TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Department of Automation

Prof Francesco Quaglia - Sistemi Operativi Avanzati

TAG-based data exchange

Author:
Emanuele Alfano

Abstract

Lo scopo del sistema è di mettere in cominicazione un numero imprecisato di processi all'interno dello stesso kernel, e permettere lo scambio istanteneo di pacchetti tra 1 **Writer** e i **Reader** in attesa su di un canale.

La logica del sistema segue lo schema publisher-subscriber, dopo un subscriber, una volta ottenuto ciò che desidera, deve manualmente accodarsi nuovamente al publisher.

Le richieste, in dettaglio, del progetto sono reperibili al link: SOA-prg

Date
01/05/2021

Contents

List of Figures

List of Tables

1 Introduzione

Lo scopo del sistema è di mettere in cominicazione un numero imprecisato di processi all'interno dello stesso kernel, e permettere lo scambio istanteneo di pacchetti tra 1 **Writer** e i **Reader** in attesa su di un canale.

La comunicazione viene svolta nel minor tempo possibile e adottando un algorimo *Semi-Lock Free*

Il comportamento desiderato è che l'arrivo di 1-**Writer** che scrive su di un canale, porta al risveglio degli n-**Reader** precedentemente in attesa sullo stesso, il tutto nel minor tempo possibile.

Il canale di comunicazione è definito su 2 livelli di ricerca:

1. **Tag-level search**
   Stanze effettivamente instanziate nel sistema, di default ne possono esistere fino a 256, ma è possibile variare a **Run-Time** questo limite facendolo crescere a piacere, o decrescere fino al numero di stanze attualmente aperto, e comunque non meno di 256 **nell'implementazione di questo progetto**.
2. **Topic-level**
   Ogni stanza, possiede a sua volta 32 sotto livelli dove effettivamente i **Reader** e **Writer** parlano

Segue che nella stessa stanza(*tag*) è possibile svolgere in parallelo più conversazioni, e far accodare diversi thread su diversi topic, fino al limite di 32 *topic* per stanza.

Le stanze possono essere pubbliche, e in tal caso gli si associa una *key* , o private, e in tal caso la *tag* è nota solo a chi ha creato la stanza.

Per interagire con il sistema sono state implementate 4 system-call "rubate" alla sys-call table dalle funzioni che puntavano alla `sys_ni_syscall`, ovvero quelle ancora implementate nel kernel.

1. `int tag_get(int key, int command, int permission)`
   Per me mezzo di questa sys-call è possibile creare una stanza publica o privata (permission) rispetto ai thread di altri processi, e scegliere se deve essere indicizzata globalmente o meno (key)
2. `int tag_send(int tag, int level, char* buffer, size_t size)`
   Questa sys-call permette a un **Writer** di publicare nella stanza desiderata al livello voluto
3. `int tag_receive(int tag, int level, char* buffer, size_t size)`
   Questa sys-call mette un **Reader** in attesa nella stanza al livello voluto finchè un **Writer** non la risveglia
4. `int tag_ctl(int tag, int command)`
   Con quest'ultima syscall è possibile inviare dei comandi all'intera stanza, nella fattispecie la chiusura e la wake-up-all

Per avere più dettagli sulle richieste delle interfacce si faccia riferimento alla pagina ufficiale del progetto: SOA-prg

In aggiunta alle richieste del progetto, questa implementazione ha la possibilità di modificare a **Run-Time** il numero di di Stanze massime del sistema, facendolo crescere a piacere, e ridurre fino al numero minimo di 256, senza però scendere sotto il numero di stanze correntemente allocate.

Viene anche fornita la libreria `user-space` che permette di trovare autonomamente i numeri delle syscall, e implementa un `perror` standard per ciascuna delle 4 syscall.

2 SubSystem of Module

Il progetto è organizzato in 3 diversi sotto sistemi::

1. SysCall Discovery
   Sviluppato direttamente dal repository del professore (Git repository), la quale è stata modificata per farla diventare una libreria del sistema
2. Tag-Based Data Exange
   Libreria core del sistema, essa implementale funzionalità richieste per il progetto
3. Char Device
   Libreria che permette di esporre il Modulo al resto del sistema operativo facendolo passare per un dispositivo a caratteri

I quali sono organizzati affinché lavorino con il maggior livello di interdipendenza possibile, facendoli mettere in comunicazione dentro *main.c*.

