Relazione per il progetto di Sistemi operativi avanzati

Tag based data exchange - Giacomo Lorenzo Rossi - 0292400

Aggiornamento: Correzioni alle criticità riscontrate

I problemi in questione erano:

- Il problema **dell'invio illecito di un messaggio dopo un cambio di permessi** dovuto alla concorrenza di una tag_send, tag_ctl (remove) e tag_get (create, only owner)
- Le **scarse performance di lettura del char device** dovute al fatto che la stringa restituita dalla read del driver era ricostruita ad ogni sua chiamata

Le soluzioni adottate sono rispettivamente:

- Correzione nella sincronizzazione della tag_send. Per verificarne la correttezza ho aggiunto un test `change_permission_during_send_test9`() in user/tag_ctl_test.c, che simula i tre casi possibili di concorrenza tra tag_send, REMOVE_TAG e CREATE_TAG con ONLY_OWNER.
- Aggiunta di un buffer su cui salvare la stringa corrente, che viene modificato ogni volta che cambia il numero di thread in attesa oppure quando viene creato il tag service. Per migliorare le prestazioni, ho utilizzato una sincronizzazione RCU per poter avere nella concorrenza 1 writer e diversi readers che eseguono in contemporanea. Le performance sono migliorate di due ordini di grandezza rispetto all'implementazione precedente (vedi `chrdev_read_performance_test7(10000)` in `user/tag_receive_test.c`). Inoltre, ho implementato un ulteriore programma per verificare i byte effettivamente letti in concorrenza di multipli scrittori e lettori `(chrdev_rw_test8(5)` in `user/tag_receive_test.c)`

Entrambe le modifiche sono state testate sia su kernel 5.4 che su kernel 3.13.

Introduzione

Il progetto consiste in un sottosistema del kernel Linux che permette di scambiare messaggi tra thread. Il modulo kernel aggiunge nelle prime 4 posizioni libere della system call table altrettante system call e registra un device driver che permette di controllare lo stato del sottosistema.

Il sottosistema gestisce MAX_TAG_SERVICE=**512** entità chiamate tag_service, ognuna suddivisa in MAX_LEVELS=**32** livelli, utilizzabili tramite le system call, con cui è possibile inviare o ricevere messaggi diversi su ciascun livello. La dimensione massima dei messaggi è di MAX_MESSAGE_SIZE=**4096** byte, compreso il carattere di terminazione stringa. Nella mia implementazione ho deciso di utilizzare un char device file read-only per ciascun tag_service istanziato, che permette di controllare lo stato dei 32 livelli del singolo tag_service.

Il progetto è formato da un **unico** modulo kernel, suddiviso in svariati file, comprensivi di headers, per una migliore modularità e riuso del codice.

Ricerca della system call table e aggiunta delle system calls

La ricerca della tabella delle system call è stata implementata in modo molto simile al programma usctm.c visto a lezione, ma con alcune modifiche. Innanzitutto non è più un modulo separato, bensì una libreria di funzioni montata nell'unico modulo di questo progetto. Sono perciò state eliminate le funzioni per l'inizializzazione e la pulizia del modulo. Inoltre sono stati fatti **ulteriori controlli nella funzione validate_page(): sia sulle entry sys_ni_syscall**, sia sui puntatori a funzione che **precedono la prima sys_ni_syscall**, per ridurre la probabilità di falsi positivi durante la ricerca della sys_call_table. Inoltre validate_page chiede come parametri dei puntatori che vengono inizializzati con i puntatori della sys_call_table e di sys_ni_syscall, per permettere al file tag_service.c di accedere a tali indirizzi.

Inoltre ho implementato **due ulteriori funzioni** richiamate dal file tag_service.c: install_syscall() e un install_syscall() che permettono di **installare o disinstallare in modo sicuro** dalla system call table le quattro system call richieste nelle posizioni richieste. Infine le posizioni delle entry libere con sys_ni_syscall vengono salvate in un array, reso disponibile all'utente perché salvato come parametro del modulo in /sys/modules/…/parameters/…

Le system call vere e proprie sono definite nel file tag_service.c : vengono installate nella funzione di inizializzazione start() e disinstallate allo smontaggio del modulo (nella funzione end()).

