PROGETTO ADVANCED OPERATING SYSTEMS 2020/2021

AOS TAG

TAG-BASED THREAD LEVEL **MESSAGE** EXCHANGE

SVILUPPATO DA **ANDREA PACI**
0286387

1 - Indice

2 - Introduzione

La seguente trattazione mira a presentare il lavoro svolto per la parte progettuale del corso di **Sistemi Operativi Avanzati** 2020/2021.
Questo documento presenta le tematiche nello stesso ordine con cui sono state pensate in fase di progettazione: si inizierà parlando della struttura generale del progetto per poi andare nel particolare con un approccio *top-down*.
Nel dettaglio, saranno prima **presentati** i due ***moduli kernel*** con i relativi compiti che assolvono, per poi essere discussi parlando inizialmente delle **strutture dati** usate a supporto ed infine la logica interna dei moduli.
Durante l'**analisi dettagliata** dei **moduli** vengono fatti *riferimenti al codice sorgente* confrontando i concetti espressi con le istruzioni relative.
Vengono successivamente presentati delle ***istruzioni*** su come installare il modulo all'interno di un OS Linux tramite un opportuno *How-to.*
Al termine dell'analisi dei due moduli segue un breve paragrafo riguardanti i **test sviluppati** e le **performance** misurate.
Infine sono presentate alcune ***considerazioni personali*** su *possibili sviluppi futuri* e *migliorie* che possono essere applicate.

Durante lo sviluppo si è scelto di seguire un approccio **più didattico** che **funzionale** su alcuni aspetti di architettura del progetto, principalmente per sperimentare maggiormente con costrutti e meccanismi poco familiari.
Queste scelte verranno opportunamente *evidenziate* durante il report, e verrà proposta una *controparte più efficiente e/o ragionevole* nel contesto in questione.

Il progetto è reso pubblico in una repository **Github**[1] dove sono presenti i sorgenti e gli script per l'installazione.

[1] github.com/andreapaci/TAG-AOS

3 - Traccia

Lo scopo del progetto è quello di **implementare** un **modulo linux** che abilità la **comunicazione tra thread** basata su **servizi Tag**. Ogni servizio **Tag** contiene un numero arbitrario di **livelli** su cui i *thread* possono interagire scambiando *messaggi*. Le strutture dei Tag e dei livelli devono essere gestiti dal *modulo*, cioè con privilegio Kernel. Per abilitare i processi di livello *user* ad interagire con il sistema è necessario instradare le richieste tramite *system call* installate sulla *system call table.* Gli scenari di impiego sono molteplici e sono racchiuse nelle 4 *system call* richieste:

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

  Con questa system call è possibile **stanziare** o **aprire** un **Tag service** identificato dalla *key.* E' possibile specificare i **permessi** del Tag service, limitando l'utilizzo al *solo utente che l'ha creato* o renderlo *disponibile* ad ogni *utente*. La chiave può essere impostata come **IPC_PRIVATE**, cioè il tag non è più *apribile* tramite tag_get. La system call, in caso di successo, **restituisce** il **tag descriptor** da utilizzare per le future interazioni con il Tag service.

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

  Tramite questa system call è possibile **inviare** un messaggio specificato nel **buffer** con una certa **size** al tag **tag** ed al livello **level**. Se nessun **ricevente** è registrato su quel tag a quel livello, il messaggio viene semplicemente scartato.

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

  Permette di **registrarsi** come ricevente al corrispettivo livello **level** del tag **tag**. Si rimane in attesa finché il messaggio non arriva o si è interrotti da un segnale di **Interrupt**.

- `int tag_ctl(int tag, int command)`

  Permette **l'eliminazione** di un **tag** oppure di risvegliare tutti i **thread riceventi** su quel determinato Tag

Come da *specifica*, il numero di **servizi Tag** stanziabili sono 256, mentre il numero di **livelli** per Tag è 32, dove ogni livello deve contenere un **buffer** per lo scambio di messaggi di 4096 bytes (4KB).

L'ultima specifica è relativa alla creazione di **un char device driver** per la lettura dello status dei Tag tramite *pseudofile*.

4 - Architettura di massima

Il sistema è composto principalmente da **due moduli livello kernel** (da montare a *run-time*) i quali contengono tutta la logica del sistema. I moduli sono destinati ad un **Linux Kernel** v. 4.17 *o superiori*.

