Advanced Operating Systems and System Security

TAG-based data exchange

Cecilia Calavaro

Matricola 0287760

AA. 2020-2021

Introduzione

Lo scopo del sistema realizzato per questo progetto è quello di implementare un sottosistema del *kernel Linux* che permetta lo scambio di messaggi fra diversi threads. In particolare, esso è caratterizzato da quattro system calls che gestiscono il servizio di messaggistica, chiamate "*tag_get*", "*tag_send*", "*tag_receive*" e "*tag_ctl*". Viene inoltre implementato un *char device driver* per consentire di visualizzare alcuni aspetti dello stato corrente del sottosistema. Nella presente relazione, per evitare ridondanze, si è scelto di non riportare la specifica del progetto. Di seguito verrà illustrata l'architettura del sistema e le scelte progettuali dietro ad essa, i dettagli implementativi delle system calls, com'è stata gestita la concorrenza fra più threads nell'accesso a risorse condivise, l'implementazione del device driver ed i test eseguiti sulle funzionalità del sistema.

Architettura e Scelte Progettuali

Come si può vedere dalla repository GitHub (*https://github.com/ceciliacal/SOA*), il progetto è stato strutturato nel seguente modo:

- Nella repository *SOA* sono presenti i files:
  - *Usctm.c*, per montare il modulo kernel ed effettuare l'hacking della system call table, in modo da trovare delle entries libere per l'inserimento delle quattro system calls del sottosistema del kernel;
  - *systemCalls.c*, in cui sono implementate le system calls;
  - *load.sh*, uno script per l'avvio del sottosistema che si occupa di montare il modulo ed il device driver;
  - *Makefile*;
  - *unload.sh*, uno script per il cleanup del modulo e del device driver;

Inoltre, sono presenti le tre directories:

- *include*: contiene gli header files relativi a tutti i files .c presenti nel progetto;
- *lib*: contiene tutti i file .c per lo sviluppo del progetto, fra cui:
    - *deviceDriver.c*, per lo sviluppo del device driver;
    - *tagService.c*, che contiene delle funzioni di backend che vengono chiamate dalle system calls (presenti nel file *systemCalls.c*), le quali si occupano di sviluppare le vere e proprie funzionalità dell'intera architettura;
    - *utils.c*, che contiene delle routine d'appoggio comuni;
    - *vtpmo.c*, di supporto al file *usctm.c*.
- *user*: contiene i files di test, che verranno illustrati nel dettaglio più avanti nell'apposita sezione.

Si è deciso di strutturare il progetto nel seguente modo per avere una migliore modularità, per permettere di separare al meglio le funzionalità del sistema ed aver un'organizzazione migliore in termini di comprensibilità del codice sorgente.

Le attività del sistema sviluppato sono contenute nei files *systemCalls.c* e *tagService.c*, che sono quelli che verranno analizzati più nel dettaglio. In particolare, le funzioni sottostanti al servizio di messaggistica, presenti in *tagService.c*, verranno illustrate nella sezione chiamata "dettagli d'implementazione".

systemCalls.c

In questo files sono presenti le seguenti funzioni:

- *int tag_get(int key, int command, int permission):*

  In essa viene inizialmente controllato che i parametri di input passati dal lato utente siano conformi sia alle specifiche sia a quanto definito a livello software. In particolare, ci si riferisce al fatto che i valori *COMMAND* e *PERMISSION* debbano valere 0 o 1, per indicare se, rispettivamente, si sta invocando un'operazione di creazione o di apertura di un tag, e con quali permessi associati all'utente; inoltre, si controlla anche che un TAG che è stato creato secondo la politica "*IPC_PRIVATE*" (key=0) non venga riaperto, come stabilito da specifica. Viene poi effettuato un controllo sul parametro "command": se si tratta di *CREATE* viene chiamata la funzione *addTag (int key, kuid_t userId, int perm)*, altrimenti la funzione *openTag (int key, kuid_t currentUserId)* (entrambe definite in *tagService.c*).

