Progetto Sistemi Operativi Avanzati 2020/2021

Marco Giorgi
Matricola: 0266724
Email: marco.giorgi90@outlook.com

ABSTRACT

Un modulo nel contesto del kernel è un componente software che può essere aggiunto al kernel in modo dinamico, quando quest'ultimo è già a runtime. Utilizzare dei moduli permette di apportare eventuali cambiamenti senza compilare necessariamente l'intero kernel, ma solamente i rispettivi moduli. Questo fatto rende la tecnologia dei moduli molto flessibile.

Keywords

hashtable, system call, rcu, device driver

1. INTRODUZIONE

In questa relazione viene illustrato come a livello implementativo è stato realizzato un sottosistema del Kernel Linux che consente lo scambio di messaggi tra thread. Come da richiesta, sono stati gestiti 256 tag service ciascuno con 32 livelli, in particolare sono state definite specifiche systemcall per la gestione di questi servizi. Più thread reader possono sottoscriversi ad un livello di un tag service ed attendere che venga scritto un messaggio da parte di un thread writer. Viene, inoltre, offerto un device driver che ha l'obiettivo di verificare lo stato corrente del sottosistema mostrando per ogni tag service e per ogni livello di questi le seguenti informazioni: - chiave - creatore - livello - il numero di thread in attesa di messaggi.

2. AMBIENTE DI SVILUPPO

Sviluppo del progetto effettuato su Virtual Machine utilizzando come Sistema operativo guest Ubuntu 18.04-desktop-amd64 e versione del Kernel Linux 4.15.0-20-generic. I test finali sono stati effettuati su versioni del kernel precedenti e successive al 4.17 al fine di verificare il corretto funzionamento del sottosistema.

3. SCELTE IMPLEMENTATIVE

Per gestire i tag service è stata implementata una hashtable, per i tag level un array ed infine per i tag descriptor liste collegate di livello kernel. Il sottosistema è realizzato cercando di sfruttare il più possibile meccanismi di sincronizzazione non bloccanti.

3.1 Hashtable

L'idea è quella di utilizzare una hashtable per tener traccia del mapping key-tag descriptor. La scelta di sfruttare questa struttura piuttosto che altre risiede nella capacità della hashtable di effettuare operazioni di ricerca, inserimento e rimozione in O(1). Questo è importante perchè l'accesso ai tag service e ai livelli per le operazioni di lettura e scrittura deve essere effettuato il più rapidamente possibile. Nel kernel Linux è definita la hashtable linux/hashtable.h che propone un'interfaccia di hashtable abbastanza semplice con macro che forniscono meccanismi di sincronizzazione di tipo RCU, adatte quindi ad implementazione del tipo read intensive. La tabella è implementata usando dei bucket, ovvero un elenco collegato che conterrà tutti gli oggetti sottoposti ad uno specifico hash. La funzione hash "myhash" distribuisce gli oggetti uniformemente tramite la funzione modulo 256 applicata al parametro chiave.

3.2 Tag Descriptor

Ogni tag service è creato e aperto tramite una specifica chiave, ma questa è diversa dal tag descriptor associato, inoltre il descrittore che individua il servizio deve essere univoco. Per scelta implementativa il limite dei tag service è di 256. I tag descriptor sono contenuti in una lista collegata: ogni volta che si crea un tag service viene rimosso dalla lista il primo descrittore presente in questa e ogni volta che viene rimosso un tag service il corrispettivo descrittore viene reinserito in coda alla lista, tornando così disponibile per nuovi tag service.

3.3 RCU

RCU è un meccanismo di sincronizzazione che è stato aggiunto dal kernel Linux 2.5 ed è ottimizzato per i sistemi read intensive. La read-copy-update di fatto evita l'uso di primitive di sincronizzazione bloccanti rendendo il sottosistema implementato più scalabile: ogni volta che un thread sta inserendo o cancellando elementi di strutture dati condivisa , tutti i thread lettori hanno la "garanzia" di vedere e attraversare sia la struttura precedente che quella nuova, evitando così incongruenze (ad esempio, dereferenziazione di puntatori nulli). Le API disponibili che sono state utilizzate per sfruttare tali meccanismi sono le seguenti:

—*rcu_read_lock()* : utilizzata da un lettore per informare lo scrittore che il lettore sta entrando in una sezione critica lato lettura RCU. È garantito che qualsiasi struttura dati protetta da RCU a cui si accede durante una sezione critica lato lettura RCU non verrà recuperata per l'intera durata di quella sezione critica.

