Relazione progetto Sistemi Operativi avanzati

Mirko Leandri 0299946

Introduzione

La specifica del progetto prevede l'implementazione di un sottosistema kernel che ci consente di scambiare messaggi tramite "TAG", attraverso la scrittura e l'inserimento di quattro nuove chiamate di sistema: tag_get, tag_send, tag_receive, e tag_ctl. Inoltre è prevista la scrittura di un device driver tenente tracia dello stato attuale del programma. Per i dettagli visitare il seguente link: https://francescoquaglia.github.io/TEACHING/AOS/AA-2020-2021/PROJECTS/project-specification-2020-2021.html.

Implementazione e scelte progettuali

System call table hacking

Per prima cosa, ho deciso di andare a fare hacking della system call table, attraverso il codice trovato sulla repository github del docente che è possibile reperire al seguente link: https://github.com/FrancescoQuaglia/Linux-sys_call_table-discoverer. Questo codice vede la creazione di un modulo che va appunto a fare hacking della system call table, a cercare gli spazi dove sono presenti ni_syscall che puntano allo stesso indirizzo x che racchiude le system_call che non hanno una reale implementazione. Questo perché difficilmente possiamo operare resizing della tabella, e non ci sono attualmente in essa entry libere. Vado a modificare il codice del professore introducendo appunto le 4 chiamate di sistema citate nel paragrafo precedente, inoltre ho inserito anche la possibilità di prendere la posizione delle ni_syscall e inserirla in un array che mi permette di poter eventualmente inserire le system call in posizioni dinamiche e dipendententi dall'esecuzione. Alla fine però pur lasciando questa parte presente nel codice utilizzo la parte scritta dal professore e le posizioni sono quindi indicate a priori con delle MACRO. Ovviamente c'è anche la funzione che va a fare lo smontaggio del modulo e a reinserire ni_syscall nelle posizioni indicate appunto dalle MACRO, inoltre libera anche le aree di memorie riservate ai tag e ai livelli che allocherò nella services.

Strutture dati

Oltre le varie costanti date in consegna, qui vengono definite le strutture dati usate dal device driver, dai tag e dai livelli, in particolare, per il tag la struct ha tra i campi più interessanti key, command e permission definiti nella specifica, il puntatore alla struttura dei livelli, exist è un intero che ci dice se il tag è stato creato oppure ancora non esiste. Per il livello invece abbiamo un puntatore a char che sarà il buffer contente il messaggio, la wait_queue con i thread che devono essere messi nello stato TASK_INTERRUPTIBLE dalla tag_receive, il numero dei lettori, e il campo is_empty che ci dice se il buffer è vuoto o meno (verra aumentato dalla send e diminuito dalla receive una volta ricevuto il messaggio), che è anche la discriminante della chiamata di sistema wait_event_interruptible(`wait_queue_head_t wq, condition`), che appunto mette in stato di TASK_INTERRUPTIBLE fino a che la `condition` non da true. Inoltre abbiamo il flag is_queued che ci serve nella tag_ctl per capire se si tratta di un thread che è stato precendemente messo a dormire.

Services

Qui implemento le funzionalità delle 4 chiamate di sistema della consegna. La prima è la tag_get, che vede per prima cosa l'allocazione della memoria per i tag e per i livelli, poi passo alla scelta data dal parametro "command". Il valore 1 ci indica che vogliamo creare un nuovo tag, mentre il valore 2 che vogliamo aprire un tag già esistente. Interessante in quest'ultima è il controllo fatto sulla key del tag che se uguale a IPC_PRIVATE non potrà essere riaperto.

Nella scelta della sincronizzazione ho tenuto conto della RCU, che abbiamo studiato durante il corso, dato che lo scenario poteva essere visto come quello in cui ci sono svariati lettori e meno scrittori, lo scrittore doveva essere bloccato, mentre i lettori potevano essere concorrenti.

