Advanced Operating Systems

2020 – 2021

TAG-based data exchange

Alessandro Amici

0280073

Introduzione

Nella presente relazione si descrive l'implementazione di un sottosistema del kernel Linux che permetta lo scambio di messaggi tra thread differenti. Dopo aver presentato le varie funzionalità richieste per il sottosistema, si presentano le strutture dati progettate ed implementate e successivamente si descrivono le quattro system calls richieste con particolare enfasi sulle principali scelte progettuali ed implementative adottate. Si passa poi a descrivere il device driver con il relativo meccanismo adottato per recuperare le informazioni richieste. Una sezione della presente relazione è inoltre dedicata alla questione della scalabilità e della sincronizzazione, esponendo le varie alternative che sono state valutate e giustificando quindi le scelte infine adottate. Si descrivono successivamente i test eseguiti ed infine le procedure per l'installazione del modulo e per l'esecuzione dei test.

Specifiche del sistema

Il sottosistema del *kernel Linux* da implementare deve permettere lo scambio di messaggi tra thread. Tale comunicazione avviene tramite *TAG services* ed ogni *TAG service* mantiene 32 livelli (*tags*). Come da specifica, devono essere gestiti almeno 256 TAG services e la dimensione massima di ciascun messaggio gestito deve essere di almeno 4KB. Il sottosistema da realizzare richiede l'implementazione e l'esposizione di quattro *system calls*, descritte di seguito.

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

  Questa system call istanzia o apre (a seconda del valore del command) il TAG service associato alla chiave key passata come parametro. Il parametro key può assumere il valore speciale IPC PRIVATE. Tale valore si deve usare come key per istanziare il TAG service in modo tale che non possa essere riaperto da questa stessa system call. Il valore di permission indica se il TAG service è creato per operazioni che possono essere eseguite solo da thread che eseguono per conto dello stesso utente che ha istanziato il TAG service, o da qualsiasi thread. Il valore di ritorno rappresenta il TAG service descriptor associato al TAG service istanziato o aperto. Tale valore è necessario per le altre tre system calls, per eseguire le operazioni di lettura e scrittura sul TAG service e per la gestione dello stesso. In caso di errore, il valore di ritorno è -1.

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

  Questa system call consegna al TAG service, avente il valore tag come descrittore, sul livello level il messaggio memorizzato in buffer ed avente una dimensione di size bytes. Messaggi di lunghezza pari a 0 sono ammessi. Tutti i thread che stanno attualmente aspettando un messaggio sul medesimo livello dello stesso TAG service devono essere risvegliati per l'esecuzione e devono ricevere il messaggio inviato. Il servizio di invio di messaggi non mantiene il log dei messaggi inviati, per cui se nessun receiver sta aspettando un messaggio sul medesimo livello dello stesso TAG service, il messaggio inviato viene scartato.

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

Questa system call implementa un servizio bloccante di ricezione di un messaggio che deve essere recuperato dal TAG service avente tag come descrittore al livello level. Tale operazione, oltre che per condizioni di errore, può fallire anche a seguito della consegna di un segnale Posix al thread mentre sta attendendo la consegna di un messaggio.

- int tag_ctl(int tag, int command)

  Questa system call permette al chiamante di gestire il TAG service avente tag come descrittore, a seconda del comando command passato come parametro. Il comando può essere AWAKE ALL (per risvegliare tutti i thread in attesa di messaggi sul TAG service, indipendentemente dal livello) o REMOVE (per rimuovere il TAG service dal sistema). Un TAG service non può essere rimosso se ci sono thread in attesa di messaggi su di esso.

Infine deve essere anche implementato e reso disponibile un device driver per controllare e visualizzare lo stato corrente del sottosistema, in particolare le attuali chiavi associate ai TAG services ed il numero di thread che sono in attesa di messaggi. Ogni linea del corrispondente device file deve essere strutturata come segue:

TAG-key TAG-creator TAG-level Waiting-threads

Progettazione ed implementazione delle strutture dati

Di seguito si descrivono le strutture dati progettate ed implementate per la realizzazione delle funzionalità del sistema.
Il nucleo del modulo è costituito da 4 strutture dati principali:

- int tag_descriptors_header_list[TAGS]
  È un array di interi che rappresenta la mappa dei tag service correntemente in uso. All'interno dell'array si possono trovare 3 tipi di valori:
  - -1 indica che il tag associato a quello slot non è in uso
  - 0 indica che lo slot è associato ad un tag privato
  - >0 indica la chiave associata al tag pubblico

  L'accesso a questa struttura dati condivisa viene regolato da uno spinlock tag_descriptors_header_lock: ciò rende l'accesso esclusivo, tuttavia essendo la risorsa consultata solo dalle syscalls tag_get e tag_ctl (solo nel caso di comando REMOVE) che dovrebbero avere una frequenza di utilizzo di molto inferiore a quella delle altre due, ed essendo le informazioni contenute all'interno dell'array molto ridotte (si parla infatti di valori interi), ciò non dovrebbe costituire un degrado per le prestazioni, soprattutto considerando che la struttura cresce in modo direttamente proporzionale al numero di tag del sistema.