—*rcu_read_unlock()* : utilizzata da un lettore per informare lo scrittore che il lettore è uscito da una sezione critica lato lettura RCU.

— *synchronize_rcu()* : Segna la fine del codice di aggiornamento e l'inizio del recupero. Lo fa bloccando fino al completamento di tutte le sezioni critiche lato lettura RCU preesistenti su tutte le CPU.

Riassumendo, la tipica sequenza di aggiornamento RCU è simile alla seguente:

(1) Rimuovere i puntatori ad una struttura dati, in modo che successivamente i lettori non possano ottenere un riferimento ad esso.

(2) Attende che tutti i lettori precedenti completino il lato di lettura della sezione critica RCU.

(3) A questo punto, non ci possono essere lettori che detengano riferimenti alla struttura dati, quindi ora può essere tranquillamente recuperato (ad esempio, kfree()).

3.4 Configurazione

Alla chiave privata IPC_PRIVATE è stato associato il valore 0, si ipotizza che ogni accesso in scrittura e lettura è preceduto da una open del tag_service, se il servizio è stato creato con chiave privata l'apertura fallisce. I permessi relativi al tag_service sono di due tipi: ACCESS_PRIVATE_TAG e ACCESS_FREE_TAG, rispettivamente di valore 0 e 1. Un accesso libero indica la possibilità di interagire in lettura e scrittura con il tag_service con qualsiasi thread, questo non è vero per un tag_service creato con permessi di tipo privato. Per un servizio con permesso ACCESS_PRIVATE_TAG le letture e scritture sono consentite solo ai thread dello stesso processo che ha creato il tag_service.

4. STRUTTURE

```c
//message
typedef struct msg{
    char *mex;
    unsigned msg_len;
    unsigned readers;

} msg;

//thread_data
typedef struct thread_data{
    struct list_head list;
    struct task_struct *t;
    struct msg **msg_ptr_addr;
}thread_data;

//levels
typedef struct tag_lvl{
    int id;
    int flag;
    struct msg *message;
    rwlock_t lock;
    unsigned sleepers; //numero di readers in attesa
    wait_queue_head_t wq;
    struct list_head reader_sleepers; //thread reader in attesa
}tag_lvl;


//tag descriptor
typedef struct my_td_list{
     struct list_head list;     //linux kernel list implementation
     int data;
} my_td_list;

//tag service
typedef struct tag_service{
    int key;
    int tag_descriptor;
    int permission;
    int owner;
    rwlock_t lock;
    struct tag_lvl lvl[MAX_LEVELS];
    struct hlist_node node;
}tag_service;
```

Fig. 1. strutture in main.h

Come è possibile osservare sono state definite 4 strutture principali:

—**struct tag_service**: è la stuttura principale che mantiene le informazioni relative al tag service, in particolar modo abbiamo il riferimento ai livelli del tag service e il descrittore corrispondente;

—**la struttura tag_lvl**: è la struttura di riferimento del livello di tag service, contiene la lista dei thread in attesa su quel livello, il numero dei thread "sleeper" e un puntatore alla struttura del messaggio in cui dovrà scrivere il thread scrittore;

— **struct thread_data**: è la struttura che contiene i riferimenti al thread in attesa, contiene quindi il task_struct del thread e l'indirizzo alla struttura del messaggio;

—**struct msg**: questa struttura serve per tener traccia dei reader che ancora devono leggere il messaggio, contiene quindi il numero di lettori in attesa e le informazioni relative al messaggio opportunamente indicate dal thread scrittore.

