

Relazione Progetto SOA

Cristiano Cuffaro

matricola: 0299838

`cristiano.cuffaro@outlook.com`

A.A. 2021-2022

Indice

| | | |
|----------|--|----------|
| 1 | Rappresentazione dei device file | 2 |
| 2 | Operazioni sui multi-flow device file | 3 |
| 2.1 | Operazione di apertura | 3 |
| 2.2 | Operazione di rilascio | 4 |
| 2.3 | Operazione di lettura | 4 |
| 2.4 | Operazione di scrittura | 5 |

1 Rappresentazione dei device file

In accordo alla specifica del progetto e ai casi d'uso che si possono derivare da essa, è stata scelta una rappresentazione in RAM per i *multi-flow device file*.

In particolare, uno stream di dati è implementato come una *linked list* di segmenti di dati dinamicamente allocati e agganciati tra una *head* ed una *tail* fittizie, come mostrato nella rappresentazione in figura 1.

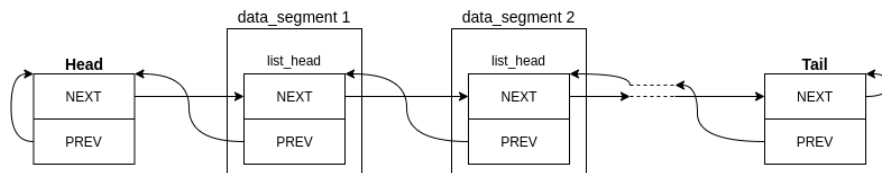


Figura 1: Lista di segmenti che implementa un data stream

Per ciascun segmento di dati, all'interno di una struttura `data_segment` sono mantenute le seguenti informazioni:

- un puntatore ai dati effettivi;
- la dimensione corrente dei dati;
- un indice che rappresenta la posizione corrente di lettura;
- una struttura `list_head` per il collegamento alla lista.

Quando è necessario allocare un nuovo segmento di dati, avviene un'unica chiamata verso lo *SLAB allocator* per ottenere un'area di memoria in cui impaccare la struttura dati nella parte alta e i dati effettivi in quella bassa. Inoltre, per limitare la dimensione reale della struttura a 40 byte è stata definita specificando l'attributo `packed`.

In una struttura `device_struct` sono definite le informazioni associate a ciascun device, in particolare:

- una waitqueue per ciascun data stream, ove si tiene traccia dei thread in attesa di prelevare o scaricare dati nel caso di letture o scritture bloccanti;
- una workqueue per mantenere il lavoro da eseguire in maniera *delayed* nel caso di scritture a bassa priorità;
- un mutex per ciascun data stream, necessario per sincronizzare le operazioni di lettura e scrittura;
- una struttura `segment_list` per ciascun data stream, contenente gli elementi fittizi della lista;
- il numero di byte validi in ciascun data stream, ovvero quelli effettivamente leggibili;

- il numero di byte liberi in ciascun data stream, ovvero quelli che determinano la possibilità di eseguire delle operazioni di scrittura;
- il numero di thread in attesa di dati per ciascun data stream, ovvero la cardinalità dell'insieme dei lettori all'interno della corrispondente waitqueue.

Per aumentare il livello di sicurezza del VFS, per questa struttura dati è stata utilizzata la dicitura `__randomize_layout`, in modo tale che un eventuale attacco finalizzato a prelevare informazioni da un device file sia ostacolato dal fatto che lo spiazamento a cui sono presenti i dati è arbitrario e cambia tra i vari kernel.

Nella funzione di inizializzazione del modulo non vengono create workqueue standard, bensì *singlethread workqueue*. Questo fa sì che il deferred work sia processato lungo un unico *kworker daemon*, che riporta le operazioni di scrittura sul data stream di bassa priorità nello stesso ordine con cui sono state schedulate, senza necessità di meccanismi di ordinamento o sincronizzazione ulteriori.

2 Operazioni sui multi-flow device file

Nelle seguenti sottosezioni sono descritte le funzioni che costituiscono il driver, ovvero che permettono di eseguire le varie operazioni sui device file che esso gestisce.

2.1 Operazione di apertura

La funzione `mfd_open` consente l'apertura di un multi-flow device file. Ad ogni invocazione, essa verifica che il minor number associato al dispositivo sia gestibile dal driver e che esso non sia correntemente disabilitato. Se i controlli vanno a buon fine, viene allocata la memoria per ospitare una struttura `session_data` contenente le seguenti informazioni:

- il livello di priorità dello stream di dati su cui si sta correntemente lavorando (bassa priorità per default);
- l'intervallo di tempo che determina la massima attesa per le operazioni di lettura e scrittura bloccanti (10 secondi per default).

Per mantenere il puntatore a questa struttura viene sfruttato il campo `.private_data` della struttura `file` corrispondente alla sessione di I/O.

Per determinare se l'operazione di apertura è stata eseguita in modalità non bloccante, viene controllato se all'interno del campo `file->f_flags` è settato il flag `O_NONBLOCK`. In caso di operazioni non bloccanti, tutte le allocazioni di memoria vengono eseguite specificando la maschera `GFP_ATOMIC`, grazie alla quale la chiamata verso l'allocatore di memoria può fallire senza portare il thread a dormire qualora non vi fosse memoria immediatamente disponibile.

2.2 Operazione di rilascio

La funzione `mfd_release` è invocata quando viene rilasciata la struttura `file` e, in maniera duale rispetto alla `mfd_open`, si occupa di riconsegnare all'allocatore l'area di memoria puntata da `file->private_data`.

