RELAZIONE PROGETTO SOA

MULTI-FLOW DEVICE DRIVER
(MATTEO FERRETTI - 0300049)

INTRODUZIONE

Per la realizzazione del progetto del corso era stato richiesto di implementare un driver per il kernel di Linux, allo scopo di permettere l'utilizzo di stream di dati da parte di thread. In particolare, il driver doveva essere in grado di mettere a disposizione uno stream ad alta ed uno a bassa priorità, sui quali era possibile leggere o scrivere dati (con modalità diverse a seconda del flusso). Era poi richiesto di mettere a disposizione alcuni parametri per controllare lo stato del sistema e di realizzare il supporto per l'ioctl così da permettere una gestione delle sessioni di I/O facilitate. Infine, è stato richiesto il supporto massimo di una quantità pari a 128 minor differenti gestibili contemporaneamente (la specifica completa è disponibile sul sito del corso, all' indirizzo *https://francescoquaglia.github.io/TEACHING/AOS/CURRENT/PROJECTS/project-specification-2021-2022.html*).
Ogni aspetto che verrà discusso in questa relazione sarà consultabile sul mio GitHub personale (*https://github.com/IronMatt97/Linux-multiflow-device-driver*), dove è disponibile il codice sorgente del progetto insieme ad un README che spiega in grandi linee come utilizzare i diversi file.

ASPETTI PROGETTUALI

Per quanto riguarda l'aspetto prettamente organizzativo del workspace, per la realizzazione del driver è stato implementato un modulo kernel chiamato "multi-flow-device.ko", ottenibile dal sorgente con lo stesso nome del suddetto all'interno della cartella radice del progetto. Per quanto riguarda tutte le altre directory sono semplicemente diversi tipi di test, lasciati al fine di osservare e stressare il comportamento del sistema in diversi modi. In particolare, come discuterò verso la fine, sono state lasciate tre utility di testing.

IMPLEMENTAZIONE DEL DRIVER

Il driver implementa diverse file operations: read, write, open e release. All'interno della init del modulo viene inizializzata quella che è la struttura principale di ogni dispositivo, la "_object_state" (insieme ai suoi dati interni a valori di default); essa è stata realizzata tramite una struct contenente tutte le informazioni relative al device. Sto parlando quindi dei flussi a bassa ed alta priorità associati, dei mutex utilizzati per garantire l'accesso non concorrente ad essi, delle code di attesa per accedere agli stream, e di variabili che tengono conto della quantità di bytes scritti in ogni flusso e dello stato attuale del dispositivo.
Un altro struct, chiamato "session" serve invece ad implementare ogni singola sessione apribile sui diversi device. Esso serve al fine di memorizzare ogni volta se la modalità operazionale vuole essere bloccante o non bloccante, su quale stream si intende scrivere (alta o bassa priorità), e a tener conto del tempo di risveglio per le operazioni bloccanti.
Mentre la struttura dei dispositivi viene allocata nell'init_module, la struttura di sessione viene allocata solo nel momento in cui effettivamente si apre una sessione verso il device tramite la dev_open. In questo modo ho reso possibile tenere traccia anche di sessioni multiple apribili contemporaneamente che non fossero caratterizzate da stessi dati associati. Al fine di utilizzare le sessioni, inoltre, va precisato che è stato utilizzato il campo private_data: grazie ad esso ho potuto ricavare in ogni operazione chiamata le informazioni di sessione. Per quanto riguarda la deallocazione delle strutture dati principali, sono effettuate all'interno della cleanup_module e della dev_release.
Gli stream sono gestiti tramite allocazione dinamica di char su richiesta tramite krealloc. In questo modo, ad ogni richiesta di scrittura o lettura lo stream viene ridimensionato

adeguatamente, tenendo anche conto di situazioni in cui i valori di ritorno del cross ring data move siano diversi da 0.

