Relazione progetto
Sistemi Operativi Avanzati
2020/2021

Daniele Mariano
0294300

***Abstract*—Un modulo del kernel è fondamentalmente un file oggetto, cioè un frammento di codice eseguibile che fa riferimento a funzioni e variabili esterne, oltre a dichiarare le funzione e le variabili che lui stesso definisce.**

I moduli vengono quindi utilizzati dai programmatori per realizzare: device driver, filesystem driver, system call, network driver software, executable interpreter software.

La specifica del progetto richiede l'implementazione di un sottosistema del kernel Linux che permette lo scambio di messaggi tra threads.

1. Introduzione

La specifica del progetto richiede l'implementazione di un sottosistema del kernel Linux che permette lo scambio di messaggi tra threads. Il servizio espone 32 livelli (che abbiamo definito tag) di default guidati dalle seguenti chiamate di sistema:

- **int tag_get(int key, int command, int permission)**
  questa chiamata di sistema istanzia o apre il servizio TAG associato alla chiave a seconda del valore del comando. Il valore IPC_PRIVATE deve essere utilizzato per la chiave per creare un'istanza del servizio in modo tale che non possa essere riaperto da questa stessa chiamata di sistema. Il valore di ritorno dovrebbe indicare il descrittore del servizio TAG (-1 è l'errore di ritorno) per gestire le operazioni effettive sul servizio TAG. Inoltre, il valore dell'autorizzazione dovrebbe indicare se il servizio TAG viene creato per le operazioni dai thread che eseguono un programma per conto dello stesso utente che installa il servizio o da qualsiasi thread.
- **int tag_send(int tag, int level, char* buffer, size_t size)**
  questo servizio consegna il messaggio attualmente presente nel buffer all'indirizzo con tag come descrittore. Tutti i thread che sono attualmente in attesa di tale messaggio sul corrispondente livello dovrebbero essere ripresi per l'esecuzione e dovrebbero ricevere il messaggio (sono comunque consentiti messaggi di lunghezza zero). Il servizio non mantiene il registro dei messaggi che sono stati inviati e se nessun destinatario è in attesa del messaggio questo viene semplicemente scartato.
- **int tag_receive(int tag, int level, char* buffer, size_t size)**
  questo servizio permette ad un thread di chiamare l'operazione di bloccante di ricezione del messaggio da prelevare dal corrispondente tag descriptor ad un dato livello. L'operazione di ricezione può fallire a causa della consegna di un segnale Posix al thread mentre il thread è in attesa del messaggio.
- **int tag_ctl(int tag, int command)**
  questa chiamata di sistema permette al chiamante di controllare il servizio TAG, definito dal rispettivo descrittore tag, secondo un comando che può essere AWAKE_ALL (per svegliare tutti i thread in attesa di messaggi, indipendentemente dal livello), oppure REMOVE (per rimuovere il servizio TAG dal sistema). Non è possibile rimuovere un servizio TAG se sono presenti thread in attesa di messaggi.

Di default il sottosistema implementato può consentire la gestione di almeno 256 servizi TAG e la dimensione massima consentita per ogni messaggio è di 4KB.

Inoltre è stato sviluppato un device driver che permette di verificare lo stato corrente, ovvero le chiavi correnti del servizio TAG e il numero di thread attualmente in attesa di messaggi. Ogni riga del file del device corrispondente deve quindi essere strutturata come "TAG-chiave TAG creatore TAG-livello di attesa-thread".

2. Sviluppo

Per l'implementazione si è scelta una strategia a step:

1) è stato eseguito una sorta di "helloworld dei moduli kernel" in si è scritto e montato un modulo che quando montato eseguiva delle stampe sul ring buffer del kernel e stessa cosa faceva quando veniva smontato.
2) Viene aggiunto un makefile con delle rules affinchè il processo di mounting e unmounting dei moduli kernel fosse quanto più automatizzato, veloce e snello.
3) Viene poi preso ed eseguito il codice del prof. Quaglia che andava a trovare le entry della syscall table in cui avevamo sys_ni_syscall ovvero entry libere che puntavano a ni_syscall. In questo

modo vengono segnate le entry che possono essere hackerate dal programmatore ovvero in cui è possibile aggiungere delle nuove syscall. Per fare ciò il codice cerca la pagina in cui la syscall table inizia e quindi in cui cercare le n entry che puntano a ni_syscall da utilizzare per le nostre syscall hackerate. Quando vengono trovare le n aree di memoria che puntano a ni_syscall la ricerca può essere fermata e viene stampato nel log del kernel il risultato della nostra ricerca ovvero le entry della syscall tabel in cui abbiamo trovato le sys_ni_syscall. Eventualmente si può ripulite tutto e smontare il modulo.