2.3 Operazione di lettura

La funzione `mfd_read` viene invocata quando si esegue un'operazione di lettura su un multi-flow device file.

Se per la sessione di I/O è settato il flag `O_NONBLOCK`, viene fatto quanto segue:

1. si esegue una `trylock` sul mutex relativo allo stream di dati da cui si vuole leggere;
2. si verifica se la quantità di dati leggibili è pari a zero;

se fallisce l'acquisizione del mutex o non ci sono dati da leggere, la funzione restituisce `-EAGAIN`.

Se l'operazione di lettura è eseguita in modalità bloccante, si procede come segue:

1. viene incrementato il numero di thread in attesa di dati provenienti dal data stream in questione;
2. viene invocata l'API `wait_event_interruptible_timeout` specificando:
 - (a) la waitqueue corrispondente al data stream;
 - (b) la condizione che determina il risveglio dei thread;
 - (c) il timeout presente tra i dati di sessione;
3. se il thread esce dal sonno per via del termine del timeout o per una segnalazione, la funzione restituisce l'errore corrispondente.

Per far sì che ogni *wake up* porti al risveglio di al più un thread posto sulla waitqueue, invece di specificare direttamente alla wait queue API la condizione che il numero di byte leggibili sia maggiore di zero, viene specificata la macro `lock_and_check`, mostrata nello snippet 1.

```

1 #define lock_and_check(condition, mutexp) \
2 ({ \
3     int __ret = 0; \
4     if (mutex_trylock(mutexp)) { \
5         if (condition) \
6             __ret = 1; \
7         else \
8             mutex_unlock(mutexp); \
9     } \
10    __ret; \
11 })

```

Snippet 1: Macro `lock_and_check`

Grazie ad essa, è possibile provare ad acquisire il lock prima di controllare la condizione reale su cui eventualmente dormire; per cui un solo thread ha la possibilità di risvegliarsi avendo già acquisito il lock, se la condizione è soddisfatta. In tutti gli altri casi il valore della macro sarà zero e i thread torneranno a dormire.

Una volta usciti dalla `wait_event_interruptible_timeout`, si decrementa il numero di thread in attesa e si prosegue con la lettura effettiva dei dati.

Sia nel caso di operazione bloccante che non, una volta acquisito il lock la lettura effettiva è realizzata invocando la funzione `actual_read` in cui si cerca di copiare tutto (o in parte) l'ammontare di byte indicato dall'utente all'interno del buffer specificato. Per farlo, dovendo rispettare una politica di consegna FIFO, vengono letti i segmenti di dati a partire dalla testa della lista, facendone avanzare l'indice di lettura ed eventualmente rimuovendoli non appena vuotati. Questa operazione altera sia il numero di byte validi per il data stream (riducendolo), che quello di byte liberi (incrementandolo).

Al termine dell'operazione di lettura (o a valle di un errore riscontrato in sezione critica) viene eseguita l'API `wake_up_interruptible` per segnalare ad eventuali thread dormienti di verificare la condizione per la quale sono in attesa e infine viene restituito il numero di byte consegnati nel buffer utente (o l'opportuno codice d'errore).

2.4 Operazione di scrittura

La funzione `mfd_write` viene invocata quando si esegue un'operazione di scrittura su un multi-flow device file.

Prima di entrare in sezione critica, viene allocata la memoria per il nuovo segmento di dati che dovrà essere collegato alla lista corrispondente al data stream su cui si sta lavorando; ciò consentirà di acquisire e rilasciare il lock nel minor tempo possibile. Sia nel caso di operazione bloccante che non, per entrare in sezione critica viene svolto lo stesso lavoro analizzato per l'operazione di lettura, con la differenza che la condizione che consente di eseguire la scrittura è che il numero di byte liberi sia maggiore di zero. Per osservare un blocco in scrittura, è necessario che la dimensione dello stream di dati raggiunga il valore `MAX_STREAM_SIZE`, che per default è stato definito pari a `4*PAGE_SIZE`.

Una volta acquisito il lock, se l'operazione di scrittura ha come target il data stream ad alta priorità, deve essere eseguita in maniera sincrona e quindi viene invocata la funzione `actual_write` che collega il segmento di dati in coda alla lista. In questo caso, anche questa operazione altera sia il numero di byte validi per il data stream (incrementandolo), che quello di byte liberi (riducendolo).

Se l'operazione è invece a bassa priorità deve essere eseguita in maniera asincrona e quindi è necessario programmare l'esecuzione. A tale scopo, prima di acquisire il lock viene allocata memoria anche per una struttura `packed_write` che embedda quella da inserire nella workqueue, contenente le seguenti informazioni:

- la `work_struct` effettiva;
- il minor number del device su cui andare a scrivere;

- il puntatore al segmento di dati da agganciare;
- il function pointer che porta al blocco di codice che esegue la reale operazione.

Per cui, la scrittura a bassa priorità si riduce all'invocazione di `schedule_deferred_work`, in cui:

1. viene eseguita una invocazione a `try_module_get` per incrementare lo `usage counter` del modulo ed evitare che possa essere smontato mentre c'è ancora del lavoro da svolgere nonostante non vi siano più sessioni verso i multi-flow device file;
2. viene inizializzata la `work_struct` indicando la `deferred_write` come funzione che il `kworker daemon` dovrà eseguire e passando l'indirizzo della struttura stessa come dato;
3. viene messo in coda il lavoro.

Fatto ciò, viene decrementato solamente il numero di byte liberi, in modo tale che le altre operazioni di lettura e scrittura non siano compromesse dal fatto che un segmento di dati verrà agganciato in un istante successivo alla sua schedulazione.