MODALITÀ OPERAZIONALE

Sia per le operazioni di read che di write, è stata implementata una logica di controllo tale da recuperare l'oggetto di sessione e controllare all'interno lo stato delle operazioni. Qualora l'operazione da eseguire sia caratterizzata da modalità bloccante, ho scelto di utilizzare delle wait queues con la loro API wait_event_timeout. Il motivo di questa scelta risiede nella possibilità di specificare un timeout dopo il quale l'attesa termini: recuperando tale valore dall'oggetto di sessione ho potuto così tener conto di un tempo di risveglio. La condizione di attesa che ho scelto di utilizzare è la try_lock, in quanto essa ritorna proprio 1 quando il lock viene preso. Ho scelto di sfruttare questo fitting del valore di ritorno nella condizione di attesa per semplicità, così da fare in modo che ogni volta che qualcuno rilascia un lock dopo aver finito di eseguire le proprie operazioni, procede anche col chiamare la wake_up. In sostanza in questo modo i thread si ritrovano ad aspettare un certo tempo all'interno della coda di qualche device, dove ogni volta che vengono segnalati provano ad acquisire il lock associato ad esso. Se riescono a prenderlo, possono uscire dall'attesa, altrimenti si rimettono a dormire fino alla prossima segnalazione o allo scadere del timeout. Inoltre, ho scelto di incrementare e decrementare un contatore atomico appena prima e dopo l'attesa: ciò è stato fatto per tenere traccia di quanti thread attendono per esporlo come parametro di modulo, come richiesto dalla traccia.
Detto ciò, per quanto riguarda la modalità operazionale non bloccante, ho scelto di implementarla tramite una semplice try_lock. Qualora il lock venga acquisito il thread procede con le operazioni, altrimenti ritorna il controllo al chiamante.
Sia nel caso bloccante che non bloccante, sia che un thread riesca ad ottenere il lock, sia che non riesca, viene notificato dal kernel l'esito tramite una printk.

SCRITTURE E LETTURE

Per quanto riguarda la write, era stato richiesto di realizzarla in maniera tale che fosse sincrona per lo stream ad alta priorità, ed asincrona per quello a bassa priorità. Nello specifico, per l'alta priorità viene semplicemente utilizzata una wait_event_timeout nel caso bloccante, o una try_lock sul mutex nel caso non bloccante. Per la bassa priorità ho scelto di implementare una logica di deferred work tramite work queues di sistema. In particolare, ogni volta che viene richiesta una scrittura a bassa priorità viene allocato uno struct di lavoro chiamato "_packed_work", contenente diversi dati, tra cui cosa scrivere sullo stream e l'indirizzo del dispositivo a cui accedere. Utilizzando il costrutto container_of, all'interno della funzione di lavoro ogni volta viene recuperato lo struct di lavoro; in questo modo, ho reso possibile il recupero del riferimento al device e dello struct di sessione. Anche per il deferred work ho scelto di implementare un controllo sulla logica operazionale, così da mantenere poi bloccato o meno il kworker che si occuperà di risolvere il task (secondo la stessa modalità già discussa prima). Le read non implementano il deferred work per la bassa priorità invece; quindi, per entrambi gli stream varia solo la modalità operazionale bloccante o non bloccante nelle stesse circostanze descritte sopra.

A livello implementativo, le write rimettono sempre a 0 l'offset di scrittura inizialmente, per poi spostarsi di una quantità di posizioni pari ai byte validi nello stream; a quel punto tramite una krealloc viene allocato per lo stream lo spazio necessario per effettuare la nuova scrittura. Per copiare i dati livello utente è stata utilizzata una copy_from_user. Nel caso deferred questa operazione viene fatta nel momento di istanziazione della struttura di lavoro, per poi andare semplicemente a copiare i dati nello stream in un secondo momento. Eseguita la scrittura, vengono aggiornati gli offset e la quantità di bytes validi sullo stream, per poi rilasciare il mutex necessario e risvegliare la wait queue interessata. Le read invece sfruttano la

copy_to_user per inviare i dati livello utente, mentre per quanto riguarda la consumazione dei dati dal buffer ho scelto di utilizzare una memmove. Grazie ad essa, dopo ogni lettura viene copiata la residua quantità di bytes all'inizio del flusso; successivamente tramite una memset viene azzerata la parte finale, così da completare l'operazione di eliminazione dei bytes letti con una pulizia del residuo precedente. Infine, viene chiamata la krealloc così da ridimensionare lo stream.
Sia la read che la write richiedono l'acquisizione di un mutex, necessario affinchè non avvengano operazioni concorrenti sui singoli stream. Gli stessi mutex sono quelli passati come parametro ai try_lock sia nei casi bloccanti con le wait queues, sia nei casi non bloccanti.

