Relazione Progetto SOA

Cristiano Cuffaro
matricola: 0299838
cristiano.cuffaro@outlook.com

A.A. 2021-2022

Indice

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.

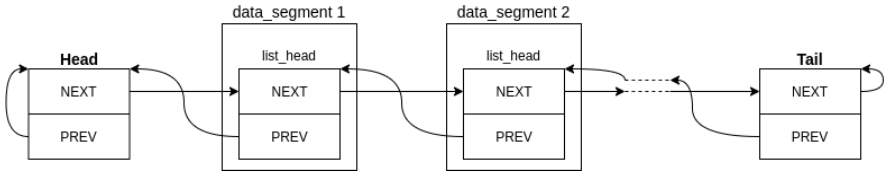

Figura 1: Lista di segmenti che implementa un data stream

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 spiazzamento a cui sono presenti i dati è arbitrario e cambia tra i vari kernel.

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, la funzione 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 specificato nei 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 event API la condizione che il numero di byte leggibili sia maggiore di zero, viene specificata la macro `lock_and_check`, mostrata nello snippet 1.

```
#define lock_and_check(condition, mutexp)       \
({                                              \
        int __ret = 0;                          \
        if (mutex_trylock(mutexp)) {            \
                if (condition)                  \
                        __ret = 1;              \
                else                            \
                        mutex_unlock(mutexp);   \
        }                                       \
        __ret;                                  \
})
```

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 si rimane in sleep.

Una volta usciti dalla wait event API, 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, la lettura effettiva è realizzata invocando la funzione `actual_read` ...