Implementazione delle system calls

All'inizio, ogni system call blocca lo smontaggio del modulo e lo sblocca prima del ritorno rispettivamente con try_module_get(THIS_MODULE) e module_put(THIS_MODULE).

int tag_get(int key, int command, int permission)

- ***Descrizione e assunzioni fatte***: Questa system call instanzia (CREATE_TAG) o apre (OPEN_TAG) il Tag Service associato alla **key** in base al valore di **command**. Il valore IPC_PRIVATE usato come **key** in congiunzione al comando CREATE_TAG permette di istanziare un nuovo tag_service, prendendo il primo spazio disponibile nell'array *all_tag_services*. Ogni volta che si usa CREATE_TAG e IPC_PRIVATE, quindi, viene creato un **nuovo tag service,** a patto che ci sia spazio nell'array. Se invece si prova ad aprire con IPC_PRIVATE la system call fallisce. **Tutti i thread/processi che hanno a disposizione il tag descriptor restituito con IPC_PRIVATE possono usarlo all'interno delle altre system call.** Se si utilizza una key numerica diversa da 0, invece, il tag descriptor corrispondente viene calcolato con una semplice funzione hash key % MAX_TAG_SERVICES. Se la **key** identifica un tag_service gia' esistente, il comando CREATE_TAG fallisce. Il valore di ritorno è il descrittore del TAG service, che **corrisponde con la posizione nell'array** *all_tag_services*. In caso di errore, viene restituito -1 e viene impostata la variabile errno di conseguenza. Inoltre le **permission** possono essere ONLY_OWNER o EVERYONE e indicano rispettivamente se il Tag service è stato **creato** per essere utilizzato da thread in esecuzione per conto dello stesso utente o da un utente qualsiasi. Quando si usa OPEN_TAG, le **permission** in input vengono ignorate e, a condizione che i permessi dell'utente sono corretti e che il tag_service e' stato già istanziato, viene semplicemente restituito il tag corrispondente alla **key** data in input. Ogni volta che si usa CREATE_TAG con successo, inoltre, viene creato un char device file che contiene le informazioni sui 32 livelli del tag_descriptor relativo al tag restituito da tag_get.

```
typedef struct my_ts_management {
    int first_free_entry;
    unsigned long remaining_entries;
    struct mutex access_lock[MAX_TAG_SERVICES];
    unsigned int major;
    tag_service **all_tag_services;
} ts_management;
```

```
typedef struct my_tag_service {
    tag_level *level;
    unsigned long thread_waiting_message_count;
    int key;
    int tag;
    int permission;
    int owner_euid;
    int owner_uid;
    int awake_request;
    struct rcu_head tag_rcu;
    int lazy_deleted;
} tag_service;

typedef struct my_ts_management {
    int first_free_entry;
    unsigned long remaining_entries;
    struct mutex access_lock[MAX_TAG_SERVICES];
    unsigned int major;
    tag_service **all_tag_services;
} ts_management;
```

- ***Dettagli implementativi:*** durante l'esecuzione di tag_get, indipendentemente dal comando, subito dopo aver calcolato il tag dalla chiave inserita, inizia una **sezione critica**, che termina prima della creazione del device file. Questo per evitare che due o più thread provino a richiedere lo stesso tag e a istanziarlo più volte. Durante un tag_get con comando CREATE_TAG vengono inizializzati i campi della struct *tag_service*

e dell'array di 32 struct *tag_level* (Vedi figura). Le permission sono state controllate verificando l'EUID del thread.

int tag_send(int tag, int livello, char* buffer, size_t size)