La struttura del messaggio viene allocata nel momento in cui un reader vuole leggere un messaggio in un livello del tag_service selezionato tramite tag_descriptor. Quando si scrive, nella struct msg associata al livello selzionato dallo scrittore si inizializza readers con il valore di sleepers presente nella struttura tag_lvl; ogni thread svegliato che legge il messaggio effettua una add-and-fetch atomica su readers per decrementarlo; il thread che dalla fetch riceve il valore 0 dealloca il messaggio.
Ogni thread reader che va a dormire alloca una struct thread_data dentro cui in "thread_data.msg_ptr_addr" posta l'indirizzo dove si aspetta di trovare al risveglio il messaggio. Bisogna fare attenzione al fatto che un thread potrebbe svegliarsi e scollegarsi dalla lista degli sleeper prima che un messaggio arrivi. Per segnalare questo quando il thread lettore si è svegliato per un interrupt modifica thread_data.msg_ptr_addr e lo fa puntare a NULL.
L'onere di deallocare le struct thread_data spetta al thread che scrive.
Si svegliano i singoli thread usando i task_struct presenti nella struttura thread_data per evitare di svegliare thread arrivati dopo che la scrittura del messaggio fosse iniziata. Il risveglio viene fatto in maniera selettiva usando la wake_up_process( ) passando come argomento il task_struct del thread sleeper in coda.

5. SYSTEM CALL

5.1 Tag_get

La chiamata di sistema Tag_get si divide in due casi: il primo caso riguarda la richiesta di apertura di tag_service da parte di un utente, il secondo caso riguarda l'istanziazione vera e propria del servizio specificato. Per la **open** semplicemente viene controllato il parametro relativo alla chiave e se questo è un **IPC_PRIVATE** viene impedita l'apertura, perchè il valore IPC_PRIVATE deve essere utilizzato per creare un'istanza del servizio in modo tale che non possa essere riaperto da questa stessa chiamata di sistema. Dopo aver effettuato il controllo, la tag_get relativa alla open restituirà il tag descriptor del servizio desiderato se esiste. Per quanto riguarda la **creazione** del tag_service per prima cosa si controlla che la chiave indicata dall'utente non esista, si controlla che ci sia un tag descriptor disponibile tra i **256** definiti nella fase di inizializzazione e infine se tutto è andato a buon fine si alloca il tag_service. Anche in tal caso se la creazione del servizio ha avuto successo si restituisce il tag_descriptor associato.

5.2 Tag_send

L'obiettivo principale di questa chiamata di sistema è quello di prendere a livello user un messaggio e "inserirlo" all'interno di un livello di tag_service specificato. Una volta individuati il tag_service e il livello indicato dall'utente, tramite **copy_from_user** viene copiato il messaggio all'interno di un buffer kernel e viene controllato il numero di sleeper, ovvero thread reader in attesa di un messaggio per il livello corrente. Se il numero di sleeper è pari a 0 non verrà consegnato il messaggio, altrimenti si procederà con il risveglio dei thread. Prima del *wake_up_process()* il messaggio verrà posto all'interno della stuttura dati msg a cui il livello corrente

fa riferimento. I thread reader vengono svegliati in maniera selettiva specificando il task_struct contenuto nella struttura thread_data accodata nella lista specifica del livello. Una volta terminata la fase di risveglio dei reader, il thread writer si occuperà di deallocare le strutture dati thread_data accodate per il livello specifico.

5.3 Tag_receive

Questa system call è implementata con l'obiettivo di permettere ad uno o più thread reader di sottoscriversi ad un livello di uno specifico tag service indicato dal lato user. Quando un reader si sottoscrive ad un livello viene allocata una struttura thread_data contenente il task_struct del thread reader e il riferimento alla struttura dati message dove dovrà essere scritto il messaggio. La struttura thread_data verrà accodata nella lista dei reader in attesa, presente nel livello di tag_service desiderato.
Se il reader è il primo thread che si sottoscrive al livello del tag_service, allora è suo compito allocare anche la struct msg per il messaggio di destinazione. Una volta individuato il livello del tag_service scelto e allocate le struttre dati necessarie, il thread reader si pone in attesa in una waitqueue specifica del livello del tag_service. Quando le condizioni di risveglio sono soddisfatte allora tramite **copy_to_user** il reader consegnerà il messaggio presente al livello in cui è sottoscritto all'utente. Naturalmente viene decrementato il numero di thread in attesa per lo specifico livello e se il reader è l'ultimo thread risvegliato (la fetch su sleepers restituisce valore 0) allora viene deallocata la struttura del messaggio. Se il thread in attesa è stato risvegliato a causa di un segnale allora il puntatore al messaggio sarà posto a NULL.

