},
        {"Standing bytes (HIGH)",                    test_standing_bytes_high},
        {"Standing threads (LOW)",                   test_standing_threads_low},
        {"Standing threads (HIGH)",                  test_standing_threads_high},
        {"Subsequent writes on low priority",        test_subsequent_low_writes},
        {"Immutable major from /sys pseudo file",    test_immutable_major_from_sys},
        {"Read a segment in two steps (LOW)",        test_read_part_of_segment_low},
        {"Read a segment in two steps (HIGH)",       test_read_part_of_segment_high},
        {"Open a disabled device",                   test_open_not_enabled},
        {NULL, NULL}
};
```

*Listato 2: Test cases realizzati*