- *int tag_send(int tag, int level, char* buffer, size_t size):*

  Dopo aver effettuato un controllo sulla validità del livello inserito come parametro, che deve essere necessariamente un intero compreso fra 1 e 32, e sulla dimensione (size) del messaggio che si vuole inviare, viene invocata la funzione *deliverMsg (int tagId, char* msg, int level, size_t size, kuid_t currentUserId)* sviluppata in *tagService.c*.

- *int tag_receive(int tag, int level, char* buffer, size_t size):*

  Si effettuano gli stessi controlli presenti in *tag_send* e poi chiamata la funzione *waitForMessage(int tag,int myLevel, char* buffer, size_t size,kuid_t uid)* sviluppata in *tagService.c*.

- *int tag_ctl(int tag, int command):*

  A seconda se il comando è *REMOVE* o *AWAKE_ALL* vengono invocate, rispettivamente, la funzione *removeTag (int tag, kuid_t currentUserId)* e *checkAwakeAll(int tag, kuid_t currentUserId))* sviluppate in *tagService.c*.

L'architettura del sistema si basa su due strutture dati chiamate ***tag_t*** e ***level_t***, entrambe definite nel file *tagService.h*, che rappresentano appunto un **TAG** ed un **livello**.

Nella struct *tag_t* sono presenti i metadati del tag, tra cui la chiave inserita lato utente e il descrittore generato lato kernel (chiamato *ID*) e in particolare un doppio puntatore ad un array di livelli composto da 32 elementi chiamato "*levels*", il numero di thread che sono in attesa di ricevere un messaggio su tutti i livelli del singolo tag ed un array di 32 spinlock, uno per ogni livello del tag.

Nella struct *level_t* sono contenuti il buffer del messaggio presente su quel livello, la wait queue per i threads in attesa su quel livello, il numero di tali threads e la condizione di risveglio dei thread dallo stato d'attesa, che consiste in un intero chiamato appunto "*wakeUpCondition*", che può avere valore pari a 0 o ad 1. Nel caso in cui sia pari ad 1 la condizione è verificata e quindi i threads, quando viene invocata una funzione che li risveglia (quest'aspetto verrà illustrato più dettagliatamente in seguito), possono uscire dallo stato d'attesa. In più, la struct *level_t* è caratterizzata da altri campi. Sono quindi presenti 32 wait queues per ogni tag, una per ogni livello.

Si è scelto di lavorare quindi con un solo array chiamato "*tagServiceArray*" di puntatori a struct *tag_t*, composto da 256 elementi che vengono allocati dinamicamente ogni volta che avviene la creazione di un nuovo tag, e con un array di 32 livelli che viene allocato interamente nel momento della creazione del tag stesso. Questa scelta deriva dal fatto che il numero massimo di tag presenti nel sistema ed il numero di livelli appartenenti a ciascuno di essi è fissa; pertanto, è stata scartata l'opzione di lavorare con strutture come le liste in quanto ritenute più utili in un contesto in cui la cardinalità degli elementi che la compongono non è nota. La scelta di utilizzare delle liste inoltre avrebbe comportato anche un *effort* maggiore in termini di operazioni da effettuare sui singoli componenti, le quali sarebbero state comunque molto frequenti in questo contesto applicativo.

Dettagli d'implementazione

Nel file ***tagService.c***, come già accennato, sono presenti le routine che implementano le funzionalità vere e proprie sottostanti all'intero sistema, che verranno illustrate qui di seguito suddividendole per system call.