4) Il codice sopra citato è stato poi modificato per essere adattato al nostro scopo, è stato aggiunto un array ed inserito una funzione che restituisce le quattro entry in cui possiamo inserire le quattro hacked syscall come da specifica, in modo da poterle utilizzare in un secondo momento. Una volta che la nostra ricerca ha avuto esito positivo, ovvero sono state trovate le quattro entry e sono stati salvati i relativi dati nell'apposita struttura, sono state create due dummy hacked syscall per testare il corretto funzinamento di tutti gli step precedenti. Per completezza, la prima eseguiva la stampa del parametro mentre la seconda eseguiva una semplice stampa della somma di due valori passati come parametri. Anche qui nel momento in cui smontiamo il modulo viene ripulito tutto.
5) Successivamente sono state sostituite le dummy hacked syscall con le nostre quattro vere syscall. Queste sono state scritte in un secondo file che viene incluso nel main file in cui abbiamo il frontend mockato delle syscall e che viene richiamato a livello user dal file user che esegue la demo delle syscall.
6) Poi è stato implementato il device drive. Quest'ultimo 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.
7) Infine sono stati scritti i test in cui sono state testate le operazioni di creazione del tag, la loro rimozione e lo scambio di messaggi tra tag aperti. Inoltre è stato scritto uno script bash per eliminare tutti i file prodotti dall'esecuzione per rieseguire i test più velocemente.

3. Architettura

Per quanto riguardano le scelte implementative sono stati rispettati i 32 livelli della specifica mentre i servizi TAG vengono fissati al valore di 256 tag e la dimensione del messaggio rispetta i 4KB.

I moduli che carichiamo nel kernel sono due: syscall_hacking e driver, a questi due moduli si affiancano le strutture dati tag e level a supporto e l'implementazione del frontend delle syscall a completare l'organizzazione dell'architettura.

3.1. Syscall_hacking

Il primo modulo che carichiamo nel kernel è quello implementato nella syscall_hacking. In questo modulo in cui viene eseguita la ricerca delle entry libere della syscall table.

Nella init del modulo si esegue l'allocazione dell'array in cui memorizziamo i risultati della syscall_number_finder() che si occupa proprio di trovare l'identificativo della entry libera da associare successivamente alle nostre hacked syscall, nel nostro caso le entry libere da trovare sono quattro (dato settato nella REQUIRED_SYS_NI_SYSCALL). Seguendo il codice del professore la syscall_number_finder() esegue la routine syscall_table_finder() che va a cercare la syscall table tra le pagine candidate. La funzione validate_page() si occupa proprio di trovare il candidato che rispetta i nostri requisiti; nel momento in cui lo troviamo il la routine si ferma e possiamo andare ad eseguire la ricerca utilizzando il risultato salvato nella variabile definita come hacked_syscall_tbl ed i relativi risultati definiti in hacked_ni_syscall con i quali riempiamo il nostro array utilizzando la fill_ni_syscall_founded.

Inserisco e definisco inoltre gli entry point delle syscall per le quali andrò ad implementare il frontend in un file esterno.

Nel momento in cui abbiamo montato il modulo viene eseguita e perfezionata questa ricerca finalizzata al secondo modulo ma nel caso in cui smontassimo il modulo syscall_hacking tutte le aree di memoria occupate o sporcate per la ricerca vengono ripulite e ovviamente ogni risultato viene printato nel log del kernel che possiamo ispezionare attraverso la *dmesg*.

3.2. Services

In questo file vengono creati i linking alle strutture di dati di appoggio per lo sviluppo dei servizi e l'inizializzazione dei servizi di sincronizzazione utilizzati. Le syscall vengono implementate come 'servizi', sono stati sviluppati quindi quattro servizi per lo sviluppo del sottosistema di scambio di dati basato su tag.

