Progettazione ed implementazione di un kernel subsystem per lo scambio di messaggi tra thread

Sistemi Operativi Avanzati 2021

Pasquale Caporaso
Studente di ingegneria informatica
Università degli Studi di Roma "Tor Vergata"
Roma, Italia
caporasopasquale97@gmail.com

I. Introduzione

In questa relazione viene presentato un modulo per Kernel Linux che permette lo scambio di messaggi tra thread in esecuzione su una macchina. L'obiettivo che mi sono posto per questo progetto, oltre a rispettare le specifiche rilasciate, è stato quello di creare un sistema sicuro ed efficente che fa pieno utilizzo delle tecniche di sincronizzazione non bloccante studiate a lezione. In questa relazione discuterò di come ho raggiunto questi obiettivi, nella sezione II descrivo le strutture che vengono utilizzate per la gestione delle informazioni, nella sezione III parlo di come le syscall interagiscono can tali strutture, nella sezione IV discuto delle difficoltà affrontate per quanto riguarda la sincronizzazione, la sezione V contiene una descrizione del driver che fornisce informazioni sul sistema ed, infine, nella sezione VI ho riportato i dettagli riguardanti i test a cui ho sottoposto il modulo. Il sistema è stato progettato per Kernel 5.8 e testato su una macchina Linux Ubuntu 20.10, il modulo dovrebbe funzionare correttamente su ogni versione di Kernel Linux successiva alla 4.6 ma non posso garantirlo in quanto non è stato testato.

II. Strutture di dati

Il sistema deve mantenere diverse informazioni per il corretto funzionamento, queste sono raccolte nella serie di strutture collegate, mostrate in figura 1, queste sono:

A. struct tag

La struct tag contiene le informazioni specifiche ad un particolare tag, questa struttura viene allocata e deallocata a seconda delle necessità, per questo motivo alcune delle informazioni, la chiave e la versione, sono salvate in vettori a parte in modo da essere mantenute anche quando la struttura viene deallocata. In totale possono essere presenti fino a 256 *struct tag* che vengono accedute tramite il vettore *tag_array* le componenti di questa struttura sono:

- *char private*, se il tag può o non può essere instanziato da una chiamata a *tag_get*, i valori possibili sono IPC_PRIVATE e IPC_PUBLIC;

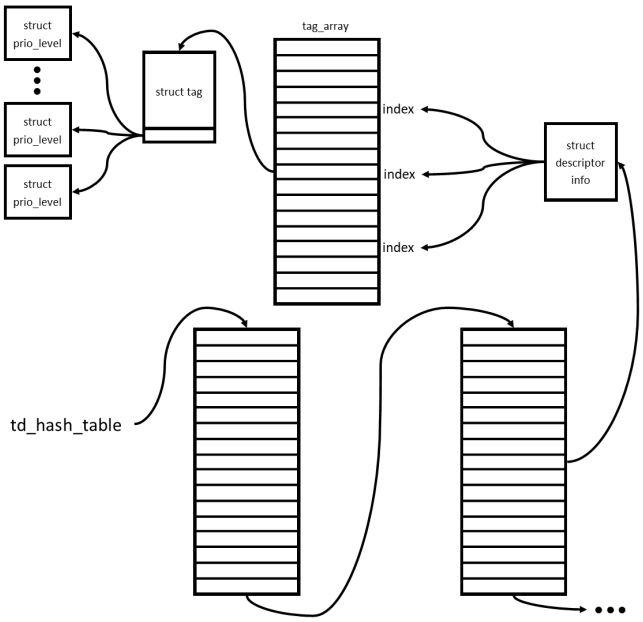

Fig. 1. Uno schema delle strutture del sottosistema

- *short reference_counter*, il numero di thread che si trovano in ascolto sul tag, usato per prevenire eliminazione, può avere valore -1 nel momento in cui il processo di eliminazione è in corso, questo impedisce a thread di mettersi in asccolto;
- *kuid_t owner*, l'utente proprietario del tag se è stato creato con la flag T_PERSONAL;
- *prio_level* priority_levels[LEVELS_MAX]*, vettore contenente la struttura dei 32 livelli su cui può essere spedito un messaggio, questa contiene un buffer e un reference counter usati nelle operazioni di scrittura e lettura;
- *ktime_t last_use*, l'ultimo istante di tempo in cui il tag è stato utilizzato, può essere utilizzato, se necessario, per riciclare tag non più utilizzati;

