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 comunicazione un numero imprecisato di processi all'interno dello stesso Kernel, e permettere lo scambio istantaneo di pacchetti tra un **Writer** e i **Reader** in attesa su 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-prj

Date
05/05/2021

Contents

1 Introduzione

Il progetto ha l'obiettivo di mettere in comunicazione un numero imprecisato di processi all'interno dello stesso Kernel; e permettere lo scambio di un messaggio tra 1-**Writer** e gli N-**Reader** precedentemente in attesa su un canale.

La comunicazione viene svolta nel minor tempo possibile e adottando un algoritmo *Semi-Lock Free* (permette e favorisce la concorrenza nelle operazioni di scambio di messaggi, ma blocca nelle operazione di creazione e rimozione di stanze dal sistema).

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

1.1 Identificazione dei canali di comunicazione

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

1. **Tag-level search**
   Codice delle stanze effettivamente in-stanziate nel sistema, di default ne possono esistere fino a 256; è tuttavia 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** (vedi la Sezione "Special Features").
2. **Topic-level**
   Ogni stanza, possiede a sua volta 32 sotto livelli dove effettivamente i **Reader** e il **Writer** possono parlare.

Segue che nella stessa stanza, identificata per mezzo di un *tag* , è possibile svolgere in parallelo più conversazioni, e far accodare i diversi Thread sui 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 al creatore della stanza.

1.2 System behavior

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 mezzo di questa syscall è possibile creare una stanza con visibilità pubblica o privata (key) per i Thread degli altri processi, e decidere se vi possono lavorare o meno (permission)
2. `int tag_send(int tag, int level, char* buffer, size_t size)`
   Questa syscall permette a un **Writer** di pubblicare nella stanza desiderata al *Topic* desiderato
3. `int tag_receive(int tag, int level, char* buffer, size_t size)`
   Questa syscall mette un **Reader** in attesa nella stanza al *Topic* desiderato 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 della stessa (possibile solo se nessun **Reader** è in attesa dentro la Stanza) o il wake-up forzato di tutti i **Reader** nella stanza.

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

1.3 Special Features

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, esposti dentro un `module_param_array` al path: `/sys/module/TAG_DataExchange/parameters/sysCallNum`, e in aggiunta è implemento il `perror` standard per ciascuna delle 4 syscall.

Per terminare, sono forniti diversi codici di test e gli script che permettono di interagire attraverso il terminale con il modulo.

1.4 SubSystem division

Il progetto è organizzato in 3 diversi Sottosistemi:

1. SysCall Discovery
   Sviluppato direttamente dal repository del professore (Git repository), il quale è stato modificato per farlo diventare una libreria integrata nel sistema, con qualche funzione di interfaccia
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 Sottosistemi sono organizzati affinché lavorino con il maggior livello di interdipendenza possibile, facendoli mettere in comunicazione dentro *main.c*.

2 SysCall Discovery

Il Sottosistema 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: `eafea1bcacb0ee7f81f2457ce03290d98c7947`; esso è stato modificato e adattato per diventare una libreria del sistema che permette di esporre le 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 considerata la `sys_call_table`. Il rischio di falsi positivi tuttavia esiste e per minimizzare un simile rischio sono stati utilizzati gli spiazzamenti di 7 diverse SysCall non implementate, così da rendere minimo il rischio di falsi positivi.

2.1 Enhanced Features

In aggiunta alle funzionalità offerte dal progetto base del professore, il Sottosistema è in grado di effettuare il restore delle entry modificate dal sistema durante lo smontamento del modulo. Ciò è verificabile montando e smontando ripetutamene il modulo: se le syscall non venissero ripristinate sarebbe impossibile individuare nuovamente la tabella.