SUPPORTO PER L'IOCTL

Il modulo espone diverse funzionalità attivabili tramite parametro passato all'ioctl. È possibile modificare infatti la modalità operazionale bloccante o non bloccante per una sessione o la scelta dello stream di un device su cui scrivere. È poi possibile modificare il tempo di risveglio per le operazioni bloccanti, così da decidere il tempo di attesa massimo sulle wait queues. Ognuno di questi dati si trova all'interno dell'oggetto di sessione.
Oltre le suddette funzionalità richieste, ho deciso di implementare anche un ulteriore comando al fine di facilitare l'abilitazione e la disabilitazione dei device, dato di stato presente nell'oggetto del dispositivo.

PARAMETRI ESPOSTI

Ho realizzato diversi parametri modulari, al fine di garantire una più semplice consultazione dello stato interno del sistema durante la sua attività.
A livello implementativo ho scelto di utilizzare dei module_param_array per rappresentare la situazione di abilitato/disabilitato per ogni device, della quantità di thread che attendono sugli stream (sfruttando il contatore atomico strategico a ridosso delle attese sulle wait queues) e della quantità di byte validi scritti nei flussi dei dispositivi. L'unico parametro modificabile da /sys/modules/ è quello di abilitato/disabilitato; gli altri sono solo consultabili.
Dato che il parametro di stato è l'unico modificabile quindi ho implementato anche un array di puntatori relazionato ad esso all'interno dello struct dell'_object_state, così da poterlo direttamente trovare nello stato dell'oggetto.

TESTING

Al fine di testare le funzionalità del modulo che ho realizzato, ho sviluppato tre differenti utilities di testing. Lo scopo di esse è mettere in luce il comportamento del sistema sotto diversi punti di vista, oltre il facilitare una fruizione controllata di esso. Ogni programma è stato riposto all'interno di una sottodirectory specifica a partire dalla directory di radice.

TEST FUNZIONALE

Questo test serve a seguire il funzionamento base di ogni attività del sistema. All'interno della directory "functions_test", è possibile trovare l'eseguibile in questione. Esso di base va ad interagire con il sistema in maniera single threaded, così da poter controllare l'esito operazione dopo operazione usando dmesg. Il programma si basa su un prompt interattivo che apre una sessione verso un dispositivo ed effettua l'operazione richiesta per poi richiuderla. Sarà possibile quindi effettuare una lettura, una scrittura o un ioctl su un device scelto.

TEST DI CONCORRENZA

All'interno della directory "concurrency_test" è possibile trovare il suddetto test, che si occupa di stressare il sistema con una serie di operazioni concorrenti al fine di testarne la solidità. Nella demo che ho scritto ho lasciato per semplicità solo thread che fanno un'operazione per sessione, ma ovviamente è possibile modificare il sorgente liberamente anche per cambiare la quantità di lavoro per sessione. Il test prevede tre fasi, una dove avvengono solo scritture concorrenti su entrambi gli stream dei diversi device, un'altra

dove avvengono letture con le stesse modalità, ed una finale dove letture e scritture avvengono contemporaneamente.
Il sistema quindi si ritroverà a gestire tramite mutex, attese su wait queues e deferred work tramite work queues tutta la mole di richieste che arriveranno. Il timeout delle operazioni bloccanti l'ho lasciato impostato di default a 20 microsecondi, così da rendere più probabile che qualche thread sin da subito si ritrovi il timeout scaduto per operazioni bloccanti. Tale valore è stato misurato sulla base di analisi prestazionale del sistema, che discuterò a breve.
Terminato il test è possibile consultare con dmesg l'esito, e quindi controllare quante operazioni sono andate a buon fine e quante no (per timeout scaduti nel caso bloccante, o semplicemente per mutex già presi nel caso non bloccante).