I due moduli sono:
- **System Call Table Hacker** (abbrev. SCTH)
- **TAG Message Exchange** (abbrev. TAG)

Nonostante i due moduli presentano **funzionalità** diverse e non logicamente collegate, non è comunque presente nessuna *ragione pratica* per utilizzare due moduli distinti invece che inglobare le funzionalità di uno nell'altro, si è però voluto separarli e montarli come indipendenti tra loro per *sperimentare l'interazione tra moduli Kernel*.

Il modulo TAG ha una dipendenza stretta con il modulo SCTH: viene infatti utilizzato quest'ultimo per esportare le funzionalità del modulo TAG al livello *userspace*. Questa dipendenza viene riflessa nella fase di montaggio e smontaggio dei moduli grazie all'ordine con cui vengono effettuate le operazioni.

Ogni modulo presenta una funzionalità di **Debug**: è infatti possibile impostare delle *print* aggiuntive sul *Log Kernel* per una migliore visione sul funzionamento interno dei moduli. Questo comportamento è gestibile da *variabile di ambiente* impostata dallo *script per il montaggio dei moduli:*
all'interno del file `TAG-AOS/start.sh` è possibile trovare la seguente dicitura alla riga 6:

```
            . . .

# set 1 to enable debug/extra printing on kernel-level log buffer, 0
otherwise
export MOD_DEBUG=1

            . . .
```

Impostando la variabile a 1 saranno attivati i *print* extra, impostandola a 0 invece non saranno presenti informazioni sull'operatività interna, ma comunque riportando le informazioni critiche.

4.1 - System Call Table Hacker

Il modulo System Call Table Hacker è la componente responsabile dell'esportazione delle funzionalità del modulo principale tramite **l'installazione di nuove system calls** fruibili in *userspace*. Per ottenere questo risultato è necessario che il modulo *trovi la locazione in memoria* della *System Call Table* (posizione non nota a run-time) e ne modifichi alcune *entry* impostando la system call desiderata. Il modulo è stato sviluppato prendendo come riferimento il seguente codice[2] e modificandolo opportunamente per rispettare alcune specifiche. Nessuna particolare ottimizzazione **prestazionale** o di **spazio in meoria** è stata portata avanti in questo modulo: data la tipologia di lavoro e la frequenza di interazioni a cui è sottoposto, non è necessario concentrarsi particolarmente sulla velocità dei sui task. Ragionamento analogo è applicabile alla memoria occupata dal modulo: non essendo presente una quantità elevata di strutture dati si è scelto di non portare avanti nessun

[2] github.com/FrancescoQuaglia/Linux-sys_call_table-discoverer

lavoro di minimizzazione della memoria utilizzata, a favore di una maggiore *semplicità* e *leggibilità* del codice.
In fase di *clean up* viene ristabilito il contenuto delle entry modificate nella system call table.

4.2 - TAG Message Exchange

Il modulo **TAG Message Exchange** è invece responsabile dell'**implementazione** delle **4 system call** precedentemente descritte e del **device driver**. Tutta la logica del modulo è legata alla definizione delle system call richieste ed alle *strutture dati utilizzate* per *memorizzare* i metadati dei Tag, livelli ed altre strutture a supporto. Il modulo è **dipendente** da SCTH per l'installazione delle system call: il modulo TAG si occupa di dichiarare e definire le routine e delega l'inserimento nella *system call table* a SCTH. Durante la fase di setup viene anche inizializzato e publicato il **char device driver** per leggere lo **status** dei vari Tag services. Il modulo è stato sviluppato con particolare focus sulle **prestazioni** e la **scalabilità**: è' possibile avere svariate interazioni in parallelo **senza *conflitti*** che potrebbero portare ad *inconsistenze,* ma senza fornire una stretta serializzazione tra le operazioni concorrenti. Meno ottimizzazioni sono state fatte per quanto riguarda la **memory footprint** che il modulo causa: data la capacità dei sistemi odierni non si è ritenuto necessario limitare l'uso della memoria con tipi di dato *packed* (i quali potrebbero eventualmente portare a problemi di *false cache sharing*) o tipi di dato con *range ammissibili* più limitati (es. *short* invece che *int*). Per alcuni dati da memorizzare, come ad esempio il **buffer** responsabile di consegnare il messaggio, si è stati indecisi se fornire **maggiori prestazioni** allocando i buffer di tutti i livelli alla *creazione del Tag* oppure favorire una **migliore gestione della memoria** allocandoli dinamicamente su richiesta. Un'altra struttura che è stata sottoposta ad una decisione simile è il **livello**: si è pensato che in un caso d'uso reale, la possibilità che vengano utilizzati tutti i livelli di un Tag possa non essere certa, perciò durante le prime fasi dello sviluppo i **livelli venivano allocati** solo quando **richiesti**, ma successivamente si è scelto di adottare una soluzione più classica ed **allocarli tutti** alla **creazione di un Tag**. Come per i livelli, anche per il buffer si è scelto un'approccio orientato alle prestazioni.
Durante la **fase di cleanup** vengono rilasciate tutte le strutture dati, liberata la memoria dai relativi Tag services ed infine eliminato il *char* device; questo per non lasciare nessuna porzione di memoria allocata successivamente allo smontaggio: a differenza dei processi *userspace*, le porzioni di memoria allocate livello kernel non vengono automaticamente deallocate, e soprattutto in alcune configurazioni potrebbero non essere sottoposte a *swap-in/out*, rendendola un'operazione critica.