3 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 usare come strutture dati:

AVL-Tree per indicizzare le stanze per *tag* e *key* :
Il sistema usa 2 Alberi-AVL di ricerca, *TagTree* e *KeyTree* , per indicizzare in maniera efficiente le **Stanze** in base alla chiave di ricerca usata (*tag* e *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, con visibilità pubblica o privata, usando permessi richiesti:
  (`Open(aperta a tutti)`/`Private(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 nel Char-Device, per generare lo "Screenshot" dello stato del sistema in quell'istante temporale. Per garantire la coerenza dei dati letti nel trovare in quale stanze vi sono dei **Reader** in attesa e quanti, viene fatto un Lock generale del sistema al livello degli Alberi-AVL, l'operazione blocca temporaneamente il procedere del sistema ma permette di eseguire uno "Screenshot" coerente. Il lock non è problematico poiché è un operazione assolutamente infrequente e User-driven.

Per maggiori informazioni sulla scelta di usare `Alberi-AVL RWLock` rispetto a una lista RCU, fare riferimento all'approfondimento presente nella 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, copra contemporaneamente entrambe le strutture dati.

Stanze (*Room*):
Il nodo della **Stanza** è da considerare 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 all'aumentare degli oggetti che puntano l'oggetto "Stanza" e decresce alla loro rimozione, il Thread che porta effettivamente il contatore della "Stanza" a 0 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 *Topic* 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 perché le *exangeRoom* contengono i **Reader** in attesa su quel *Topic* , ed è possibile che un *Topic* sia senza **Reader** in attesa.

Questa scelta ha però generato la necessità di un uso non proprio lecito delle API di `refcount`, poiché, 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 e come queste 3 strutture dati interagiscono per permettere lo scambio di messaggi, fare riferimento alla sezione Exange Data protocol.

3.1 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 rallenta solo durante la Creazione o Rimozione delle stanza (operazioni supposte infrequenti) e nelle operazioni di comunicazione richiede un unico *Soft Lock* implementato come un *custom lock di enable/disable Free* (vedi Memory Free lock Protocol), 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** .

Ciò fa "catturare" tutti i **Reader** serializzati fino al suo arrivo e svegliare i **Reader** addormentati senza concorrenza con altri **Writer** , che possono eseguire in tranquillità parlando con i nuovi **Reader** in attesa.

3.1.1 Reader e Writer Protocol

Vediamo ora lo pseudo-codice dei 2 flussi:

Algorithm 1 Writer ExangeDataProtocol

```
1:  procedure TAGSEND(tag, level, msg)
2:      ReadLock(TagTree)
3:      roomSearch ← TreeSearch(tag, TagTree)
4:      if roomSearch == FOUND then
5:          RoomRefInc(Room)
6:          ReadUnLock(TagTree)
7:          if permissionCheck(roomSearch) == True then
8:              newER ← makeExangeRoom()                ▷ La coda vuota da Sostituire
9:              exange ← CompareSwap(newER, currER(level))   ▷ Swap atomico
10:             ExangeRefInc(exange)
11:             copyToExangeRoom(msg)
12:             wakeUp(exange.WaitQueue)
13:             tryFreeExangeRoom(exange)
14:             TryFreeRoom(roomSearch)
15:         else
16:             TryFreeRoom(roomSearch)
17:             return −1                               ▷ Operation fail
18:         end if
19:     else
20:         ReadUnLock(TagTree)
21:         return −1                                   ▷ Operation fail
22:     end if
23: end procedure
```

Algorithm 2 Reader ExangeDataProtocol

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

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

Il protocollo risulta essere fluido e non sono presenti lock espliciti all'interno della *exangeRoom* all'infuori dell'attesa del **Reader** , necessaria e desiderata.

Da osservare i comandi in blu del codice, essi evidenziano i 2 diversi tipi di FreeExangeRoom:

Exit Free :
In caso il contatore **scenda a 0**, non elimina la stanza, poiché l'area di memoria è in realtà ancora valida e raggiungibile

Try Free :
Quando il contatore **scende a 0**, si può effettivamente procedere all'eliminazione dell'oggetto, poichè non è più riferito da nessuno.