- tag_descriptor_info *tag_descriptors_info_array[TAGS]
  È un array di puntatori a strutture dati tag_descriptor_info, ognuna delle quali mantiene un set di informazioni relative al tag associato.
  All'interno della struttura si trovano i campi:
  - int key: la chiave associata al tag service (può assumere gli stessi valori di una cella del tag_descriptors_header_list)
  - int perm: indica i permessi specificati per il tag durante la creazione (0 se accessibile da tutti; 1 se accessibile solo dai thread che eseguono per conto dell'utente che ha creato il tag)

- kuid_t euid: struttura atta a contenere l'id dell'utente che ha creato il tag

L'accesso ad ogni cella dell'array viene sincronizzato dal read_write lock corrispondente all'interno di un array di rwlock_t lock_array[TAGS].
L'array e le strutture da esso puntate vengono inizializzato ed allocate nella fase di startup del modulo all'interno della funzione int init_tag_service(void).

- tag *tags[TAGS]
  È un array di puntatori a strutture dati tag che costituiscono i tag service veri e propri. Ogni struttura tag è costituita da 2 campi:
  - tag_level *levels[LEVELS]: un array di puntatori a strutture dati di tipo tag_level che rappresentano i singoli livelli interni al tag in esame
  - spinlock_t levels_locks[LEVELS]: un array di spinlock_t, ognuno dei quali si occupa di sincronizzare l'accesso alla struttura dati tag_level del livello corrispondente.

  L'accesso ad ogni tag dell'array viene sincronizzato dal read_write lock corrispondente all'interno dell'array di rwlock_t lock_array[TAGS].
  L'array dei tag viene inizializzato interamente a null, in quanto essendo strutture dati pesanti, le si allocano ed inizializzano solo in caso di creazione del tag associato, così da limitare l'impronta di memoria del modulo e renderne l'esecuzione meno dispendiosa in termini di risorse richieste.

- struct tag_level
  È la struttura dati che rappresenta il singolo livello all'interno di un tag. Questa è costituita dai seguenti campi:

  - int threads_waiting: il numero di receivers correntemente in attesa di ricevere un messaggio su quel livello

- char *buffer: puntatore al buffer relativo al livello
- size_t size: dimensione del buffer di livello
- int awake: condizione di risveglio per i thread in waitqueue (se impostata ad “1” indica il risveglio)
- wait_queue_head_t wq: la waitqueue associate al livello

Queste struct vengono allocate ed inizializzate dai thread receivers: il primo receiver che vuole aspettare un messaggio su un livello e lo trova inesistente si occupa di allocarlo, ad eccezione del buffer che viene allocato ed inizializzato dal sender. La deallocazione e distruzione del livello e del buffer stesso, avviene ad opera dell’ultimo receiver risvegliatosi sul livello stesso.
Delegare i meccanismi di allocazione e deallocazione della struttura dati del livello ai sendere e receiver permette di mantenere al minimo la quantità di memoria richiesta dal modulo.

Per gestire la sincronizzazione sui livelli si è deciso di utilizzare degli spinlock_t poiché sia la send che la receive che la ctl vanno ad apportare delle modifiche alla struct tag_level che necessitano di atomicità.
Per gestire la sincronizzazione dei tag invece si sono scelti degli rwlock_t per aumentare il grado di parallelismo del modulo, garantendo piena concorrenza ad operazioni (send e receive) su livelli diversi.
Le operazioni che coinvolgono la creazione o distruzione di un intero tag (get e ctl) continuano invece ad essere pienamente esclusive ed atomiche grazie all’acquisizione dei write lock.