2.1 SysCall Discovery

Il Sotto-sistema che si occupa di ricercare la posizione all'interno della memoria della `sys_call_table` è stato sviluppato partendo dal lavoro del professore Francesco Quaglia (Git repository).

In particolare il progetto è stato forkato al commit: `eafea1bcacb0ee7f81f2457`ce03290d98c7947, il quale è stato modificato e adattato per diventare una libreria del sistema che espone la funzionalità di modifica e ripristino delle syscall "Libere" (le syscall che puntano a `sys_ni_syscall`) necessarie.

Esso comincia una ricerca lineare lungo i primi 4GB della Ram, e verifica, byte per byte, la struttura della memoria sottostante, se essa coincide con quella della `sys_call_table` nell'area di memoria in esame. Se il test ha successo, allora l'area di memoria trovata viene considera la `sys_call_table`. Il rischio di falsi positivi tuttavia esiste, e per minimizzare un simile rischio sono state utilizzate gli spiazzamenti di 7 diverse SysCall non implementate, così da rendere minimo il rischio di falsi positivi.

Mediante questo metodo di attraversamento lineare di tutto l'address space è viene trovata la posizione della `sys_call_table` evitando falsi positivi.

Le posizioni trovate, vengono salvate su un array e la libreria permette di modificare queste entry. Durante lo smontamento del modulo, la libreria si occupa di ripristinare la tabella alla sua forma originale. Ciò è verificabile montando e smontando ripetutamene il modulo: se le syscall non fossero ripristinate, sarebbe impossibile individuare nuovamente la tabella.

2.2 Tag-Based Data Exange

Il cuore del progetto (lo scambio di messaggi tra processi) viene realizzato dal TBDE SubSystem.

Per poter svolgere la sua funzione, si è scelto di organizzare le strutture dati in:

AVL-Tree :

Il sistema usa 2 Alberi-AVL di ricerca, *tagTree* e *KeyTree* , per indicizzare in maniera efficiente le **Stanze** per *tag* e per *key* .
L'albero principale è il *tagTree* , mentre il *KeyTree* è di supporto per la ricerca efficiente delle stanze con visibilità "pubblica".
Gli alberi sono sincronizzati mediante degli `Spinlock RW` che permettono di svolgere in **reale parallelismo tutte le operazioni comunicazione** e vincolano a 1 thread alla volta per le operazioni di:

- *Aggiunta*
  Creazione di una stanza, pubblica o privata, con i permessi richiesti:
  (`aperta a tutti/solo ai thread del processo creante`)
- *Rimozione*
  Rimozione di una stanza, se i permessi lo permettono, il SUPER-UTENTE è sempre considerato valido
- *Status-Print*
  Operazione eseguita dal driver, quando va a fare il cat dello stato del sistema corrente, per garantire la coerenza dei dati letti nel trovare in quale stanze vi sono dei reader in attesa e quanti. (Operazione assolutamente infrequente e user-driver)

Per maggiori informazioni sulla scelta di dell'uso di `Alberi-AVL RWLock` rispetto a una lista RCU, riferirsi all'approfondimento alla sezione AVL-Tree Performance.

I nodi degli Alberi puntano alle **Stanze** allocate nel sistema, non una copia personale per l'albero, ma proprio la stessa istanza, ciò implica che i 2 alberi devono essere consistenti tra di loro, e che il lock descritto precedentemente, copre contemporaneamente entrambe le strutture dati.

Stanze (*Room*):

Il nodo della **Stanza** è considerabile come un oggetto allocato nell'heap del kernel, si è reso quindi necessario usare le API dei `refcount` presenti all'interno del kernel. Il contatore cresce al crescere degli oggetti che la riferiscono, e decresce alla loro rimozione, il thread che la porta effettivamente a 0 il contatore esegue anche le operazioni di liberazione della memoria.

Topic (*exangeRoom*):

Ogni stanza ha a sua volta 32 *exangeRoom* (i Topic), che permettono lo scambio effettivo delle informazioni tra 1 **Writer** e gli n-**Reader** precedentemente in attesa su questo livello dall'arrivo del precedente **Writer** .