B. struct descriptor_info

La *struct descriptor_info* viene associata ad un thread e contiene informazioni su tutti i descrittori aperti da tale processo, la struttura è ereditata dai figli al primo acesso al sistema e viene allocata all'apertura del primo descrittore e deallocata quando non ci sono più descrittori collegati. Le componenti della struttura sono:

- *unsigned char* descriptors*, vettore che associa i numeri dei tag descriptor ai numeri del tag associato, viene usato un vettore per garantire tempo di accesso costante e questo viene riallocato se è necessario più spazio;
- *short* versions*, vettore che contiene le versioni dei tag al tempo di collegamento dei descrittori, questo è necessario per assicurarsi che il tag collegato non è stato chiuso e riaperto;
- *bitmap descripto_bitamp*, bitmap che indica quali dei descrittori nel vettore sono validi;
- *int curr_alloc*, la dimensione attuale del vettore, usata per capire quando è necessario riallocare;
- *real_pid*, pid del processo associato a questa struttura, questo valore viene usato per risolvere collisioni;

Per tenere traccia delle strutture viene utilizzata la *td_hash_table* che associa ad ogni pid la rispettiva struttura, la tabella hash usa come indice il pid del processo MOD 512 in modo da occupare esattamente una pagina di memoria da 4 Kb, in caso di collisioni una nuova tabella viene allocata e collegata alla precedente, un organizzazione di questo tipo permette di eseguire la traduzione da "Tag descripor" a "Tag" in tempo costante, questo è fondamentale in quanto è una operazione che deve essere eseguita ad ogni utilizzo del sistema.

III. System Calls

Le syscall, come richiesto dalla specifica, sono le seguenti:

A. tag_get

La syscall *tag_get* viene usata per creare o aprire un Tag associato ad una chiave. L'esecuzione della syscall procede nel seguente modo:

1) Controlla se le flag inserite sono corrette;
2) Se la chiave non è già associata ad un tag, una nuova struttura *tag* viene allocata, la chiave viene salvata nel vettore, viene aggiornata la versione e si passa al passo 4;
3) Se la chiave è già presente si controlla se il tag è pubblico e si vede se l'utente linux attuale può accedere al tag, solo se tutti i controlli hanno successo si va al passo successivo;
4) Viene aggiunto un nuovo descrittore nella struttura *descriptor_info* del processo, allocandola o espandendola se necessario, e dopo si ritorna il numero all'utente;

La creazione di nuovi tag è un'operazione particolarmete sensibile dal punto di vista della sincronizzazione, discuterò in dettaglio la sua gestione nella sezione IV.

B. tag_receive

La syscall *tag_receive* viene usata per ricevere un messaggio su da tag ad un particolare livello. L'esecuzione della syscall procede nel seguente modo:

1) Controlla se i valori inseriti sono corretti, in particolare il livello deve essere positivo e minore di LEVELS_MAX;
2) Recupera il struct *tag* associata al descrittore leggendo la struct *descriptor_info* a lui corrispondente;
3) Si mette in attesa di un messaggio sulla struttura *prio_level* corrispondente al livello prendendola dalla struttura *tag*;
4) All'arrivo del messaggio il thread si sveglia, legge il buffer della struttura del livello e lo consegna all'utente;
5) Controlla se ci sono altri thread in attesa, se non ce ne sono, dealloca il buffer e il livello;

Per avere sincronizzazione non bloccante tra i thread le letture e le scritture vengono associate ad una particolare struttura *prio_level* che viene scambiata ad ogni scrittura, questo viene spiegato nei dettagli nella sezione IV.