- **Tag_get(int key, int command, int permission)** la chiamata di sistema istanzia o apre il servizio Tag associato alla key dipendente dal valore command. Il valore di ritorno della chiamata, se non dovesse restituire errore, ci consegna il descrittore del tag service.
  Andando più alla pratica il servizio esegue:
  - un primo controllo si assicura che nel caso in cui la lista di tag sia vuota, ovvero avviene la prima esecuzione o esecuzioni successive alla prima ma con tutti i tag precedentemente creati che sono stati rimossi; alloca la

memoria necessaria ad accogliere il servizio di scambio di messaggi.
  - continuando poi se il comando è 1, il valore corrisponde alla creazione del tag (**CREATE**, vedi services.h) settiamo i valori corrispondenti che sono stati passati tramite syscall nelle rispettive parti della struttura del tag e nel caso in cui il tag aperto sia di tipo IPC_PRIVATE questo viene settato assegnando al tag il valore della key pari a 0 per essere riconosciuto per un eventuale futura riapertura da bloccare.
    Oltre ad inizializzare la struttura 'tag' eseguo lo stesso procedimento per quella che è stata definita 'level', ad ogni livello viene associato il tag ed inizializziamo la wait queue che analizzeremo meglio nel capitolo della sincronizzazione.
  - nel caso in cui invece il comando è 2 si tratta quindi di una apertura del tag (**OPEN**, vedi services.h) in questo caso ci dobbiamo assicurare che il tag sia apribile (ovvero non sia IPC_PRIVATE) e dobbiamo controllare il permesso con il quale il tag è stato aperto per capire se solo il tag che lo ha creato può riaprire il servizio o se anche altri tag.

- **Tag_send(int tag, int level, char* buffer, size_t sizeBuff)**
  prende il messaggio da buffer e manda sul livello dentro un determinato tag, in questo momento tutti i thread bloccati si svegliano e leggono. La chiamata di sistema, quindi, consegnerà al tag con descrittore il messaggio in buffer, tutti i thread in attesa sul valore level dovrebbero essere ripresi e ricevere il messaggio; se nessuno è in attesa il messaggio viene scartato. Tutto ciò avviene utilizzando la struttura TAG.list a cui abbiamo associato tutti i tag e le relative informazioni, necessarie all'utilizzo del sottosistema in fase di creazione del servizio tag.
- **Tag_receive(int tag, int level, char* buffer, size_t sizeBuff)**
  il thread si blocca e aspetta che arrivi un messaggio. Il servizio permette quindi a un thread di ricevere il messaggio e può fallire con un segnale mentre un utente è in attesa.
  Per ottenere ciò nel momento in cui chiamiamo una receive aumentiamo il contatore dei lettori su quel determinato tag di uno attraverso la __sync_fetch_and_add e ci mettiamo in attesa dei messaggi sempre sul nostro tag a meno che un segnale (-ERESTARTSYS) arrivi e ci faccia fallire l'attesa della ricezione del messaggio.
  Sia nel caso in cui l'attesa fallisca (nel caso del segnale) o nel caso in cui invece durante l'attesa ricevo un messaggio vado a settare i rispettivi campi delle strutture con le informazioni ricavate dall'evento.
- **Tag_ctl(int tag, int command)**
  si può andare ad operare sul tag con differenti comportamenti a seconda dei diversi valori di command che vengono passati al servizio: nel caso in cui abbiamo il valore 3 siamo nel caso dell'**AWAKE_ALL** in cui vengono svegliati tutti i thread in attesa di messaggi. Mentre se il valore del campo command è 4 siamo nel caso **REMOVE** e abbiamo intenzione quindi di rimuovere il tag in quesione; un tag non può essere rimosso se sono presenti thread in attesa su di esso.
  Per operare queste due funzionalità ci si appoggia al file util_tag in cui sono state implementate due funzioni di utility: la prima search_for_level che si occupa di capire se su un livello di un determinato tag abbiamo un messaggio (e quindi se il tag da rimuovere può essere effettivamente rimosso); mentre la delete_tag ci viene in soccorso nel caso in cui la rimozione può andare a buon fine implementando la logica di pulizia dei campi di un tag tale da rappresentarlo come rimosso.

Ovviamente tutto ciò viene eseguito se i parametri passati in input alla chiamata di sistema sono validi ed utilizzabili attraverso relativi controlli.

L'avanzamento di queste operazioni viene notificato nel log del kernel.

3.3. Tag struct

Il tag è una struct che ha le informazioni che ci permettono di utilizzare le syscall implementate, ovvero capire se un tag può essere aperto, creato, se un messaggio può essere inviato, ricevuto o meno.

La struttura è composta dai campi:

- **exist**: flag che consente di capire se il tag esiste o meno all'interno della tag list.
- **key**: che rappresenta la chiave del tag.
- **command** = valore che rappresenta il comando che gli abbiamo passato.
- **permission** = flag che ci consente di capire se il tag è privato o meno.
- **private**: campo che ci permette di salvare l'id del thread che ha aperto il tag in modalità privata in modo da permettere solo a lui di riaprirlo.
- **opened**: flag che ci consente di capire se il tag è aperto.
- puntatore ad una struttura **level**.