- ***Descrizione e assunzioni fatte***: Questo servizio consegna al Tag Service indicato dal **tag** il messaggio posto nel **buffer utente** lungo **size** bytes. Tutti i threads che sono correntemente in attesa di ricevere un messaggio sul corrispondente **livello** (e sullo stesso tag service), vengono svegliati non appena lo ricevono. I messaggi di dimensione 0 sono permessi. Se nessun thread era in ricezione quando la send ha inviato il messaggio, il messaggio viene perso. Se viene fornito in input un messaggio maggiore della dimensione massima, ho deciso di **troncare il messaggio alla massima lunghezza possibile** e impostare la variabile **errno** per avvertire l'utente dell'accaduto.
- ***Dettagli implementativi***: la system call tag_send è sincronizzata tramite il mutex corrispondente al **tag** descriptor. Per evitare che le performance degradino, la copy_from_user(), che potrebbe essere bloccante, viene fatta prima della sezione critica (*access_lock*) e all'interno della sezione utilizzo **memcpy()** per copiare il messaggio utente da un **buffer intermedio** nel membro **message** della struct tag_level corrispondente. Dopo aver copiato il messaggio, sveglio tutti gli eventuali thread in attesa con wake_up_all(). Inoltre per una **sincronizzazione più scalabile**, utilizzo la API **rcu_call()** per azzerare il contenuto del messaggio e reimpostare la condizione di risveglio (*message_ready*) una volta che tutti i thread in ricezione hanno finito di leggere il messaggio, ovvero quando termina il **grace period**.

int tag_receive(int tag, int livello, char* buffer, size_t size)

- ***Descrizione e assunzioni fatte***: tag_receive permette a un thread di chiamare l'operazione **bloccante** di ricezione del messaggio dal tag service corrispondente al **tag** e a un suo dato **livello**. L'operazione può fallire a causa dell'invio di un segnale Posix al thread mentre esso sta attendendo il messaggio. Se il buffer punta a memoria non mappata, la system call fallisce. Se il numero di byte da leggere è superiore al massimo vengono letti al massimo MAX_MESSAGE_SIZE bytes. Se però una tag_send invia meno byte del massimo, vengono letti solo quei byte, in base alla *size impostata da tag_send() nella struct tag_level*.
- ***Dettagli implementativi:*** l'attesa di questa system call è stata implementata tramite **wait_event_interruptible**, per svegliare il thread in uno dei tre seguenti casi:
    - Arriva un **segnale posix**
    - Viene svegliato da una tag_ctl con il comando AWAKE_ALL
    - Viene svegliato da una tag_send perchè viene inviato un messaggio al thread

```
wait_event_interruptible(the_queue,
                         condition: ts->level[level].message_ready == READY
                        || ts->awake_request == YES);
```

Le due condizioni di attesa vengono reimpostate NOT_READY e NO in maniera **deferred** tramite le funzioni rcu corrispondenti system call tag_ctl() e tag_send(). La tag_receive, infatti, usa una **sincronizzazione non bloccante basata su Read-Copy-Update** attraverso le chiamate rcu_read_lock/unlock(). La sezione critica RCU inizia subito dopo che il thread viene svegliato e dura finché non viene letto il messaggio o si gestiscono eventuali segnali o la condizione di risveglio per AWAKE_ALL. Ogni volta che un thread entra in attesa, inoltre, incrementa il contatore atomico dei thread nel livello e nel tag, rispettivamente per poter essere **lette dal device driver** e per poter **bloccare le eventuali rimozioni** del tag. Quando termina il lavoro, il contatore viene decrementato.

int tag_ctl(int tag, int command)

- ***Descrizione e assunzioni fatte:*** L'ultima system call implementata permette al chiamante di controllare il Tag service identificato dal **tag**, con uno dei seguenti **command**:

- AWAKE_ALL: la system call sveglia tutti i thread in attesa di un messaggio, indipendentemente dal livello su cui stanno attendendo
- REMOVE: la system call permette di rimuovere il Tag Service dal sistema. La rimozione del Tag Service fallisce se ci sono dei thread in attesa di ricevere messaggi su di esso, in almeno un livello. In tal caso restituirà -1 e scriverà un errore su **errno**

- **Dettagli implementativi:** entrambi i comandi vengono sincronizzati in modo bloccante tramite mutex.
  - AWAKE_ALL: per svegliare i thread esegue due operazioni: imposta a YES la condizione di risveglio e poi sveglia tutti i thread del tag scelto tramite wake_up_all. Per reimpostare a NO la condizione di risveglio ed **evitare che ulteriori receive vengano svegliate immediatamente utilizzo nuovamente la api rcu_call**, che oltre a questo **termina anche la sezione critica**.
  - REMOVE_ALL: per prima cosa imposta a YES il membro *lazy_deleted* del tag_service, così da **bloccare eventuali system call** (comprese ulteriori REMOVE_ALL) **che provano ad utilizzare lo stesso tag_service in un periodo immediatamente successivo**, ma permettendo a tutte le altre di terminare il loro lavoro. Subito dopo viene terminata la sezione critica e posso eliminare il tag service e il suo char device file corrispondente.