5.4 Tag_ctl

La chiamata di sistema Tag_ctl assume un comportamento diverso a seconda del parametro *command* specificato dall'utente. Se viene indicato come comando quello della **REMOVE** allora si desidera rimuovere il tag_service indicato dall'utente. La deallocazione del servizio è possibile solo se non ci sono thread reader presenti in tutti i livelli del tag_service, quindi se almeno in un livello ci sono degli sleeper allora la rimozione non potrà avvenire. Se invece il controllo su tutti i livelli restituisce 0 sleeper allora si può procedere con la deallocazione della struttura dati tag_service relativa al tag_descriptor passato dall'utente. Viene recuperato il descrittore allocando nuovamente un nuovo nodo tag_descriptor da inserire in coda ai descrittori disponibili. Nel caso in cui l'utente indica come comando la **AWAKE_ALL** allora, una volta individuato il tag_service associato al descrittore passato come parametro, tramite una *wake_up_all()* verranno risvegliati tutti i thread sleeper presenti nella wait queue di ogni livello inviando un messaggio standard di risveglio. Una volta risvegliati tutti i thread reader per ogni livello, si procede con la deallocazione dei thread_data come fatto nella chiamata di sistema tag_send.

5.5 Note

Per gestire una rimozione sicura del tag_service si utilizzano i **RW_lock**: quando un reader vuole "interagire" con un tag_service specificato dall'utente la prima cosa che viene effettuata è un controllo *try_read_lock()* sul lock del tag_service. Se la try_read_lock fallisce vuol dire che un thread sta cercando di rimuove il tag_service quindi il reader deve necessariamente fallire. Dall'altro lato infatti, quando si effettua una REMOVE tag per una chiamata di sistema Tag_ctl, la rimozione viene effettuata all'interno di una sezione critica delimitata con *write_lock()* sul lock del tag_service.

6. DEVICE DRIVER

Come richiesto dalla traccia progettuale è stato implentato un driver che permetta di mostrare all'utente lo stato corrente dei tag service presenti nel sistema. Sono state definite quindi le operazioni di apertura, rimozione e lettura del driver. Una volta identificato il major number disponbile per il device driver è possibile definire più istanze interagendo con thread diversi semplicemente passando un minor diverso per ognuno. Quando la funzione di lettura viene invocata il driver restituisce allo user una sorta di snapshoot del sistema, mostrando riga per riga, per ogni tag sevice e per ogni livello di tag service, le seguenti informazioni: Tag-key, Tag-creator, Tag-level, Waiting-thread. Sono stati definiti massimo 8 minor per il device e per la gestione della concorrenza in lettura è stato definito uno spinlock.

7. TEST

Al fine di valutare il corretto funzionamento del sottosistema sono stati implementati i seguenti test presenti nella directory *user* :

—multi_read_write_level.c: in questo test si verifica la capictà del sottosistema nel gestire la concorrenza tra più writer e reader su 32 livelli di uno stesso tag_service. I writer si occupano di inoltrare il messaggio che dovrà essere letto dai reader, i writer che troveranno i livello bloccato da un altro scrittore possono non trovare più lettori per quel livello perchè questi saranno già stati "risvegliati" alla consegna del messaggio del writer "vincitore". All'inizio dell'esecuzione viene creato il tag_service se non già esistente, se esiste viene semplicemente aperto. Alla fine dell'esecuzione il tag_service viene opportunamente rimosso. Non sono stati testati in questo user code i casi con la chiave PRIVATE_K e il permesso di accesso "privato".