C. tag_send

La syscall *tag_send* viene usata per inviare messaggi a un tag su un particolare livello. L'esecuzione della syscall procede nel seguente modo:

1) Controlla se i valori inseriti sono corretti, in particolare il livello deve essere positivo e minore di LEVELS_MAX;
2) Recupera il struct *tag* associata al descrittore leggendo la struct *descriptor_info* a lui corrispondente;
3) Controlla se sulla struttura *prio_level* associata ci sono dei thread in ascolto, se non ce ne sono la call fallisce;
4) Alloca una nuova struttura di livello e la scambia con quella attuale, salvandola da parte, questo scambio permette a nuove scritture di usare la nuova struttura senza andare in conflitto con l'attuale;
5) Inserisce il buffer nella struttura *prio_level* e sveglia tutti i thread in ascolto;

Faccio notare che la syscall di send non aspetta che tutte le letture siano avvenute per terminare, questo non è necessario in quanto tutte le strutture impiegate vengono rilasciate dall'ultimo listener. Questa scelta permette delle scritture non bloccanti, molto più veloci, che però non garantiscono che il messaggio arrivi a tutti i thread in ascolto, credo che comunque questa sia la scelta migliore in quanto l'obiettivo di questo modulo non è quello di scambiare messaggi in maniera reliable.

D. tag_ctl

La syscall *tag_ctl* permette all'utente di eseguire due azioni: AWAKE_ALL, per svegliare tutti in thread in ascolto su un tag; REMOVE, per eliminare un tag. L'esecuzione del comando AWAKE_ALL procede nel seguente modo:

1) Recupera il struct *tag* associata al descrittore leggendo la struct *descriptor_info* a lui corrispondente;
2) Scambia le strutture *prio_level* di ogni livello ed esegue una scrittura vuota su ognuna di esse, in particolare

imposta il valore di *size* a -1, facendo così i thread in ascolto sanno di essere stati svegliati senza un messaggio ed ignorano il buffer;

L'esecuzione del comando REMOVE procede nel seguente modo:

1) Recupera il struct *tag* associata al descrittore leggendo la struct *descriptor_info* a lui corrispondente;
2) Controlla che non ci siano thread in ascolto su questo tag, se ce ne sono la call fallisce;
3) Imposta il valore di *reference_count* a -1, questo segnala ad ogni thread che ha accesso a questo tag che l'eliminazione è iniziata;
4) Procede con la deallocazione della struttura del tag, il cancellamento della chiave e l'invalidazione di ogni descrittore associato;

La chiusura del Tag è un'azione che ogni utente dovrebbe fare quando ha smesso di utilizzarlo. Il sistema deve però considerare anche in caso di utenti distratti o malevoli che non chiudono il tag al termine del loro programma, questo problema viene risolto con la variabile di *last_use* nella struttura del tag che segna il tempo dell'ultimo utilizzo del servizio, il modulo provvede in maniera automatica all'eliminazione di tag che sono inutilizzai per lungo tempo in caso ci sia bisogno di spazio.

IV. Gestione della sincronizzazione

Uno degli obiettivi che mi sono posto con questo modulo è stato quello di ottenere il massimo livello di sincornizzazione non bloccante tra i processi, questo è stato possibile nella maggior parte dei casi semplicemente utilizzando gli accessi atomici alla memoria offerti dalle operazioni di sincronizzazione di *gcc*, ritengo però che ci siano due casi in cui lo schema di coordinazione è risultato più complesso e che quindi, richiedono una maggiore discussione, questi sono:

A. Creazione di nuovi tag e gestione delle chiavi

La creazione di un nuovo tag è naturalmente un'operazione complessa, bisogna prestare particolare attenzione ad impedire a due thread con la stessa chiave di cominciare simultaneamente il processo di inserimento, questo causerebbe la creazione di due tag con la stessa chiave generando un generale malfunzionamento del sistema.

Tuttavia rendere atomico l'intero processo di ricerca della chiave e eventuale creazione del tag avrebbe un impatto molto pesante sulle performance e coinvolgerebbe anche i thread che non stanno cercando di creare un tag, ma che invece vogliono accedere ad uno già esistente. Inoltre, non vogliamo neanche limitare thread che stanno cercando di creare tag con chiavi diverse in quanto queste sono operazioni che possono tranquillamente essere eseguite in parallelo.

Per ottenere il massimo livello di parallelismo ho utilizzato una *insert_list* che contiene tutte le chiave che sono in fase di inserimento del sistema, l'accesso a questa lista è atomico ed avviene solo quando si cerca di creare un nuovo tag, quindi non coinvolgendo l'ottenimento di un descrittore da un tag esistente.