Inizializzazione e cleanup del modulo

Al momento del montaggio del modulo che implementa il sottosistema di scambio di messaggi, viene effettuato l'hacking della system call table. Tale meccanismo è compatibile anche con le ultime versioni del kernel Linux (5.x). Dopo aver individuato l'indirizzo di partenza della system call table, si individuano le prime quattro entries che puntano alla funzione sys ni syscall, per poter inserire le quattro system calls proprie del sistema implementato. Gli indici trovati nella system call table vengono memorizzati nel buffer circolare del kernel e posso essere recuperati tramite dmesg, per poterli utilizzare lato user. Sempre in fase di inizializzazione, vengono allocate le strutture necessarie al corretto funzionamento del sistema. Per quanto riguarda la fase di shutdown del modulo, come di consueto viene ripristinato il contenuto originale della system call table. Inoltre, vengono deallocate tutte le strutture dati utilizzate ancora attive. Altro aspetto che è stato tenuto in considerazione riguarda l'aggiornamento del contatore di utilizzo del modulo. In particolare, si è voluta prevenire la possibilità di smontare il modulo mentre è in utilizzo da parte di qualche thread. Ovviamente il decremento del contatore non si sarebbe potuto inserire nella fase di cleanup perchè in questo modo ogni volta che si fosse richiesto lo smontaggio del modulo, il relativo contatore di utilizzo sarebbe stato sempre maggiore di 0. Il decremento del contatore deve essere precedente all'invocazione della cleanup module. Si è pensato quindi di aggiornare tale contatore ad ogni accesso ed uscita di un thread che invoca una delle quattro system calls implementate. Dell'aggiornamento dello usage counter si è tenuto conto anche nell'implementazione del device driver, specificando nella struttura file operations come owner il modulo corrente.

Progettazione ed implementazione delle system calls

A seguire viene illustrata la logica di implementazione e realizzazione delle 4 system calls che costituiscono il nucleo del modulo.

- tag_get
  Inizialmente si controlla che i valori assunti dai parametri command e permission siano quelli illustrati nelle specifiche, quindi si controlla che la chiave inserita dall'utente sia un valore intero positivo, altrimenti si ritorna il valore di errore -1.
  Si distinguono quindi le 2 casistiche: chiave privata o chiave pubblica. Nel primo caso se il comando è OPEN si ritorna il codice di errore, se il comando è CREATE invece si procede a verificare che ci siano ancora slot disponibili per l'allocazione di un nuovo tag nella mappa dei tag liberi/occupati ed in caso affermativo si inserisce la chiave (0) all'interno dell'array e si procede ad allocare la struttura relativa al tag. Anche se per la ricerca dello slot libero, nel caso peggiore occorresse scorrere tutto l'array, non si è ritenuto che ciò fosse un problema o che inficiasse sulle prestazioni sia per la natura ridotta del numero di tag che si gestiscono, sia perché le operazioni di creazione dei tag sono molto meno frequenti di quelli di invio e ricezione dei messaggi.
  Nel caso di chiave pubblica, la creazione del tag avviene in modo analogo a quanto descritto sopra, con l'accortezza, mentre si scorre la lista per ricercare lo slot libero, anche di verificare che la chiave passata come parametro per la creazione del tag non sia già in uso: se già in uso si ritorna il codice di errore, altrimenti, se si trova lo slot libero, si procede ad allocare il nuovo tag con le relative info ed ad inserire nell'array dei tag liberi/in uso la chiave fornita.
  Nel caso invece di apertura di un tag già esistente, si verifica che la chiave fornita sia presente nell'array dei tag in uso, altrimenti si ritorna il codice di errore. Se la chiave è presente si procede a verificare i permessi dell'utente rispetto a quelli del tag: se l'accesso

è consentito viene ritornato il tag descriptor corrispondente, altrimenti si restituisce il codice di errore.
Sia sulla creazione che sull'apertura, il file descriptor che viene ritornato è proprio la posizione dell'array in cui è stato memorizzato quel tag: ogni tag occupa la stessa posizione all'interno di tutti e 3 gli array principali (quello dei tag liberi/occupati, delle informazioni dei tag e l'array delle strutture tag vere e proprie). Con questo espediente, è possibile accedere in modo istantaneo a tutte le strutture relative allo stesso tag, quando si vanno ad utilizzare le altre 3 system calls.