—reader.c e writer.c: vengono passate in ingresso rispettivamente i numeri di system call del tag_get , del tag_receive e tag_send individuati in fase di inizializzazione del modulo: ./reader 134 177 , ./writer 134 174. Il test reader per prima cosa crea un tag_service con chiave e permessi opportuni (nel caso testato chiave diversa da IPC_PRIVATE e permessi ACCESS_FREE ovvero accessibile da tutti). Una volta creato il tag_service il programma chiama la system call tag_receive passando in ingresso: il tag_descriptor(restituito alla creazione), il livello desiderato, il buffer in cui deve essere inserito il messaggio e la size del buffer. Il reader va dunque in attesa fin quando il messaggio nel livello non viene scritto oppure arrivi un segnale di interruzione (opportunamente gestito). Il wirter, invece apre il tag_service desiderato, passando i parametri scelti e chiama la system call tag_send passando in ingresso il descrittore del tag, il livello in cui scrivere e il messaggio con relativa size. Se il writer è stato chiamato prima del reader il messaggio non verrà consegnato e verrà indicata l'assenza di sleeper, viceversa il reader leggerà il messaggio consegnato dal writer se i parmetri sono stati passati in modo opportuno. In questi test non vengono rimossi i tag_service che vengono creati.

—driver_prova.c : è il programma che testa il corretto funzionamento del drive. Sono necessari in input il nome del device, in questo caso /dev/miodev, il major number del device, individuato nella fase di istallazione del modulo e il numero di minor con cui si vuole operare.
Esempio : *sudo ./driver_prova /dev/miodev 243 3 >prova.txt*.
Il test effettua l'apertura del device, la creazione di 256 tag_service e la read del device che restituisce correttamente lo stato corrente del sottosistema. Alla fine del test tutti i tag service

creati saranno rimossi tramite la system call tag_get con command remove. Sono state eseguite prove con 1 minor, 4 minor, 8 minor e 10 minor. Nell'ultimo caso dato che il massimo minor per il device è definito a 8, 2 thread falliranno in accordo con le aspettative.

—wake_up_all.c: vengono creati 256 tag_service con accesso "libero" e vengono definiti 2 reader per ogni livello di tag_service. Una volta che i thread lettori entrano in attesa viene chiamata la wake_up_all al fine di risvegliarli tutti. I reader che vengono svegliati dalla system call tag_ctl con comando di risveglio, leggono e stampano a schermo il messaggio standard "Mi sono svegliato per colpa della wake up". Quando la wake_up_all ha concluso il suo lavoro viene eseguita una remove tag che rimuove i tag_service creati precedentemente.

—remove_tag_service.c: in questo test vengono creati 256 tag_service e successivamente rimossi uno ad uno. Se ci sono thread reader in attesa di messaggio in un livello del tag_service la rimozione non andrà a buon fine.

—reader_complete.c: rappresenta un test customizzato del reader. In questo programma è possibile specificare la chiave che può essere privata o no, i permessi di accesso, e il livello a cui è interessato il thread reader. Non viene testata la concorrenza in questo esempio, ma utilizzando anche le semplici versioni di reader/writer si può osseravere il comportamento in casi di scrittura e/o lettura concorrenti. In questo test vengono sempre rimossi i tag_service che sono stati creati.

Dai test è emerso che nonostante sia possibile una probabilità di collisione tra i thread questa risulta molto bassa. Anche nel caso di stress test interagendo con tutti i 256 tag_service, tutti e 32 i livelli e più thread readr/writer i test indicano un buon comportamento e una buona scalabilità del sottosistema.

8. REFERENCES

[1] Generic hash table:
`https://lwnnet/Articles/510202/`

[2] Linux kernel source code:
`https://elixir.bootlin.com/linux/latest/source`

[3] RCU:
`https://lwn.net/Articles/262464/`
`https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html`

[4] ListRCU:
`https://www.kernel.org/doc/Documentation/RCU/listRCU.txt`

[5] Rw_lock:
`https://www.kernel.org/doc/html/latest/locking/spinlocks.html`