3.1.2 Memory Free lock Protocol

Purtroppo la sezione Rossa del **Reader** ExangeDataProtocol necessita di poter essere eseguita **Atomicamente** per garantire che l'incremento del contatore avvenga prima che lo stesso possa essere eliminato da una *TryFreeRoom(roomSearch)*. Per eliminare la corsa critica nell'acquisizione della *exangeRoom* e il successivo incremento del *refcount*, si è scelto di implementare un protocollo di `Memory Free lock`:

Esso è inspirato alla logica delle liste RCU, con una differenza nel modo di attendere il **tempo di grazia**: classicamente è il **Writer** a dover attendere il periodo di grazia, in questa versione invece

è semplicemente il Thread più "lento" ad eseguire il lavoro (dovuto magari a rallentamenti per lo scheduler o la priorità del Thread).

Per risolvere la corsa critica 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)
```

Queste API sono usate:

- Dai **Reader** (`freeMem_Lock` e `freeMem_unLock`) attorno alla sezione critica (Rossa)
- All'interno della *TryFreeRoom(roomSearch)* (la `waitUntil_unlock`) per poter verificare, **solo al raggiungimento dello 0**, che non sia in corso nessuna operazione rischiosa in memoria, e in tal caso attendere la fine della sezione critica e reagire di conseguenza alle modifiche del sistema.

Per minimizzare al massimo l'impatto di questo `Soft-lock`, ogni *Topic* è dotato del suo personale memoryLock, così da circoscrivere ai soli Thread interessanti il rallentamento. Va in oltre precisato che il rallentamento è sperimentato solo quando il contatore arriva a 0, durante i valori superiori del refcount, il contatore è gestito atomicamente dalle API del *refcount*, senza l'uso di sezioni critiche.

3.2 AVL-Tree Performance

Dopo aver visto l'architettura del sistema, ragionevolmente ci si può chiedere:

Perchè usare degli Alberi-AVL con dei RWlock invece di una lista RCU?

Per rispondere a questa domanda, e per capire il perchè siano state fatte le scelte realizzative è opportuno tenere in mente le priorità del sistema:

1. Scambio di messaggi rapido e affidabile tra i processi.
2. Parallelizzare le operazioni (in particolare quelle più frequenti)
3. Evitare operazioni inutili

3.2.1 RWLock AVL-Tree vs RCU List

Andiamo ora a comparare le prestazioni possibili tra:

`Albero-AVL con Spinlock RW`	`Lista RCU`
☺ **Writer** e **Reader** possono accedere in contemporanea e ci mettono al massimo $O(log(N))$ per arrivare alla stanza di loro interesse	☹ **Writer** e **Reader** possono accedere in contemporanea e ci mettono al massimo $O(N)$ per arrivare alla stanza di loro interesse
😐☹☺ Inserire/Rimuovere nodi con 1 Thread, dovendo fermare i **Writer** e **Reader** al più per $O(log(N))$	😐☺☹ Inserire/Rimuovere nodi con 1 Thread, senza mai fermare i **Writer** e **Reader** , ogni operazione richiede $O(N)$

Essendo entrambe le strutture dati organizzate come dei nodi da accedere, non è possibile avere su nessuna delle 2 un accelerazione HW particolare, e specie per sistemi molto grandi, **l'Albero AVL scala meglio lungo un periodo**, per capire meglio l'ultima affermazione consideriamo il seguente esempio al caso peggiore:

Nel sistema sono presenti 256 stanze, il SO ha a disposizione 8 Thread per poter eseguire le operazioni, prendiamo come periodo 256 operazioni (supponiamo che ogni mossa tra un nodo e l'altro richieda 1 operazione), ecco le prestazioni massime che ci possiamo aspettare, supponendo che tutte le operazioni puntino al 256-esimo nodo:

`Albero-AVL con Spinlock RW`	`Lista RCU`
☺ Search $8[Thread] \cdot \frac{256[OpTime]}{log_2(256[Room])} = 256[Op]$	☹ Search $8[Thread] \cdot \frac{256[OpTime]}{256[Room]} = 8[Op]$