- System call ***tag_get***:
    - **addTag***(int key, kuid_t userId, int perm)*: l'obbiettivo di questa funzione è quello di aggiungere un nuovo tag all'array *tagServiceArray* dichiarato globalmente nel file. Se il parametro key è diverso da 0, che rappresenta la chiave per i tag di tipo IPC_PRIVATE, si controlla se in *tagServiceArray* è già presente un tag avente stessa chiave: in tal caso, l'operazione fallisce e non viene creato alcun tag, in quanto ne è già presente uno con stessa chiave all'interno dell'array. Una volta superato questo controllo, viene allocato lo spazio necessario alla creazione della struct *tag_t* e viene allocato il relativo array di livelli. I singoli livelli vengono inizializzati e allocati chiamando la funzione *initLevels*, all'interno di cui vengono assegnati i diversi campi della struct *level_t* appena creata. Vengono quindi assegnati i metadati del tag ai diversi campi della struct e generato un ID univoco, ossia il descrittore del tag, utilizzando un contatore incrementale atomico tramite *__sync_fetch_and_add*. È stata ritenuta una scelta valida in quanto lo stesso meccanismo del contatore incrementale è utilizzato anche per generare gli indici di tabelle all'interno dei database relazionali. Inoltre, rappresenta un modo semplice e allo stesso tempo efficace per generare un descrittore unico per ogni tag. Infine, si scorre l'array *tagServiceArray*, si assegna il puntatore all'area di memoria del nuovo tag alla prima cella vuota che si trova e viene restituito l'ID del tag creato.
    - **openTag***(int key, kuid_t currentUserId)*: questa routine permette di effettuare il *retrieve* di un tag già presente nel sistema in base alla chiave. Prima di controllare se il tag è effettivamente presente in *tagServiceArray*, si controlla che l'utente che ha richiesto l'operazione sia lo stesso che ha creato il tag, qualora i permessi lo prevedano. Infine, se presente, viene restituito l'ID del tag avente la chiave passata come parametro.

- System call ***tag_send***:
    - **deliverMsg***(int tagId, char* msg, int level, size_t size, kuid_t currentUserId)*: questa funzione serve per far sì che il thread chiamante consegni il messaggio contenuto in "msg" (passato da *char* buffer* nella *tag_send*) ad il livello *level* del tag identificato da *tagId*. Viene quindi recuperato il tag dall'array *tagServiceArray* ed eventualmente controllato se l'utente ha i permessi necessari per svolgere quest'operazione. Questo controllo viene ripetuto in quanto già effettuato preventivamente nella funzione *openTag*, in modo da gestire uno scenario in cui un thread venga a conoscenza dell'ID di un tag che ha permessi riservati. In seguito, viene recuperato lo specifico livello d'interesse a partire dal puntatore all'array di livelli contenuto nel field del *tag_t*. Se il buffer del livello è già allocato significa che è già in corso un'altra operazione di

send, quindi l'operazione corrente fallisce e viene restituito -1. Altrimenti, viene allocato il buffer (il campo "*msg*" di *level_t*) in cui andare a scrivere il messaggio passato dall'utente tramite la funzione *__copy_from_user*. Subito dopo, viene impostata la condizione di risveglio dei threads a 1 e vengono risvegliati i thread in attesa presenti sulla wait queue del livello in considerazione tramite l'invocazione della funzione *wake_up_all*. Se le operazioni vanno a buon fine, la funzione *deliverMsg* restituisce il valore 0.