Ognuna delle *exangeRoom* è anch'essa un oggetto del sistema, e viene gestita con le API dei `refcount`, l'unica differenza è che per le *exangeRoom* risulta ammissibile che la stanza sia valida con un contatore a 0, questo perchè le *exangeRoom* contengono i **Reader** in attesa su quel Topic, ed è possibile che un Topic sia senza **Reader** in attesa, questa scelta è stata presa allo scopo di massimizzare i tempi di risposta del sistema, ed evitare incoerenze. Questa scelta ha però generato la necessità di un uso non proprio lecito delle API di `refcount`, poichéè per queste API un refcount = 0 equivale a un area di memoria che deve essere liberata, e non viene permesso l'incremento. Per ovviare al problema , si è sostituita la `refcount_inc` con il suo equivalente `atomic_inc`, che però bypassa i check di consistenza.

Per approfondire il comportamento del sistema nello scambio di messaggi fare riferimento alla sezione Exange Data protocol.

2.2.1 Exange Data protocol

Fin dall'inizio della sua progettazione, l'obiettivo principale del sistema è stato permettere a un **Writer** di comunicare con tutti gli n-**Reader** precedente accodati sul Topic, nel minor tempo possibile ed evitando lock di sincronizzazione, il risultato è un sistema rapido e snello, che ha come **unico lock** un *custom lock di enable/disable della Free* (vedi seguito per dettagli), che comunque rallenta al più per un paio di cicli macchina, ma evita una rara, seppur possibile, corsa critica nell'accesso a un area di memoria.

Per accodare e addormentare i **Reader** in attesa sono state utilizzate le `API di Wait-Queue` del Kernel che già evitano il problema del Wake up Lost Problem.

Il protocollo di scambio dati è progettato per garantire che un **Writer** possa prendere tutti i **Reader** accodati fino al momento del suo arrivo, evitando che 2 **Writer** possano parlare agli stessi **Reader** .

Esso consiste nel lasciare sempre disponibile ai **Reader** del sistema, una coda su cui andare a dormire, questa coda viene **Atomicamente Swappata** con una vuota all'arrivo di un **Writer** , così da fargli "catturare" tutti i **Reader** serializzati fino al suo arrivo; da questo momento il **Writer** ha tutto il tempo che vuole per condividere i suoi dati e svegliare i **Reader** addormentati.

La logica dello scambio dati è molto simile a quella di una lista RCU da 1 nodo, la differenza sta nel modo in cui viene considerato il periodo di grazia: in questo sistema, dove sono presenti i `refcount`, ogni thread, **Writer** compreso, tenta di fare una free della stanza, ma solo l'ultimo, colui che porta il contatore a 0, la esegue, così facendo il sistema è lock-free, e solo l'ultimo thread ha il compito di liberare la memoria, evitando di tenere gli altri in attesa in caso un thread per qualche motivo dovesse subire un rallentamento (priorità più bassa o simili).

Vediamo ora lo pseudo-codice dei 2 flussi:

Algorithm 1 Writer ExangeDataProtocol

```
1:  procedure TAGSEND(tag, level, msg)
2:      Read-Lock(tagTree )
3:      roomSearch ← Tree_search(tag, tagTree)
4:      if roomSearch == FOUND then
5:          RoomRefInc(Room)
6:          Read − unLock(tagTree)
7:          if permissionCheck(roomSearch) == True then
8:              newER ← makeExangeRoom()
9:              exange ← CompareSwap(newER, currER(level))        ▷ Swap atomico
10:             ExangeRefInc(currER)
11:             copyToExangeRoom(msg)
12:             wakeUp(WaitQueue)
13:             tryFreeExangeRoom(exange)
14:             TryFreeRoom(roomSearch)
15:         else
16:             TryFreeRoom(roomSearch)
17:             return −1                                         ▷ Operation fail
18:         end if
19:     else
20:         Read − unLock(tagTree)
21:         return −1                                             ▷ Operation fail
22:     end if
23: end procedure
```

Algorithm 2 Reader ExangeDataProtocol

```
1:  procedure tagRecive(tag, level, msg)
2:      Read-Lock(tagTree )
3:      roomSearch ← Tree_search(tag, tagTree)
4:      if roomSearch == FOUND then
5:          RoomRefInc(Room)
6:          Read − unLock(tagTree)
7:          if permissionCheck(roomSearch) == True then
8:              newER ← makeExangeRoom()
9:              exange ← currER(level)          ▷ Ottengo la stanza attualmente sincronizzata
10:             ExangeRefInc(currER)
11:             InterruptableSleep(WaitQueue)
12:             ...
13:             Wake Up Event
14:             if Wake up for signal then
15:                 ExitFreeExangeRoom(exange)
16:                 return −1                                   ▷ Signal Wake up
17:             end if
18:             copyFromoExangeRoom(msg)                        ▷ Normal Wake up
19:             tryFreeExangeRoom(exange)
20:             TryFreeRoom(roomSearch)
21:         else
22:             TryFreeRoom(roomSearch)
23:             return −1                                       ▷ Operation fail
24:         end if
25:     else
26:         Read − unLock(tagTree)
27:         return −1                                           ▷ Operation fail
28:     end if
29: end procedure
```