- tag_send
  Inizialmente si controlla la size del buffer contenente il messaggio da inviare: se questa è inferiore ad 1 (buffer vuoto) o superiore alla massima dimensione consentita (MAXSIZE = 4096) si ritorna il codice di errore. Quindi si procede a controllare i permessi dell'utente rispetto a quelli del tag che vuole utilizzare. Se i permessi sono corretti si accede in modo esclusivo al livello desiderato: se il livello non è inizializzato significa che non ci sono receiver in ascolto, perciò la send viene scartata, ritornando comunque un valore di successo dell'operazione. Se il livello è presente ma il buffer associato non è nullo, si scarta ancora una volta il messaggio, in quanto tutti i thread in attesa di un messaggio su quel livello, hanno già ricevuto un altro messaggio prima di questo, che quindi non troverebbe alcun receiver disponibile: anche in questo caso si ritorna un valore di successo. Se invece il livello esiste ma il suo buffer è nullo, significa che ci sono receivers in attesa di questo messaggio che viene quindi copiato dal buffer utente a quello lato kernel. Se la copia va a buon fine si provvede ad impostare la condizione di risveglio per i thread in waitqueue ad 1 e si vanno a svegliare i suddetti thread, quindi si ritorna successo.

- tag_receive

  Analogamente alla send si controllano la size del buffer fornito ed i permessi dell'utente. Se tali controlli non falliscono si procede all'accesso al tag service e quindi al livello desiderato. Se il livello non è ancora stato allocato ed inizializzato significa che questo è il primo receiver per quel livello, quindi procederà lui stesso ad istanziare il livello, ad incrementare il contatore dei thread in attesa su di esso ed ad andare in sleep sulla waitqueue. Se altrimenti il livello è già presente, si limita ad aumentare il contatore dei thread in attesa sullo stesso ed a collocarsi sulla waitqueue. Al risveglio dalla waitqueu si controlla la condizione: se questa è falsa si è stati risvegliati a causa di un segnale posix, quindi si decrementa il contatore dei thread in attesa e si lascia il livello. Se la condizione risulta vera, si va a controllare la presenza del buffer contenente il messaggio: se il buffer è null ci si è risvegliati a causa di una awake all, quindi si decrementa il contatore e si abbandona il livello; se il buffer contiene un messaggio si è stati risvegliati dalla send, quindi si trasferisce il messaggio dal buffer kernel a quello utente proprio di ogni receiver, si decrementa il contatore dei thread in attesa e si lascia il livello. Ogni receiver prima di abbandonare il livello controlla di non essere l'ultimo receiver per quel livello: se così è, prima di andarsene, si occupa di deallocare il livello stesso.

- tag_ctl

  Ancora una volta si controllano i valori associati al comando ed i permessi dell'utente rispetto al tag su cui vuole operare. Quindi se il comando è AWAKE_ALL si procede ad accedere a tutti i livelli del tag presenti (non nulli), ad impostarne la condizione di risveglio della waitqueue ad "1" ed a triggerare il risveglio dei thread in attesa su quel livello.
  Se il comando è invece REMOVE, si acquisisce (come nella caso della creazione del tag) un lock esclusivo su tutto il tag (write_lock) così da

evitare il sopraggiungere di nuovi thread sul tag che potrebbero impedirne la rimozione. Quindi si procede a controllare i livelli del tag: se si trova anche un solo livello presente, significa che ci sono dei thread in attesa su quel tag, che quindi questo non può essere rimosso e l'operazione fallisce ritornando un errore. Se invece tutti i livelli sono nulli, non essendoci thread in attesa sul tag, si procede alla rimozione dello stesso: si acquisisce anche il lock esclusivo sull'array dei tag liberi/occupati e si procede a deallocare tutte le strutture dati del tag: la struttura del tag vero e proprio viene distrutta mentre la struttura relativa alle sue informazioni viene rinizializzata ai valori di default. Quindi si ripristina il valore di default (-1) anche nella cella dell'array dei tag liberi/occupati.

Progettazione ed implementazione del device driver

Si è deciso di implementare il device driver come un char device driver. In fase di inizializzazione del modulo, viene registrato il driver ed allocato il buffer necessario per l'header, che è unico per tutto il device driver, mentre viene deregistrato in fase di cleanup, quando si dealloca anche il buffer. Il corrispondente device file `e di tipo read-only in quanto serve unicamente per recuperare informazioni sul sistema e non per modificarle, permettendo così l'accesso in piena concorrenza alla risorsa.
Le operazioni di open, realease e write del device sono fittizie: tutta la logica viene implementata nella read. All'interno di quest'ultima operazione si accedono le strutture dati del modulo e da queste si ottengono le informazioni desiderate per ogni tag e per ogni livello in uso: in particolare viene prodotta una nuova riga per ogni nuovo livello in uso. Ogni riga viene concatenata alle informazioni ottenute in precedenza andando a formare un unico buffer finale, che viene poi trasferito dal buffer lato kernel del device al buffer lato utente che è stato fornito.