- System call ***tag_receive***:
    - **waitForMessage***(int tag,int myLevel, char* buffer, size_t size,kuid_t uid)*: al contrario della *deliverMsg*, questa funzione permette di mettere il thread che la invoca in attesa della consegna di un messaggio, in modo da restituirlo all'utente. Analogamente alla funzione precedente, viene recuperato il tag specificato dall'ID passato come parametro (int *tag*), controllati nuovamente i permessi, e recuperato il livello d'interesse. Il thread viene poi messo effettivamente in attesa di un messaggio: vengono quindi incrementati *due contatori*, ossia quello del numero di thread in attesa nell'intero tag, sia quello del numero di thread in attesa sul singolo livello, che quindi popolano la relativa wait queue. Viene poi chiamata la funzione *wait_event_interruptible*, che mette appunto il thread in stato di wait finché non si verifica la condizione di risveglio. Tale condizione è un intero presente come field nella struct *level_t* (è quindi un campo del livello) che deve essere pari a 1. Questo valore può essere impostato a 1, cioè un thread può risvegliarsi, o grazie alla funzione *deliverMsg* che implementa la system call *tag_send*, oppure alla funzione *checkAwakeAll* che implementa la system call *tag_ctl*. In entrambi i casi, infatti, si vogliono andare a risvegliare i threads in attesa sulle code di messaggi, e la condizione deve essere sempre vera affinché ciò avvenga. Possono presentarsi tre diversi scenari:
        - *wakeUpCondition* ha un valore pari ad **1** ed il buffer "**msg**" del livello è **pieno**: il risveglio è avvenuto a causa di una chiamata di sistema *tag_send*, poiché è in essa, in particolare nella funzione *deliverMsg*, che viene allocato il buffer e scritto da un thread;
        - *wakeUpCondition* ha un valore pari ad **1** ed il buffer "**msg**" del livello è **vuoto**: il risveglio è avvenuto a causa di una chiamata di sistema *tag_ctl* con comando *AWAKE_ALL*, poiché i thread sono ancora nella wait queue in attesa di un messaggio;
        - *wakeUpCondition* ha un valore pari a **0** ed il buffer "**msg**" del livello è **vuoto**: il risveglio è causato da un *segnale POSIX*, perché nessun thread ha impostato la condizione di risveglio a 1, ed il buffer è vuoto, perché se fosse stato pieno a causa di una *tag_send* anche la condizione di risveglio sarebbe stata pari a 1.

      Dopo che i thread sono stati risvegliati, viene letto il messaggio consegnato all'interno del buffer "msg" del livello tramite una *__copy_to_user* ed inserito nel buffer "*buffer*" passato come parametro dal lato utente. Vengono quindi

decrementati sia il contatore del numero dei thread in sleep all'interno dell'intero tag, sia quello relativo al livello stesso. Infine, l'ultimo thread receiver va a deallocare il buffer del livello e a impostare nuovamente la condizione di risveglio a 0. La funzione *waitForMessage* restituisce 0 in caso di successo.

- System call ***tag_ctl***:
  - **removeTag***(int tag, kuid_t currentUserId)*: questa funzione permette l'eliminazione di un tag dal sistema, deallocando la struct *tag_t* avente l'ID specificato come parametro e rimuovendo il suo puntatore dall'array *tagServiceArray*. Prima della rimozione vera e propria si verifica che non ci siano threads in attesa su nessun livello del tag considerato, andando a controllare il campo della struct *tag_t* che riporta tale informazione. Se sono presenti dei threads in qualche wait queue, l'operazione fallisce e restituisce -1, altrimenti 0.
  - **checkAwakeAll***(int tag, kuid_t currentUserId)*: lo scopo di questa routine è quello di andare a risvegliare ogni thread in attesa all'interno di un tag, indipendentemente dal livello. Viene quindi recuperato, in base all'ID, il tag dal *tagServiceArray*, effettuato un controllo sui permessi, ed infine, per ogni livello di quel tag, settata la condizione di risveglio a 1 e chiamata la funzione *wake_up_all* sulla relativa wait queue.

Gestione della concorrenza

In questa sezione viene illustrato come è stata gestito l'accesso concorrente a risorse condivise fra i diversi threads presenti nel sistema. Sono stati utilizzati:

- Un array chiamato "***tagLocks***" composto da **256 *rwlock_t***, uno in corrispondenza di ogni eventuale tag puntato dalle celle dell'array *tagServiceArray*. Si è scelto di utilizzare quest'approccio, dove ogni singolo tag ha un proprio lock, per aumentare il più possibile la concorrenza fra le operazioni e le performance, in modo da poter lavorare su tag diversi simultaneamente in maniera non bloccante. Si è ritenuto che utilizzare un singolo lock su tutti i tag fosse, per quanto più semplice da un punto di vista meramente implementativo, molto più degradante da un punto di vista prestazionale, in quanto avrebbe eventualmente bloccato, ad esempio, le operazioni su due tag differenti in modo parallelo.
  Inoltre, si è deciso di fare uso di *rwlock* rispetto agli *spinlock* in quanto le letture sono non bloccanti e solo la scrittura è esclusiva: in questa maniera, più threads possono leggere in parallelo su una stessa risorsa condivisa senza mettere in attesa gli altri lettori, cosa che invece sarebbe successa con gli spinlock.
- Un array di **32 *spinlock_t*** chiamato "***levelLocks***" e dichiarato nella struct *tag_t*. Ogni tag, quindi, possiede un array di spinlock dove ogni elemento corrisponde ad uno dei 32 livelli. Qui sono stati utilizzati spinlock perché i livelli vengono coinvolti solo nelle operazioni di send e receive, che effettuano delle scritture.