La dipendenza con SCTH si riflette nell'**ordine di montaggio/smontaggio** del modulo **TAG**.

5 - Dettagli implementativi

Nel seguente paragrafo vengono presentati i dettagli implementativi dei due moduli identificando le varie funzionalità all'interno dei corrispettivi file.

5.1 - SCTH

Il modulo **SCTH** è presente nella folder `TAG-AOS/syscall-table-disc`. Di seguito la sua struttura:

```
syscall-table-disc
├── Makefile
├── arch-common
│   ├── arch-common.c
│   └── arch-common.h
├── include
│   └── syscall-handle.h
├── module.h
├── ptexplorer
│   ├── ptexplorer.c
│   └── ptexplorer.h
├── scth.h
├── scth_discovery.c
└── scth_module.c
```

In **arch-common** sono racchiuse le routine **dipendenti dall'architettura** (in questo caso x86 a 64 bit) usando codice **machine-dependant.** Le funzionalità che esporta *arch-common* sono:

- Ottenere l'indirizzo della **Page table**
- **Abilitazione** del bit **Write Protection** in CR0
- **Disabilitazione** del bit **Write Protection** in CR0

In **ptexplorer** è presente la routine per navigare la **page table** la quale restituisce l'indirizzo di potenziali frame in memoria che potrebbero contenere la **system call table**. Una piccola modifica che è stata effettuata rispetto alla versione di SCTH di riferimento è il riconoscimento di pagine di **2MB** nella tabella di 3° livello come possibili candidati.

Le funzioni di **init_module** e **cleanup** sono situate nel file `scth_module.c`. Durante la fase di **inizializzazione** viene effettua la **discovery** della system call table, memorizzando in un'apposita variabile l'indirizzo di quest'ultima per renderlo disponibile nel momento dell'installazione e rimozione delle 4 system call. Nel codice è presente anche la possibilità di **installare una system call "dummy":** questa funzionalità è stata introdotta durante lo sviluppo per verificare il corretto funzionamento del modulo, ma successivamente è stata adottata per il **testing** di due **strutture dati** fondamentali nel modulo TAG che verranno discusse più nel dettaglio nei paragrafi successivi (Hashmap e Bitmask).
Le system call vengono inserite nelle entry della tabella che sono *inutilizzate.* Nonostante queste entry possano essere modificate a piacimento senza ripercussioni sul sistema, durante la fase di *cleanup* la system call table viene riportata allo stato precedente.

Il file **scth_discovery.c** presenta la logica effettiva del modulo: **ricerca** della **system call table** tramite **pattern matching** e **inserimento della system call**: quest'ultima funzione, non presente nella versione di riferimento del modulo, permette di **inserire** una **routine arbitraria** nella prima entry libera della tabella e ne restituisce il *displacement*.
syscall_insert è l'unica funzione **esportata** come **simbolo** del modulo (tramite la *macro EXPORT_SYMBOL*), ed è infatti utilizzata dal modulo TAG per inserire le 4 system call.