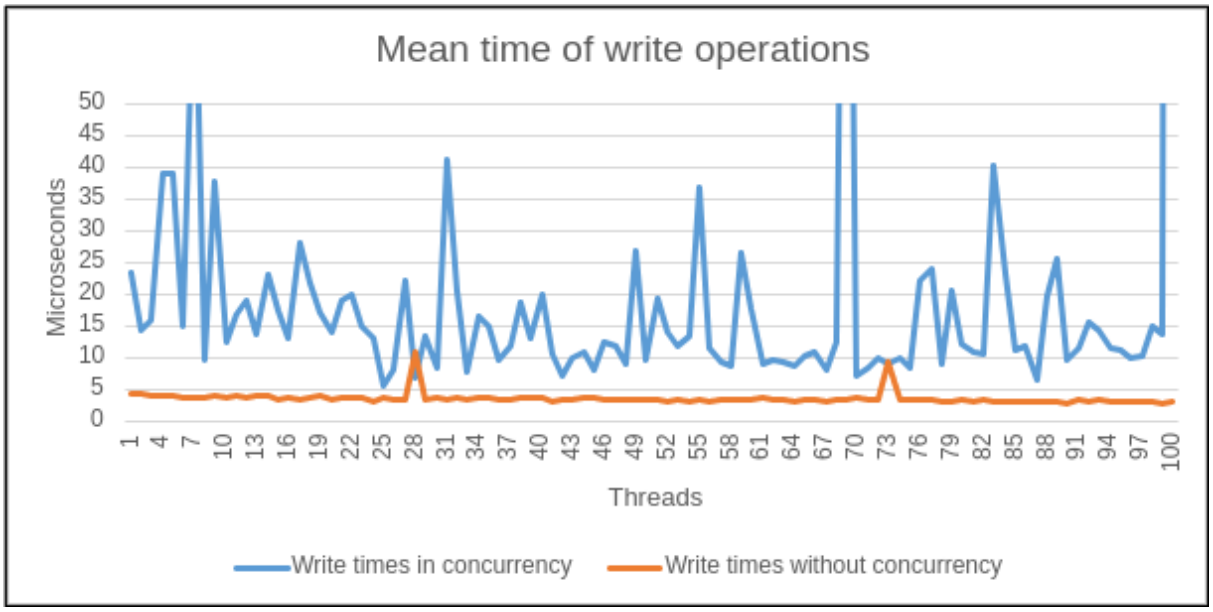
TEST PRESTAZIONALE

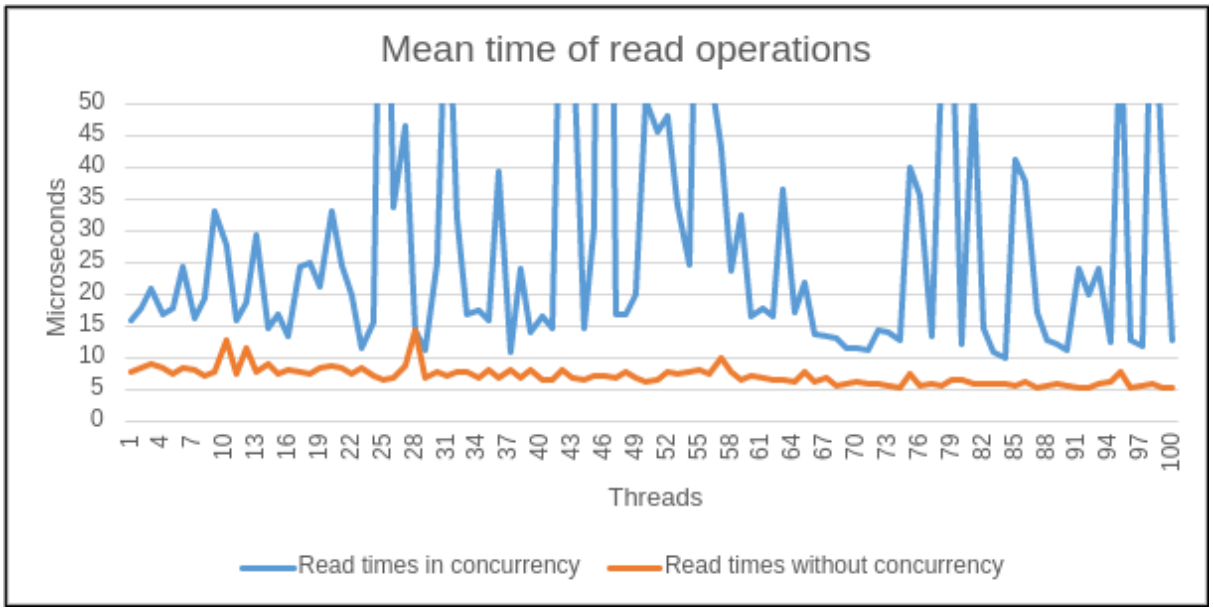
Questo test è stato realizzato al fine di misurare i tempi di risposta del sistema per le operazioni di lettura e scrittura. Per quanto dipendente poi dall'hardware su cui il modulo verrà montato, ho ritenuto comunque utile andare a studiare le prestazioni del sistema al fine di averne un'idea. Inoltre, è risultato utile anche per riuscire a trovare un valore di attesa valido per le operazioni bloccanti tale da testarne il tempo di scadenza.
Il test lavora di default sul dispositivo con minor 0, e prevede lo spawning di un numero di thread configurabile, ripetuto per una quantità di volte configurabile. Ogni volta, ogni thread calcolerà il tempo di risposta del sistema tramite l'API clock della libreria time.h.
Il test viene effettuato quindi prima spawnando tutta la suddetta quantità di thread contemporaneamente, e poi viene ripetuto invece misurando i tempi con un thread alla volta. I risultati del test vengono salvati in un log locale alla directory "performance_test", dove è appunto presente anche questa utility di testing.
Sempre all'interno della cartella ho lasciato anche un foglio di calcolo dove ci sono i dati già analizzati da me, sulla base di 10 run con 100 thread al fine di ottenere un valore medio indicativo delle prestazioni.
Il test può tornare utile qualora si decida di far magari partire il codice su una diversa macchina, così da misurare i tempi di risposta del sistema e modificare sensatamente il default timeout del modulo.

