Università degli studi Roma Tor Vergata
Corso di Advanced Operating System (9 CFU)
Relazione Progetto Multi-flow device file

Benedetti Simone - 0295385

1. Introduzione

Il progetto consiste nell'implementazione di un device driver che implementi due flussi di dati distinti in base alla priorità (alta o bassa). Il data delivery segue una politica FIFO.

Il driver deve supportare fino a 128 device. Inoltre si chiede di implementare il supporto per il servizio di Ioctl() per gestire la sessione I/O e dei module parameters per la gestione del device e per fornire un'immagine dello stato corrente del device.

2. Implementazione

a. Strutture dati

La struttura dati che contiene le informazioni riguardanti un singolo device è Object_State. Essa contiene due mutex per gestire la sincronizzazione interna delle operazioni (un thread per volta scrive o legge per flusso), due wait_queue object usate per gestire le richieste di scrittura e lettura in modo bloccante, una work_queue_struct per gestire scritture asincrone richieste per la bassa priorità, due contatori per conoscere la quantità di byte presenti nei due stream e, infine, i due stream di dati. Molti di questi parametri sono doppi per poter gestire il doppio flusso di stream in parallelo.

La seconda struttura è Packed_work. Essa contiene gli stessi parametri passati dalla funzione di scrittura più il parametro di workstruct usato anche per la fare la retrieve dell'oggetto packed_work tramite la funzione container_of(). Infine la terza struttura è session_struct che è l'oggetto che identifica la sessione di I/O. Contiene le info di sessione quali alta/bassa priorità, se bloccante o non bloccante e il timeout in millisecondi.

b. Module Parameters

I parametri di modulo aggiunti sono in totale 5 e sono tutti array di lunghezza pari al numero di Minor massimi supportati. Status riguarda l'enable e il disable di un device file specifico, in termini di minor number. Num_byte_(hp/lp) differiscono per la coda in questione e identificano il numero di byte presenti nel flow specifico.

Num_th_in_queue_(hp/lp) differiscono per la coda in questione e identificano il numero di thread che aspettano i dati lungo il suddetto flusso.

c. Open()

La funzione Open crea un nuovo oggetto di sessione a cui associare il valore di timeout, se bloccante o non bloccante e se si vuole scrivere/leggere in alta o bassa priorità. Inoltre controlla se per un determinato minor, il device file è abilitato o disabilitato. Nel secondo caso la funzione fallisce.

d. Write()

La funzione di scrittura, dopo un iniziale copia del dato preso a livello user in un buffer temporaneo tramite l'uso di copy_from_user, decide come

effettuare la scrittura in base al setup effettuato in precedenza. Nel caso con alta priorità bloccante, è stata implementata una wait queue con l'utilizzo della wait_queue_timeout() nella quale la condizione di uscita dalla coda è prendere il lock per poter scrivere sullo stream specifico. Se dopo un certo timeout inserito nel setup il thread in coda non prende il lock, il thread viene risvegliato. Se prende il lock con successo, viene usata krealloc() per riallocare spazio in modo dinamico per poter scrivere nello stream, viene usata memset per settare a 0 quell'area di memoria eventualmente sporca e infine viene inserita la stringa nello stream tramite strncat prendendola dal buffer temporaneo (funzione write_op). Infine vengono aggiornati gli indici di offset e quelli relativi al valore dei byte scritti e viene rilasciato il lock ritornando il numero di byte scritti. Nel caso ad alta priorità non bloccante si cerca direttamente di prendere il lock tramite mutex_trylock e se si prende, si esegue la funzione write_op.
Nel caso con bassa priorità non si fa distinzione tra bloccante e non bloccante in quanto è richiesta un'esecuzione asincrona basata su delayed work. È stata implementata tramite l'uso di Work Queue con la sua queue_work che inserisce un thread di lavoro nella coda. Ogni work è rappresentato da una struct packed_work che si usa per ricavare le informazioni necessarie per poter effettuare la scrittura come il buffer da dove prendere i caratteri da scrivere, la lunghezza dei caratteri e l'offset di partenza. L'offset in generale è sempre prima azzerato perché creava problemi in alcune scritture essendosi sporcato. In delayed work, le scritture vengono sempre sincronizzate tramite l'utilizzo del mutex, che viene preso ogni volta prima di effettuare la scrittura e dopo aver aggiornato i vari offset, byte validi e parametri del modulo, esso viene rilasciato.

e. **Read()**

La funzione di lettura è stata implementata in modo da avere sempre una esecuzione sincrona quindi sono state adottate le wait queue, una per ogni livello di priorità, per il caso bloccante. La condizione di uscita dalla coda è la stessa della scrittura.
Nel caso non bloccante, per entrambi i flussi, si cerca di prendere il mutex tramite la mutex_trylock. La lettura avviene prima passando i dati dallo stream ad un buffer temporaneo allocato all'inizio della funzione, poi viene spostato l'inizio dello stream del numero di byte letti tramite memmove e infine settata a 0 la parte di memoria finale che è ridondante tramite la memset. Infine vengono aggiornati i parametri relativi ai byte letti e il mutex viene sbloccato per dar spazio al thread successivo. La read, a questo punto, termina con una copy_to_user fuori dal lock, che copia i byte dal buffer temporaneo al buffer di livello

user e la funzione torna il numero di byte letti.

f. **Ioctl()**

La funzione Ioctl è stata implementata per gestire la sessione, in quanto serve a fare il setup del livello di priorità e delle operazioni bloccanti o non bloccanti e regolare il valore di timeout per risvegliare i thread dormienti delle operazioni bloccanti. Per la realizzazione è stato creato un header file "ioctl.h" presente nel path di progetto che contiene 4 (+1) possibili setup cioè le 4 combinazioni tra alta e bassa priorità e bloccante o non bloccante. Per il caso di bassa priorità la scelta di operazioni bloccanti o no riguarda solo le operazioni di lettura in quanto quelle di scrittura sono solo in deferred work. Il quinto setup possibile riguarda l'abilitazione o la disabilitazione di un device file specifico (cioè quello riferito dal file descriptor che prende in input la funzione ioctl).

g. **Init_module()**

Questa funzione è chiamata al caricamento del modulo e inizializza l'array di struct object_state, oltre che i module parameters e infine registra il driver nel kernel ricavandone il Major number.

h. **Cleanup_module()**

Questa funzione è chiamata quando viene smontato il modulo dal kernel e libera la memoria occupata dalle strutture dati usate dal modulo stesso.

i. **Release()**

Questa funzione libera memoria occupata dalla struttura dati allocata in filp->private_data, cioè l'oggetto di sessione.

3. **Test**

I test presenti sono 3 e sono nella cartella /user del progetto. Per usarli, prima compilare con il comando "Make". Il test write prende in input da riga di comando il major number, 0/1 per indicare se low/high priority, 0/1 per indicare se bloccante o non bloccante e infine il messaggio da scrivere nello stream. Il test read è analogo e prende gli stessi valori da riga di comando con la sola differenza dell'ultimo parametro che è il numero di byte da leggere. Il test spawn_thread prende in input da riga di comando il major number, il minor number e il numero di thread da far spawnare. Questo è uno test che fa spawn di thread in concorrenza in scrittura o lettura(in base a quale selezione di pthread_create viene effettuata nel main).

4. **Setup e installazione**

Per installare il modulo, prima bisogna compilarlo usando il MakeFile e usando il comando "make" da terminale. Poi inserire il modulo del driver con il comando "insmod", prendere il major number tramite comando "dmesg" e usarlo nei vari test spiegati precedentemente per poter testare il modulo.