Relazione progetto Sistemi Operativi Avanzati 2020-2021

Giorgia Marchesi

Abstract— **La possibilità di compilare parti del kernel Linux sotto forma di modulo, anche a run-time, permette agli sviluppatori di mantenere il sistema sempre aggiornato e diinteragire e sperimentare nuove soluzioni. L'obiettivo del progetto è quello di implementare un sottosistema del Kernel Linux che consenta lo scambio di messaggi tra thread tramite la realizzazione di system call.**

Parole chiave— **Kernel Linux, Modules, Hashtable, RCU, rw_lock.**

I. Introduzione

L'obiettivo di tale progetto è quello di implementare un sottosistema del Kernel Linux che consenta lo scambio di messaggi tra thread. In particolare, si richiede l'implementazione di 4 system call che consentano lo scambio e la gestione di messaggi tra servizi denominati *tag*. Il sottosistema deve occuparsi della gestione di 256 tags ciascuno formato da 32 livelli. I thread si sottoscrivono ad uno specifico livello di un determinato tag per inviare/ricevere messaggi. Inoltre, viene offerto un device driver per verificare qual è lo stato attuale del servizio fornendo informazioni quali la chiave del tag, il creatore del tag, i livelli associati ed i corrispettivi lettori in attesa di ricevere un messaggio su ciascun livello.
Nel proseguo di questa relazione verranno illustrate nel dettaglio le strutture dati utilizzate per la realizzazione del progetto, la logica d'esecuzione adottata per le chiamate di sistema, una breve spiegazione dei test effettuati ed una panoramica della struttura della repository Github.

II. Strutture dati e sincronizzazionne

In questo capitolo verranno discusse le strutture dati utilizzate per la gestione e lo storage dei TAG services e dei corrispettivi livelli. In particolare, sono stati adottate le Hashtable per i TAG services, un array per i livelli interni al TAG service ed una coda circolare per l'assegnazione dei tag descriptor.

A. Hashtable

Per un sistema performante, l'accesso in lettura e scrittura deve essere veloce e questo comporta la necessità di avere ricerche veloci. Per questo motivo è stata utilizzata una hashtable per mantenere traccia delle coppie key-tag descriptor poiché consente di effettuare inserimento, rimozione e lookup in O(1) quando la funzione hash distribuisce gli item in modo uniforme e ci sono più bucket di hash che oggetti da archiviare.
In particolare, è stata utilizzata la hashmap definita nel kernel Linux in `linux/hashtable.h` che propone un'interfaccia abbastanza semplice ed include meccanismi di sincronizzazione RCU. Questo la rende adatta ad un carico mostly-read e quindi al progetto, dal momento in cui avremo presumibilmente molte letture e scritture una tantum. Entrando più nel dettaglio, l'implementazione della tabella hash si basa sul concatenamento di elementi: ogni bucket nella tabella hash è un elenco collegato che conterrà tutti gli oggetti sottoposti ad hash nello stesso bucket. L'idea è che la tabella hash sia abbastanza grande da contenere tutti gli elementi in diversi bucket e la funzione hash dovrebbe essere abbastanza buona da distribuirli uniformemente, ma nel caso in cui si verifichi una collisione, gli elementi sono concatenati. Si è quindi cercato, nel codice, di ideare una buona funzione di hash che si assicurasse di inserire gli elementi in O(1) in ogni bucket.

B. FIFO Queue

Data una chiave quando viene creato un tag deve essere restituito un tag descriptor. Per l'assegnazione di quest'ultimo si è deciso di utilizzare una coda circolare FIFO di 256 elementi (tanti quanti i tag descriptor): ogni volta che si vuole creare un tag service viene estratto un elemento dall'inizio della coda che verrà assegnato come tag descriptor; quando un tag service viene eliminato, il corrispettivo tag descriptor viene immesso in fondo alla coda per poter essere poi riutilizzato. Nel momento in cui la coda è vuota, vuol dire che è stato raggiunto il massimo numero di tag service (i.e. 256) per cui non ne potranno essere creati di nuovi a meno che qualcun altro non venga prima rimosso.

C. Implementazione strutture dati

Le strutture dati definite nel codice sono quindi le seguenti:

```
1  struct message{
2      char *text;
3      unsigned msg_len;
4      unsigned readers;
5  }
6
7  struct thread_data{
8      struct list_head list;
9      struct task_struct *t;
10     struct message **msg_ptr_addr;
11 }
12
13 typedef struct level{
14     int id;
15     int flag;
16     struct message *msg;
17     rwlock_t lock;
18     unsigned sleepers;
19     wait_queue_head_t wait_queue;
20     struct list_head reader_sleepers;
21 }
22
```

```
23 typedef struct tag_service{
24     int key;
25     int tag_descriptor;
26     int permission;
27     int owner;
28     rwlock_t lock;
29     struct level levels[MAX_LEVELS];
30     struct hlist_node node;
31 }
32
33 typedef struct available_tag_descriptor{
34     int first, last, available_tags;
35     int capacity;
36     int *array;
37 }
```

Riepilogando, i tag service vengono mantenuti all'interno di una hashtable; i messaggi vengono mantenuti in un struct per tenere traccia dei lettori che ancora devono leggere il messaggio. Ogni thread che va a dormire alloca una struct thread_data in cui posta l'indirizzo dove si aspetta di trovare al risveglio il puntatore al messaggio (thread_data.msg_addr).