Device Driver

Il device driver è stato implementato come un *char device driver* e ha l'obbiettivo di mostrare lo stato corrente del sistema, in termini di chiavi dei TAG service presenti, user Id che ha creato tale TAG, numero di livello e numero di threads in attesa su quel livello.

Nel file ***deviceDriver.h*** sono state implementate le funzioni per la registrazione e la de registrazione del driver, che avvengono rispettivamente in fase d'inizializzazione e di cleanup del modulo. Inoltre, dal momento che il device è di tipo *read-only*, è stata realmente implementata solo l'operazione di read, che si occupa di recuperare i puntatori sia al *tagServiceArray*, sia all'array di rwlock_t *tagLocks* per poter scorrere i tag presenti nel sistema e recuperarne le informazioni. Vengono stampate solamente i tag che hanno effettivamente dei threads in attesa ed i relativi livelli.

Testing

Nella cartella user sono presenti i test sul modulo realizzato, che verranno descritti di seguito:

- ***test_tagGet.c***: contiene dei test relativi alla system call *tag_get* che riguardano diverse casistiche sulla combinazione dei parametri;
- ***test_openPermission.c***: questo test va effettuato dopo aver eseguito il file *test_tagGet.c* e cambiato utente (ad esempio eseguendolo con *sudo*) perché va a testare i permessi su dei tag creati nel file precedente;
- ***test_getCtl.c***: combina l'utilizzo delle system call *tag_get* e *tag_ctl*. Vengono creati 100 threads che chiamano *tag_get(0,CREATE,NO_PERMISSION)*, ossia che generano 100 nuovi tag, e poi creati dei threads che rimuovono 20 tag invocando *tag_ctl* con command *REMOVE*. Vengono poi generati altri threads che ricreano dei nuovi tags, i quali verranno inseriti al posto dei tag eliminati. Infine, viene chiamata nuovamente la syscall *tag_ctl* ed effettuata l'*AWAKE_ALL* su uno dei tag presenti nel sistema, che fallisce perché non vi sono thread in attesa in quanto non è stata chiamata la *tag_receive*;
- ***test1_sendRcv.c***: si occupa di testare dei threads receivers concorrenti che leggono un messaggio inviato sullo stesso livello di uno stesso tag;
- ***test2_sendRcv.c***: viene inizialmente creato un tag; il test si suddivide in due round per testare il comportamento del modulo quando si hanno molteplici send (15 thread senders) e numerosi receivers (100 threads) in attesa. Nel primo round vengono creati i primi 100 receivers e poi 15 thread che inviano messaggi diversi sullo stesso livello del tag creato inizialmente. Nel secondo round vi è una ripetizione dello scenario.
- ***test_tagCtl.c***: l'obbiettivo principale è quello di testare la system call *tag_ctl*. Vengono creati 30 threads receiver che vanno in attesa sui primi trenta livelli dello stesso tag. In seguito, due threads sender inviano messaggi sui primi due livelli dello stesso tag, andando così a risvegliare solo due dei 30 receiver, e mantenendo i rimanenti 28 threads ancora nelle wait queues. Viene quindi invocata la system call *tag_ctl* su tale tag, sia con il comando *REMOVE*, per testare che il tag non venga effettivamente rimosso perché vi sono dei threads in attesa nei suoi livelli, sia con il comando *AWAKE_ALL* per andarli a risvegliare, indipendentemente dal livello.

È possibile avere una visione più dettagliata andando a consultare direttamente i files d'interesse.