☺ Insert/Delete (alternate) $1[Thread] \cdot \frac{256[OpTime]}{log_2(256[Room])} = 32[Op]$	☹ Insert/Delete (alternate) $1[Thread] \cdot \frac{256[OpTime]}{256[Room]} = 1[Op]$

Inoltre, all'aumentare dei nodi e del periodo la vittoria dell'Albero-AVL risulta particolarmente evidente, queste considerazioni hanno portato all'esclusione delle liste RCU per questa specifica implementazione del sistema, specie considerando che è possibile modificare a **Run-Time** il numero massimo di Stanze, facendo aumentare sempre più il vantaggio di usare un Albero-AVL.

3.2.2 RWLock AVL-Tree vs RCU AVL-Tree

Alla fine di questi calcoli viene normale chiedersi:

Perché non usare direttamente una libreria di Alberi-AVL RCU?

Qui la risposta è da trovare nei paper pubblicati: ad oggi (05/05/2021) l'argomento è dibattuto in ambito universitario, e sono state proposte delle soluzioni per estendere la logica RCU anche agli Alberi-AVL, esse però richiedono un enorme quantitativo di memoria extra, dovuto alla necessità di creare una copia di tutto il sotto albero del nodo che si intende modificare, con il rischio che al termine di questa operazione, il nodo non sia più valido perchè un altro Writer ha alterato un altro settore. Non si è quindi ritenuto opportuno procedere per una simile strada, e si è favorita la strada di uno Spinlock RW.

4 Char Device

Al fine di poter osservare lo stato del sistema, è stato prevista l'aggiunta di un device-driver, che permetta di ottenere una stringa inerente lo stato del sistema. Per implementare il driver sono state usate le seguenti `File Operation Function`:

open :
Viene fatto uno "Screenshot" dello stato attuale del sistema sotto forma di stringa, e l'istantanea viene salvata nel **Per-FileDescriptor Memory**.

read :
Permette di trasferire la stringa al lettore contenente le informazioni del sistema.

lseek :
Permette di muovere lo spiazzamento di lettura per poter tornare indietro, in caso di bisogno

relase (close) :
Permette di cancellare la memoria precedentemente allocata per bufferizzare lo "Screenshot" del sistema

Usando queste 4 operazioni è possibile far leggere lo stato del sistema al momento della Open e aggiornando, chiudendo e riaprendo il File Descriptor.

4.0.1 Analisi dell'Output

Essendo il sistema organizzato con 2 alberi, la soluzione migliore per mostrare lo stato, tanto del sistema, quanto degli alberi, è stata usare ogni riga per rappresentate un **Nodo**, e le righe **Superiori** e **Inferiori** per i propri figli.

Per leggere lo stato della stanza abbiamo che le informazioni sono:

1. *tag* e *key* della stanza, e il suo indirizzo di memoria.
2. Pid del creatore e tipo della stanza *Open for all=0 / Usabile solo dal processo creante=1.*
3. Lista dei *Topic* in cui vi è almeno 1 **Reader** in attesa, se non è presente nessun Reader viene scritto "no Waiters"
4. Somma dei **Reader** in attesa in tutta la stanza

I numeri tra parentesi rappresentano il fattore di sbilanciamento dell'Albero-AVL e per via della struttura non possono mai essere diversi da: -1, 0, +1.

Nel *TagTree* sono presenti effettivamente tutte le stanze del sistema; il *KeyTree* raccoglie solo quelle che hanno visibilità pubblica.

4.0.2 Example of Output

Di seguito l'output del sistema dopo aver eseguito il 7° test, che apre, parla e comunica a caso con un migliaio di processi, alcuni topic non vengono quindi mai scritti, e si ha l'effetto che rimangono in attesa.

In seguito, digitando `Ctrl+C` viene mandato a tutti i Thread un sigint che non essendo gestito user-space, porta al risveglio dalla wait, ma causa anche la terminazione del processo, ciò lascia un sistema "Sporco" e senza lettori, mostrato nel secondo listato:

System Screen-shot after running `./test/7_roomExange_LOAD.out`