III. Sincronizzazione

A. RCU

Per l'aggiornamento e la rimozione sicura delle strutture condivise è stato utilizzato RCU. RCU è un meccanismo di sincronizzazione che è stato aggiunto a partire dal kernel Linux 2.5 ed è ottimizzato per i sistemi read intensive. L'idea alla base di RCU è quella di suddividere gli aggiornamenti in fasi di "rimozione/aggiornamento" e di "recupero risorse". Nella fase di rimozione vengono rimossi i puntatori ad una struttura dati, in modo che i successivi lettori non possano ottenere un riferimento ad essa e si attende che tutti i lettori precedenti abbiano completato la lettura RCU. A quel punto è possibile passare alla fase di "recupero" in cui la struttura dati può essere recuperata, ad esempio tramite una *kfree()*. Il fatto di attendere il termine dei lettori consente di avere una sincronizzazione leggera e, in alcuni casi, alcuna sincronizzazione, consentendo inserimento, rimozione e sostituzione atomica degli elementi in una struttura collegata senza interrompere i lettori che accederanno alle vecchie versioni.

Le API utilizzate nel progetto sono le seguenti:

- `rcu_read_lock()`: utilizzata da un lettore per informare il "recuperatore" che sta entrando in una sezione critica lato lettura RCU. In questo modo si garantische che la strtutura dati non verrà recuperata per l'intera durata di quella sezione critica;
- `rcu read unlock()`: utilizzata da un lettore per informare il "recuperatore" è uscito da una sezione critica lato lettura RCU;
- `synchronize_rcu()`: segna la fine della fase di aggiornamento e l'inizio di quella di recupero. In pratica, blocca il recupero fino al completamento di tutte le sezioni critiche lato lettura RCU preesistenti (non le successive alla synchronize_rcu());

B. RW_lock

Gli RW_lock sono la versione degli splinlock adatta ad un pattern di accesso a strutture condivise di tipo mostly-read. Consentono infatti a più lettori di trovarsi contemporaneamente nella stessa area critica, ma se qualcuno vuole modificare le variabili deve ottenere un blocco di scrittura esclusivo. In particolare, sono stati utilizzati per gestire la rimozione sicura del tag service: il reader, infatti, cerca innanzitutto di prendere il lock e se non ci riesce vuol dire che è in corso la rimozione del tag service da cui vuole leggere (nella remove viene infatti preso un lock in scrittura).

IV. Funzionalità del modulo

In questo capitolo verrà descritta la logica di implementazione delle chiamate di sistema introdotte nel progetto.

A. *int tag_get(int key, int command, int permission)*

La chiamata di sistema *tag get* si divide in due casi:

- TAG_OPEN: caso in cui viene richiesto di aprire un tag service. Innanzitutto si effettua un check-up della chiave in input: se la chiave è IPC_PRIVATE_KEY viene impedita l'apertura poiché l'istanza del tag service non può essere riaperta da questa stessa chiamata di sistema. Successivamente, verrà restituito il tag descriptor relativo alla chiave indicata.
- TAG_CREATE: caso in cui viene richiesto di creare un tag service. Per prima cosa si controlla che la chiave indicata dall'utente non esista già e che ci siano ancora tag descriptor disponibili. Solo dopo aver passato i controlli, viene creato il tag service e, in caso di successo, verrà restituito il tag descriptor associato;

B. *int tag_send(int tag, int level, char* mex, size_t size)*

Questa chiamata di sistema si occupa di gestire i thread scrittori. Quindi, preleva un messaggio a livello user e lo copia all'interno del livello del tag service specificato in input. Dunque, innanzitutto vengono individuati il tag service e poi il livello specificati e tramite una `copy_from_user` viene copiato il messaggio all'interno di un buffer kernel. In seguito, si verifica se ci sono lettori in attesa su quel livello monitorando il valore della variabile `level->sleepers`: se è uguale a 0, non ci sono lettori per cui lo scrittore esce ed il messaggio andrà perso; viceversa, se è maggiore di 0 il messaggio viene inserito nella struttura del messaggio corrispondente al livello e si procede al risveglio dei lettori in attesa. Si notic che i thread readers vengono svegliati in modo selettiva prelevando i thread lettori della lista `level->reader_sleepers` in modo tale da svegliare solo quelli che si sono messi in attesa prima dell'invio del messaggio. Una volta terminata questa fase, il thread scrittore si occuperà di deallocare le strutture dati thread data associate ai lettori in attesa per quello specifico livello.