L’operazione di recupero delle informazioni è stata implementata nella read così da permettere l’aggiornamento dei dati in tempo reale: più read consecutive all’interno di una stessa sessione sul device driver sono così in grado di catturare i cambiamenti relativi allo stato del modulo.
Tale funzione di read gestisce anche la presenza di un offset passato come parametro, così da garantire che tutti i dati desiderati possano essere ottenuti.
Per limitare il più possibile l’allocazione di memoria da parte del device, ogni volta che si concatena una nuova informazione alle precedenti (e quindi si alloca un nuovo buffer), il buffer precedente viene prontamente deallocato, così che in ogni istante ci siano solo 2 buffer attivi per ogni read: il buffer relativo alle informazioni di una nuova riga ed il buffer contenete le informazioni ottenute fino a quel momento.

Installazione e rimozione del modulo

Per l’installazione del modulo e del device implementati, è possibile eseguire lo script load.sh mentre per la rimozione di entrambi è presente lo script unload.sh.
Entrambi gli script richiedono permessi di root.
Lo script di unload.sh si occupa anche di pulire l’ambiente di sviluppo da tutti i file necessari per il montaggio del modulo e del device che non costituiscono i file sorgenti del progetto.

Test eseguiti

Nella directory user sono riportati tutti i test che sono stati eseguiti già compilati. Di seguito se ne presenta una panoramica:

- test_tag_get.c

  Sorgente per testare tutte le possibili combinazioni dei parametri nell'uso della tag_get. Crea il massimo numero di tag possibili sia con chiavi private che pubbliche, sia accessibili al solo utente creatore del tag (permessi ristretti) che a tutti gli utenti (permessi estesi) e testa il funzionamento delle possibili combinazioni del comando open sui tag creati. Quindi rimuove i tag creati.

- test_tag_get_permissions.c

  Sorgente che permette di verificare il corretto funzionamento dei permessi. Questo file è stato eseguito con permessi di root così che l'apertura del tag precedentemente creato avvenisse ad opera di un utente diverso dal creatore del tag stesso: se il tag è stato creato con permessi ristretti al solo utente creatore entrambe le syscall "tag_get" per l'apertura del tag falliscono.

- test_tag_send_receive.c

  Sorgente per la verifica del corretto funzionamento delle syscall di invio e ricezione dei messaggi in un ambiente altamente concorrente (vengono testati 10 000 receiver e 1000 sender su 100 tag su svariati livelli). La funzione crea diversi tag e genera un elevato numero di receiver che mette in ascolto su vari livelli dei suddetti tag. Quindi genera un insieme di threads che ricoprano il ruolo di senders, inviando messaggi differenti in contenuto e lunghezza. Le creazioni di receiver e senders vengono ripetute nuovamente per creare un ambiente maggiormente dinamico ed assicurarsi che il modulo risponda correttamente se sottoposto a stress. Quindi distrugge i tag creati.

- test_tag_ctl.c

  Sorgente per testare la syscal tag_ctl. Verifica che sia la awake all che la remove falliscano in assenza del tag target, quindi crea il tag e verifica che la awake all abbia successo in assenza di receiver in attesa sul tag. Genera quindi un insieme di thread receiver che si mettono in ascolto su diversi livelli del tag: verifica che la rimozione del tag fallsica in quanto è in uso da parte di altri thread. Verifica il corretto funzionamento della awake all nel risvegliare tutti i thread in attesa, quindi verifica il corretto funzionamento della remove per la distruzione del tag stesso, ora non più occupato.

- test_device_driver.c

  Sorgente per testare l'utilizzo del device driver in modo concorrente e per testarne il corretto funzionamento (aggiornamento) in caso di read mulitple all'interno di un'unica sessione. La funzione crea 2 tags e genera un numero di receiver che mette in attesa di messaggi su vari livelli di questi tags. Genera quindi degli utilizzatori del device per leggere le informazioni del modulo. Effettua una send di un messaggio su alcuni livelli dei tag al fine di modificare le informazioni del tag stesso e quindi del modulo e di verificare che tali modifiche vengano osservate dagli utilizzatori del device driver. Utilizza quindi due awake all per risvegliare i thread rimasti in attesa sui due tag e verifica nuovamente che questa modifica dei dati relativi al modulo venga catturata dagli utilizzatori del device. Quindi distrugge i due tag.
  Il device può anche essere testato a run time con il comando bash “cat /dev/ tag_system_device”.

Maggiori informazioni sui singoli test sono riportate all’interno di ciascun file.