```
tagTree (#room = 6; maxRoom = 256):
     .--{@tag(61)-@key(3)-->000000007ce9f0c4} [Creator=13919-perm=0 |L1=77| sum = 77]
 (-1)
    |     `--{@tag(51)-@key(7)-->000000008aec9ce4} [Creator=13459-perm=0 |L1=113| sum = 113]
 (+0)
---{@tag(45)-@key(5)-->000000002e6a0bb1} [Creator=13185-perm=0 |L1=104| sum = 104]
 (+0)
    |     .--{@tag(23)-@key(6)-->0000000007894bbe5} [Creator=11755-perm=0 |L1=112| sum = 112]
 (+0)
     `--{@tag(22)-@key(0)-->00000000b0a6d2fa} [Creator=11753-perm=0 |L1=93| sum = 93]
 (+0)
          `--{@tag(10)-@key(2)-->00000000a6a5c13b} [Creator=11209-perm=0 |L1=92| sum = 92]
 (+0)


keyTree:
     .--{@tag(51)-@key(7)-->000000008aec9ce4} [Creator=13459-perm=0 |L1=113| sum = 113]
 (-1)
    |     `--{@tag(23)-@key(6)-->0000000007894bbe5} [Creator=11755-perm=0 |L1=112| sum = 112]
 (+0)
---{@tag(45)-@key(5)-->000000002e6a0bb1} [Creator=13185-perm=0 |L1=104| sum = 104]
 (+0)
    |     .--{@tag(61)-@key(3)-->000000007ce9f0c4} [Creator=13919-perm=0 |L1=77| sum = 77]
 (+0)
     `--{@tag(10)-@key(2)-->00000000a6a5c13b} [Creator=11209-perm=0 |L1=92| sum = 92]
 (+0)
          `--{@tag(22)-@key(0)-->00000000b0a6d2fa} [Creator=11753-perm=0 |L1=93| sum = 93]
 (+0)
```

System Screen-shot after sending the signal `Ctrl+C`

```
tagTree (#room = 6; maxRoom = 256):
     .--{@tag(61)-@key(3)-->000000007ce9f0c4} [Creator=13919-perm=0 <no Waiters>]
 (-1)
    |     `--{@tag(51)-@key(7)-->000000008aec9ce4} [Creator=13459-perm=0 <no Waiters>]
 (+0)
---{@tag(45)-@key(5)-->000000002e6a0bb1} [Creator=13185-perm=0 <no Waiters>]
 (+0)
    |     .--{@tag(23)-@key(6)-->0000000007894bbe5} [Creator=11755-perm=0 <no Waiters>]
 (+0)
     `--{@tag(22)-@key(0)-->00000000b0a6d2fa} [Creator=11753-perm=0 <no Waiters>]
 (+0)
          `--{@tag(10)-@key(2)-->00000000a6a5c13b} [Creator=11209-perm=0 <no Waiters>]
 (+0)


keyTree:
     .--{@tag(51)-@key(7)-->000000008aec9ce4} [Creator=13459-perm=0 <no Waiters>]
 (-1)
    |     `--{@tag(23)-@key(6)-->0000000007894bbe5} [Creator=11755-perm=0 <no Waiters>]
 (+0)
---{@tag(45)-@key(5)-->000000002e6a0bb1} [Creator=13185-perm=0 <no Waiters>]
 (+0)
    |     .--{@tag(61)-@key(3)-->000000007ce9f0c4} [Creator=13919-perm=0 <no Waiters>]
 (+0)
     `--{@tag(10)-@key(2)-->00000000a6a5c13b} [Creator=11209-perm=0 <no Waiters>]
 (+0)
          `--{@tag(22)-@key(0)-->00000000b0a6d2fa} [Creator=11753-perm=0 <no Waiters>]
 (+0)
```

5 Build And Test

Per concludere, è possibile reperire sul: Project Git Repository, un recup delle features implementate e una lieve guida alla compilazione e caricamento della libreria nel Kernel.

I comandi necessari sono stati embeddati all'interno del file `Makefile`.

Il programma è stato scritto usando come editor Visual Studio Code, con l'enviroment impostato per leggere i sorgenti del Kernel "Linkati" tramite link simbolico alla directory `/usr/src` in questa maniera::

```
/usr/src
|-- linux -> linux-headers-5.11.0-16
|-- linux-generic -> linux-headers-5.11.0-16-generic
|-- linux-headers-5.11.0-16
|-- linux-headers-5.11.0-16-generic
```

Realizzabile con i permessi di root usando da terminale `ls -s <Target dir> <Link name>`.