C. *int tag_receive(int tag, int level, char* mex, size_t size)*

Questa system call permettere ad uno o più thread lettori di sottoscriversi al livello di un tag service e di mettersi in attesa di eventuali messaggi da parte degli scrittori. Quando un readers si sottoscrive ad un livello viene allocata una struttura `thread_data` contente il task struct del thread corrente ed il puntatore al messaggio che verrà scritto dal thread scrittore. La struttura thread_data verrà accodata nella lista dei readers in attesa per quello specifico livello del tag service. Il primo lettore che si sottoscrive al livello, allocherà la struttura del messaggio e poi, come anche i successivi lettori, si porrà in attesa nella `wait_queue` del livello. Quando verrà risvegliato dallo scrittore, tramite `copy_to_user` il reader consegnerà il messaggio lato user il messaggio presente nel livello cui si è sottoscritto. Ad ogni consegna, viene il numero di lettori in attesa (i.e. `level->sleepers`). Viceversa, se il thread in attesa viene risvegliato a causa di un segnale, il puntatore al messaggio sarà posto a *NULL*.

D. *int tag_ctl(int tag, int command)*

Come la *tag_get*, la *tag_ctl* assume un comportamento diverso a seconda del parametro command specificato dall'utente:

- AWAKE_ALL_TAG: si procede come nella *tag_send* con la differenza che verranno svegliati tutti i thread sleepers di tutti i livelli del tag service, inviando un messaggio custom di risveglio (*"AWAKE_ALL woke me up!"*).
- REMOVE_TAG: comando invocato nel caso in cui si vuole rimuovere il tag service indicato dall'utente. Tuttavia, la rimozione è possibile SOLO SE non ci sono thread sleepers in alcun livello del tag service. Si effettua quindi questa verifica che se ha esito negativo (non ci sono sleepers in attesa), consente di procedere alla deallocazione della struttura dati del tag service e di recuperare il tag descriptor (che verrà rimesso nella coda dei tag descriptor disponibili).

E. Device driver

Il device driver ha lo scopo di consentire all'utente di osservare lo stato corrente dei tag service presenti nel sistema. Sono state definite soltanto le operazioni di apertura, rimozione e lettura del driver (la scrittura infatti non è necessaria). Inoltre, il device driver è stato realizzato secondo un modello multi-instance per garantire l'accesso concorrente dei thread grazie a minor differenti. Quando la funzione di lettura viene invocata il driver si occupa di costruire lo stato a partire dai dati attuali, effettuando uno "snapshot" del sistema, e restituisce all'utente le seguenti informazioni per ogni livello di ogni tag service: Tag-key, Tag-creator, Tag-level, Waiting-thread. Essendo lo snapshot una operazione onerosa, negli sviluppi futuri si potrebbe sostituire con una versione *light-weight* in cui si potrebbe ridurre lo spazio dello snapshot effettuando solo la copia dei dati aventi waiting-thread diverso da zero.

V. Test effettuati

Il progetto è stato sviluppato e testato su Macchina Virtuale VMWare Fusion 12.1.1 su cui è stato eseguito il sistema operativo Ubuntu 18.04.4 a 64 bit con versione del Kernel Linux 4.15.0. I test finali sono stati eseguiti anche su kernel successive alla 4.17 per verificare anche in tal caso il corretto funzionamento. Nella directory `user` è possibile trovare i test attuati sul progetto:

- *test_create_open_remove.c*: si verifica il corretto funzionamento delle operazioni di create, open e remove sotto varie condizioni (es. assenza di tag descriptor disponibili, apertura di chiave IPC_PRIVATE, rimozione di un tag già rimosso ecc.);
- *test_wakeupall.c*: si verifica il corretto funzionamento della funzionalità di risveglio qualora ci siano thread lettori pendenti;
- *test_driver.c*: si verifica il corretto funzionamento del device driver acceduto in concorrenza da più thread che deve quindi restituire lo snapshot del sistema;
- *reader.c/writer.c*: test base per il funzionamento di lettura e scrittura su un tag service. Occorre prima lanciare il reader e poi il writer. Il reader crea un tag service e si mette in attesa di messaggi mentre il writer invia un messaggio sul tag service creato;
- *reader_custom.c/writer_custom.c*: come il test precedente verifica il funzionamento di lettura e scrittura su un tag service ma, in questo caso, è possibile definire parametri di input customizzati per lettore e scrittore atti al controllo di diverse condizioni. Inoltre, lo scrittore può immettere da tastiera i messaggi che desidera, i quali verranno visualizzati dal reader;
- *multi_reader_writer_multi_level.c*: test per verificare la sincronizzazione quando letture e scritture avvengono tra più livelli dello stesso tag service;
- *multi_reader_writer_multi_tag.c*: test in cui vengono lanciati più lettori e scrittori su tag differenti per verificare che non ci siano problemi di sincronizzazione;
- *multi_reader_writer_concurrency.c*: test per verificare la sincronizzazione quando letture e scritture avvengono tra tag services. Inoltre vengono nuovamente testate le funzioni di risveglio e rimozione dei tag service in presenza ed assenza di lettori pendenti;

VI. Struttura della repository

Link alla repository:

`https://github.com/giomarc/ProgettoSOA`

Per brevità, nello schema non è stata riportata né la cartella `user`, in cui sono presenti i codici per effettuare i test, nè la cartella `test_results`, in

cui si possono visionare i risultati prodotti dai test all'interno di file *.txt*.