Un thread che vuole inserire una chiave controlla la lista e, se la sua chiave è presente, si mette in attesa della conclusione della conclusione dell'inserimento, se non c'è invece, sarà lui stesso ad avviarlo. Utilizzando questo schema ci si assicura che i thread siano coordinati e si ottiengono le migliori performance possibili.

B. Lettura e scrittura di un messaggio

Le operazioni di lettura e scrittura dei messaggi sono al cuore del modulo, per questo motivo ho prestato particolare attenzione alla loro gestione e sono riuscito ad ottenere una sincronizzazione non bloccante completa, assicurando che letture e scritture possano avvenire senza attese in ogni momento.

Per ottenere questo ho fatto utilizzo la struttura *prio_level* che viene sostituita in ogni scrittura, come mostrato in figura 2 i listener si mettono in ascolto salvando un puntatore alla struttura del livello, quando avviene una scrittura una nuova *struct prio_level* viene allocata e sostistuita atomicamente tramite CAS, in questo modo abbiamo già una nuova struttura su cui i thread possono mettersi in ascolto e su cui possono avvenire altre scritture. Il writer che ha eseguito la CAS ha, invece, salvato la struttura precedente e proseguira su di essa l'operazione di sscrittura.

Questo parallelismo è particolarmente importante perchè le letture, dovendo svegliare molti thread, potrebbero avvenire in un tempo relativamente lungo e questo potrebbe penalizzare molto le performace in situazioni con molto traffico, utilizzando questo sistema, invece, si ottiene massima concorrenza tra thread al prezzo di dover mantenere in memoria qualche struttura di livello in più.

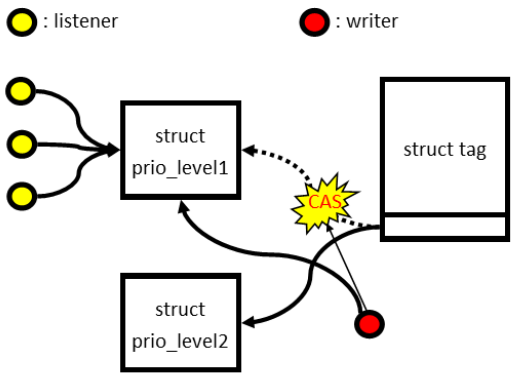

Fig. 2. Operazione di scrittura

V. Driver per la lettura delle informazioni

All'interno del modulo che implementa il tag service è presente anche un driver che permette l'accesso alle informazioni del sistema in tempo reale, questo driver espone tutti i dati sui tag aperti, le loro chiavi, i loro proprietari ed eventuali thread in attesa su ogni livello.

La sua implementazione è relativamente semplice, viene mantenuto in memoria un buffer con le informazioni che viene aggiornato leggendo le strutture durante la read se è passato un tempo sufficentemente lungo, questo conportamento può essere configurato tramite un parametro per ottenere aggiornamenti più o meno frequenti.

Il buffer viene, inoltre, deallocato se è passato un tempo sufficentemente lungo dall'ultimo utilizzo, tenendo conto di eventuali thread che potrebbero farne riferimento.

VI. Testing

Sono stati eseguiti diversi test sul sistema, questi non vogliono essere una test suite formale ma semplicemente fornire un esempio di utilizzo e una garantire il funzionamento delle funzionalità più basilari.

I test eseguiti sono i seguenti:

- Simple use, semplice scambio di messaggi tra due thread
- Awake, test della funzionalità di AWAKE che sveglia tutti i thread in ascolto
- Concurrency, uno "stress" test in cui molti thread si scambiano messaggi diversi su tag diversi
- Level, simile al Concurrency, solo che i messaggi vengono scambiati sullo stesso tag, ma a livelli diversi
- Same Owner, un test che controlla che l'accesso sia correttamente limitato in caso di Tag con proprietario
- Private, test che controlla che l'accesso sia correttamente negato in caso di Tag creati con la flag IPC_PRIVATE
- Table Stress, uno stress test che assicura il corretto ridimensionamento della tabella hash in situazione di forte traffico di thread