La differenza tra `Exit_Free` e `Try_Free` stà nel fatto che il primo, in caso il contatore scenda a 0, non elimina la stanza, mentre il secondo si.

N.B. Il protocollo descritto permette di far eseguire i Writer e Reader dello scambio dei messaggi in reale parallelismo!!!

Il protocollo è inspirato alla logica delle liste RCU, con la differenza che non è il **Writer** necessariamente a dover attendere il periodo di grazia, ma semplicemente il Thread più "lento" è anche colui che libera la memoria.

Per un limite tecnico degli attuali processori, le **operazioni realmente ATOMICHE** non comprendono + di 1 variabile alla volta, ciò fa si che il sistema sia esposto per qualche ciclo macchina al rischio di fare una Free di un area di memoria prima che si possa fare +1 al suo refcount. Per ovviare questo limite, è stata scritto un piccolo Spinlock custom (reperibile in tbde.h):

```
// Custom lock define
#define freeMem_Lock(atomic_freeLockCount_ptr)          \
  do {                                                  \
    preempt_disable();                                  \
    atomic_inc(atomic_freeLockCount_ptr);               \
  } while (0)

#define freeMem_unLock(atomic_freeLockCount_ptr)        \
  do {                                                  \
    atomic_dec(atomic_freeLockCount_ptr);               \
    preempt_enable();                                   \
  } while (0)
```

```
#define waitUntil_unlock(atomic_lockCount_ptr)              \
  do {                                                      \
    preempt_disable();                                      \
    while (arch_atomic_read(atomic_lockCount_ptr) != 0) {   \
    };                                                      \
    preempt_enable();                                       \
  } while (0)
```

Partendo dal presupposto che i Thread siano già arrivati nella Room corretta, quello che si troveranno davanti saranno `32 exangeRoom`, esse a livello implementativo altro non sono che delle *wait queue* allocate dinamicamente, e ogni stanza ha la sua variabile di controllo per il risveglio. Le API del kernel sono già implementate al fine di evitare il problema del Wake up Lost Problem, resta però da gestire la concorrenza e la memoria.

Ogni stanza è un oggetto a parte, e in quanto tale ha anche lei il problema di dover essere riferita un numero consistente di volte, per risolvere il problema sono state usate anche qui le API del `refcount`, con una leggera modifica rispetto all'uso classico, poichè è possibile che un area sia valida ma senza nessuno che la possegga, invece di usare l'API `refcount_inc` si è optato per il suo equivalente `atomic_inc`, questo perchè se si esegue un incremento su una variabile con contatore pari a 0 vengono emessi dei warning e la variabile viene portata al numero negativo massimo.

Analizziamo ora il comportamento del sistema in uno scenario normale:

- ➔ Room 5 Creation (create on tree)
- ➔ Read on Room 5 at level 3
- ➔ ...
- ➔ Get up and read data...
- ➔ try free of current Exange Room
- ➔ ...
- ➔ Room 5 Open (search on tree system)
- ➔ Write on Room 5 at level 3
- ➔ Wake up the reader on the level
- ➔ try free of current Exange Room

Iniziamo

Una volta arrivati alle Stanze i thread liberano i lock sugli Alberi, e sono garantiti che l'area di memoria su cui si trovano resta valida grazie all'incremento del `refcount` corrispondente. Nello scambio di informazioni tra il **Writer** e gli n-**Reader** , il sistema è progettato per rendere minimo il tempo di attesa da quando viene segnalata la presenza di un nuovo dato a quando essa viene consegnata, l'unico rallentamento necessario è sperimentato dall'ultimo thread che

2.2.2 AVL-Tree Performance

L'architettura degli Alberi-AVL del sistema è stata pensata per minimizzare il tempo di attesa percepito dai thread durante lo scambio di messaggi, penalizzando lievemente le operazioni di creazione-controllo-distruzione sulle stanze all'interno del sistema.

2.3 Char Device