3.4. Level struct

La struttura è composta dai campi:

- **bufs**: buffer che ci permette di salvare il messaggio corrispondente al livello-tag.
- **lvl**: descrittore univoco del livello.
- **tag**: descrittore univoco del tag (proprietario del livello).

- **is_empty**: flag che ci aiuta a capire se un livello è vuoto o meno più velocemente.
- **is_queued** = flag che ci è utile per andare ad eliminare il tag nella tag_ctl, infatti possiamo entrare nel ciclo per eliminare solamente un thread che è stato messo in coda.
- **wq**: campo di tipo wait_queue_head_t che ci permette di tenere traccia dei thread in attesa.
- **reader**: campo in cui vengono aggiornati (atomicamente) i lettori di quel livello.

3.5. Driver

Il secondo ed ultimo modulo che carichiamo nel kernel è quello implementato nel driver in cui viene implementata la logica richiesta del device driver.

Questo modulo si occupa di implementare un semplice device driver che permette di controllare lo stato del servizio in termini di tag, livelli e threads in attesa. Il device file corrispondente viene aggiornato sfruttando una funzione del modulo tag, tag_info, che scorre tutti i servizi presenti in tags ed aggiunge una nuova riga di informazioni per ogni livello.

Come spiegato a lezione è stata implementata la funzione init_module per inizializzare il driver identificato dal MAJOR_NUMBER che possiede un numero di oggetti pari al valore di MINOR_NUMBER. Il lavoro principale è stato fatto nelle funzioni di read e di write, con la read dopo aver lasciato i controlli del codice natio, vado a scrivere sull'oggetto. A differenza di quello che abbiamo utilizzato fin'ora nel device driver vengono utilizzati i mutex per sincronizzare e bloccare le operazioni di lettura e scrittura.

3.5.1. Driver struct. La struttura è composta dai campi:

- **tag**: key del tag.
- **thread**: identificativo del thread.
- **sleepers**: numero dei thread che stanno aspettado.

4. Sincronizzazione

Sono stati utilizzati diversi metodi per garantire la sincronizzazione all'interno del sottosistema come le **spin_lock** e le **spin_unlock** per eseguire le modifiche nelle strutture dei tag, per inviare e ricevere i messaggi.

Uno spin lock è un modo per proteggere una risorsa condivisa dalla modifica di due o più processi contemporaneamente. Il primo processo che tenta di modificare la risorsa "acquisisce" il blocco e continua sulla sua strada, facendo ciò che era necessario con la risorsa. Tutti gli altri processi che successivamente tentano di acquisire il blocco vengono arrestati; si dice che "ruotano in posizione" in attesa che il blocco venga rilasciato dal primo processo, da cui il nome spin lock.

Queste strutture sono infatti ampiamente utilizzate all'interno del kernel Linux utilizza per molte cose, come ad esempio quando si inviano dati a una determinata periferica. La maggior parte delle periferiche hardware non è progettata per gestire più aggiornamenti di stato simultanei. Se devono verificarsi due diverse modifiche, una deve seguire rigorosamente l'altra, non possono sovrapporsi. Un blocco spin fornisce la protezione necessaria, garantendo che le modifiche avvengano una alla volta.

La **wake_up_interruptible** viene utilizzata per poter implementare la logica di invio e di ricezione dei messaggi infatti l'API in questione mette in attesa il thread finchè una condizione non si verifica.

Quando questo accade tutti i processi nella coda di attesa passati in parametro vengono messi in uno stato RUNNING e cancellati dalla coda; nel nostro caso l'evento scatenante è l'invio di un messaggio. Questa API ci da quindi la possibilità al messaggio di poter essere ricevuto da tutti i thread che erano in attesa di ricevere e che erano stati inseriti nella wait queue presente in ogni livello di ogni tag creato nel sottosistema.

La **__sync_fetch_and_add** viene utilizzata per aggiornare atomicamente i contatori o i valori delle strutture nella manipolazione dei messaggi o degli eventi del flusso. Infatti quando viene richiamata questa funzione viene creata una full memory barrier che ci permette di operare sul dato passato che viene restituito poi come valore di ritorno.

La **copy_from_user** e la **copy_to_user** sono state utilizzate per inviare e ricevere i messaggi, ovvero per manipolare i buffer di appoggio e delle relative strutture. Infatti la prima ci permette di copiare un blocco di dati dallo user space verso il kernel space mentre la seconda copia un blocco di dati dal kernel space nello user space, l'operazine inversa.

5. Repository

Si allega qui il collegamento alla repository in cui si trova il codice dell'implementazione della specifica, la relazione in formato pdf ed il file di readme.