Implementazione del char device

KEY	EUID	LEVEL	#THREADS
205	1000	0	0
205	1000	1	0
205	1000	2	0
205	1000	3	0
205	1000	4	0
205	1000	5	4
205	1000	6	0
205	1000	7	0
205	1000	8	0

Descrizione e assunzioni fatte: Poiché è stato richiesto un device driver che permetta di leggere lo stato del sottosistema, ho deciso di implementarlo come **read-only**. Ogni device file corrisponde a un singolo tag_service e contiene **tante linee quanti sono i livelli**. Le linee di un singolo device file sono strutturate nel seguente modo: chiave del tag_service, EUID dell'utente proprietario, livello e numero di thread in attesa del livello. Ho scelto di non aggiungere il tag del tag_service perché essendo un valore interno al modulo kernel non dovrebbe essere disponibile all'utente.

Dettagli implementativi: Durante il montaggio del modulo, **inizializzo l'array dei cdev e dei mutex device_lock, separatamente dalla struct tag_service,** registro un nuovo device driver, alloco la regione di MAX_TAG_SERVICES minor numbers, creo un'unica class in /sys/class tramite class_create() e faccio in modo che **tutti i device file successivi siano accessibili da tutti gli utenti**, impostando il puntatore a funzione *ts_class->dev_uevent*.

Per creare un char device file prima inizializzo e aggiungo al sistema la struttura **cdev, salvandola nell'array nella posizione corrispondente al tag descriptor** (cioè al **minor**) e poi utilizzo device_create() passandogli la class creata in precedenza per creare il nodo di I/O in /dev. I **dati relativi a un particolare tag_service sono salvati nella funzione open del driver, sfruttando il membro** *filp->private_data,* così da poter essere utilizzati dalla read del driver. **La funzione write** del driver restituisce semplicemente il numero di byte da scrivere, ma **non scrive nulla**. La **funzione read,** invece, **costruisce la stringa da restituire a partire dalla struct** *tag_service*, senza usare ulteriori buffer per memorizzare i dati. Il tag descriptor corrisponde al MINOR number del char device file. Inoltre la read contiene una sezione critica per evitare che vengano eseguite troppe letture di seguito dai programmi cat o less.

Durante lo **smontaggio del modulo prima distruggo tutti i nodi di I/O eventualmente rimasti nel sistema**, e poi elimino la registrazione del driver e della class.

Test e verifica delle funzionalità

Per la verifica del software di livello kernel ho fatto uso di due diversi programmi utente: un programma che esegue **30 test** sulle singole funzionalità del device driver e delle system call, soprattutto a livello di **concorrenza**, e un programma interattivo che permette di utilizzare le system call e leggere il char device tramite linea di comando.

Il progetto è stato testato su kernel 5.4.x e su kernel 3.13.x e tutti i test vengono eseguiti con successo su entrambi i kernel.

Problemi riscontrati e soluzioni adottate

- Perché ho separato i cdev dalla struct del tag service? Perché se li avessi lasciati nel tag service, dopo una REMOVE_TAG, la cdev_put avrebbe causato un errore che avrebbe mandato in deadlock la system call, perché avrebbe provato a liberare un cdev deallocato, in maniera deferred.
- rcu_call non può essere chiamata troppo frequentemente, perciò ho terminato la sezione critica all'interno della funzione chiamata in modo deferred da call_rcu.

Istruzioni di Utilizzo

- Compilazione: make
- Installazione: make monta
- Disinstallazione: make smonta
- Compilazione ed Esecuzione dei test (senza sudo): make run
- Compilazione ed Esecuzione di tutti i test: make zudo_run
- Compilazione ed Esecuzione del programma utente: make user_app