ANALISI PRESTAZIONALE

I tempi di risposta del sistema sono dipendenti dall'hardware su cui andranno a girare; è ovvio quindi che i risultati che discuterò di seguito sono quelli ottenuti sulla mia macchina e non è garantito che saranno gli stessi su altre. Tuttavia, ho ritenuto fosse utile ed interessante avere comunque a disposizione un'analisi indicativa dei tempi prestazionali, dato che in "tempi umani" l'hardware non cambi poi tanto le cose. A livello indicativo, le operazioni di lettura e scrittura si aggirano, in situazioni ideali (un thread per volta), sui 5-10 microsecondi. Nel caso in cui invece il sistema sia sottoposto a thread concorrenti, variano tra i 20 e i 100 microsecondi. Questa varianza è data dal fatto che ognuno può ritrovarsi ad aspettare dietro ad altri; quindi, misurando dei tempi di attesa che comprendono il lavoro di chi arriva prima di lui. Le ripetizioni di uno stesso test modificabili in maniera parametrica nel sorgente dell'eseguibile servono proprio a ridurre l'impatto di quella varianza: i grafici di analisi riportati sotto sono infatti ottenuti sulla base della media ottenuta da testing ripetuti.

Seguono a pagina seguente i grafici dei tempi di esecuzione per le operazioni di lettura e scrittura volti a confrontare i tempi del sistema quando è libero e quando invece è inondato di richieste.


Mean time of write operations
Microseconds
Threads
Write times in concurrency
Write times without concurrency


Mean time of read operations
Microseconds
Threads
Read times in concurrency
Read times without concurrency