Questo ci viene incontro nell'implementazione appunto della trag_send e della tag_receive.
Nella "send", dopo aver effettuato controlli sull'esistenza del tag, sul "dominio" della sua chiave e sul flag che ci indica se è aperto o meno, effettuo il controllo sul numero di lettori, un parametro che viene aumentato di 1 (per il tag e il livello chiamato) dalla tag_receive, se questo intero è minore di 1 il messaggio viene scartato in quanto non sono presenti lettori. A questo punto metto una spin_lock per scrivere sull'area di memoria del livello, che potrebbe essere scritta altrimenti da piu thread, e usiamo per

scrivere la funzione copy_from_user. Infine aumentiamo atomicamente con una synch_fetch_and_add il parametro is_empty e mandiamo una wake_up_interruptible ai thread che sono stati bloccati e che aspettano un messaggio in quel livello di quello specifico tag.

La “receive” per prima cosa va aumentare atomicamente il numero di lettori e il flag che indica che il tag sta per essere messo a dormire, poi metto uno spin_lock per andare ad allocare la memoria specifica per quel livello che verrà liberata una volta letto il messaggio (solo se non è ancora stata allocata). Liberato il lock, parte la wait_event_interruptible che mette in attesa i thread fino a che il flag is_empty di quel livello sia diverso da 0, oppure fino alla ricezione di un segnale. Se il valore di ritorno di questa è 0, significa che la condizione è stata rispettata, a quel punto imposto rcu_read_lock(), diminuisco atomicamente i lettori, e tramite una copy_to_user metto nel buffer passato dall’utente il messaggio presente nel livello specifico. A questo diminuisco il flag is_empty e se il flag dei lettori è minore di 1 vado a liberare l’area di memoria del buffer.

L’ultima chiamata di sistema che ho implementato è la tag_ctl, anch’essa come la tag_get ha due command possibili, 1 sta per “AWAKE_ALL”, che cicla su tutti i livelli di tutti i tag inserendo nei loro buffer un messaggio standard, va ad aumentare il flag is_empty che come nel caso precedente è la condizione della wait_event_interruptible, quindi viene chiamata una wake_up_interruptible per ognuno di questi thread bloccati.

Il secondo comando permette di cancellare i tag, ci appoggiamo a due funzioni presenti nel file util_tag.c, che sono search_for_level e delete_tag: la prima va a scorrere tutti i livelli di un dato tag e se non sono presenti messaggi ci ritorna TRUE, che è la condizione necessaria a far partire la delete_tag, che semplicemente crea una nuova struct tag in cui inizializza i paramentri e poi la assegna alla struttura principale.

Device Driver

Attraverso un altro modulo vado a inserire un device driver, basandomi sul codice visto a lezione concurrency_driver, la funzione init_module ci permette di inizializzare un device driver identificato da MAJOR_NUMBER e in esso una quantità di oggetti pari a MINOR_NUMBER. Anche qui è presente anche una funzione di clean-up. Il lavoro principale è stato fatto nelle funzioni di read e di write, con la read dopo aver lasciato i controlli del codice natio, vado a scrivere sull'oggetto il numero di reader, la key del tag e "permission", che per scelta di implementazione ho fatto uguale a 0 se il tag può essere acceduto in maniera pubblica, oppure uguale al tid del thread se può essere acceduto solo da quest'ultimo. Nella read tramite una copy_to_user vado a inserire il contenuto degli oggetti in un buffer fornito dall'utente. Nel device driver uso una sincronizzazione con i mutex e vado a rendere bloccanti le operazioni di lettura e scrittura su di esso.

Parte User

Nel file user.c ho implementato una simulazione di un eventuale uso che un utente potrebbe fare delle chiamate di sistema. In particolare inizio implementando i frontend delle chiamate di sistema che vanno a chiamare "syscall" con i numeri predefiniti assegnati dal syscall_filler.c, poi abbiamo vari test, che inserisco nella funzione main, nel primo test_create_open_remove vado a testare appunto la creazione, la rimozione e l'apertura di un certo numero di tag, provando anche il funzionamento della logica IPC_PRIVATE.
Il test multithread prevede la creazione appunto di più thread, alcuni dei quali creano dei tag e poi usano una receive per mettersi in attesa, mentre altri inviano messaggi attraverso una send.