5.2 - TAG-Module

Il modulo TAG contiene l'effettive funzionalità che abilitano i thread a comunare tra loro tramite questo servizio *Tag-based* e l'implementazione del *char device*. I file relativi si trovano nella cartella `TAG-AOS/tag-module` e la sua struttura è la seguente:

```
tag-module
├── Makefile
├── include
│   └── tag.h
├── module.h
├── tag-dev-driver.c
├── tag-dev-driver.h
├── tag-module.c
├── tag-struct.h
└── tag-syscall.c
```

Le funzioni di **inizializzazione** e **clenup** sono presenti nel file `tag-module.c` ed è dove vengono allocate/deallocate tutte le strutture dati necessarie al corretto funzionamento del modulo, ed è anche dove viene creata/distrutta una istanza di *char device*. Nel file vengono anche definiti 4 **parametri del modulo** in **modalità Read-only**: questi parametri sono utilizzati per esportare il *displacement* nella system call table delle 4 system call installate al fine di poterli inserire dinamicamente all'interno dell'*header* file livello user.

```
module_param(tag_get_nr     ,int, S_IRUGO);
module_param(tag_send_nr    ,int, S_IRUGO);
module_param(tag_receive_nr ,int, S_IRUGO);
module_param(tag_ctl_nr     ,int, S_IRUGO);
```

Il meccanismo di rendere disponibile nell'header livello user gli offset delle system call è stato automatizzato nel **Makefile** dell'applicazione userspace: successivamente alla corretta inizializzazione del modulo (e quindi successivamente all'installazione delle system call) viene compilato l'applicativo che interagisce con il modulo TAG, passando come parametri gli offset ottenuti leggendo i parametri del modulo nella folder `/sys/module/TAGMOD/parameters`.

Le strutture dati che tengono traccia dello stato del sistema sono contenute nel file `tag_struct.h` e in `module.h.`
Tra queste strutture sono presenti in particolare **2 strutture** che servono a supportare il **corretto inserimento e rimozione** di Tag services: **Hashmap** e **Bitmask**, dichiarate in *module.h* ma **definite** in `TAG-AOS/utils:`

```
utils
├── bitmask
│   └── bitmask.c
├── hash-struct
│   └── hashmap.c
└── include
    ├── bitmask.h
    ├── common.h
    └── hashmap.h
```

L'**Hashmap** è la struttura che mantiene la mappatura **key - tag descriptor** dei Tag services stanziati in un determinato momento, mentre la **Bitmask** è la struttura che tiene traccia dei possibili **tag descriptor** da associare alla creazione di un nuovo Tag service.
Prima di proseguire con la descrizione delle due strutture è necessario menzionare che queste sono state implementate esclusivamente a **scopo didattico**: l'hashamp permette di ottenere la **mappatura** in tempistiche **O(1)** mentre la **Bitmask** riduce la **memoria** necessaria per tenere traccia dei tag descriptor liberi utilizzando **un singolo bit** per ogni tag descriptor.
Entrambe queste strutture non portano delle effettive migliorie alle performance o alla memoria utilizzata che possono essere notate in un caso d'uso reale: vengono accedute solo nel momento in cui si fa una **tag_get** (sia in caso di creazione che in caso di apertura di un Tag service) e alla **rimozione** di un Tag, le quali dovrebbero essere operazioni richieste con una bassa frequenza rispetto alle corrispettive *send e receive*, quindi l'incremento prestazionale ottenuto dall'hashamp rispetto all'impiego di un comune *array/lista collegata* non giustifica questa scelta. Anche per l'impiego della memoria si può fare un discorso analogo: supponendo di volere 256 possibili tag services, con una bitmask si occupano esattamente **256 bit** (più eventuali metadati), mentre utilizzando, per esempio, un *array di short* dove per ogni entry (da 0 a 255) è segnato se il tag descriptor è libero oppure no, si va ad occupare **256 x 8 bit = 2048 bit = 2Kb**, una quantità di memoria abbastanza irrisoria in una macchina moderna.

Nonostante ciò, sono stati comunque portati avanti dei **test prestazioni e funzionali** su entrambe le strutture per misurare l'effettivo guadagno nell'impiego di queste due strutture e verificare il corretto funzionamento sotto condizioni limite.

5.2.1 - Hashmap

Le **entry dell'hashmap** sono anch'esse dei tipi di dato complesso: **tag_table_entry_t** (definito in *tag_struct.h*):

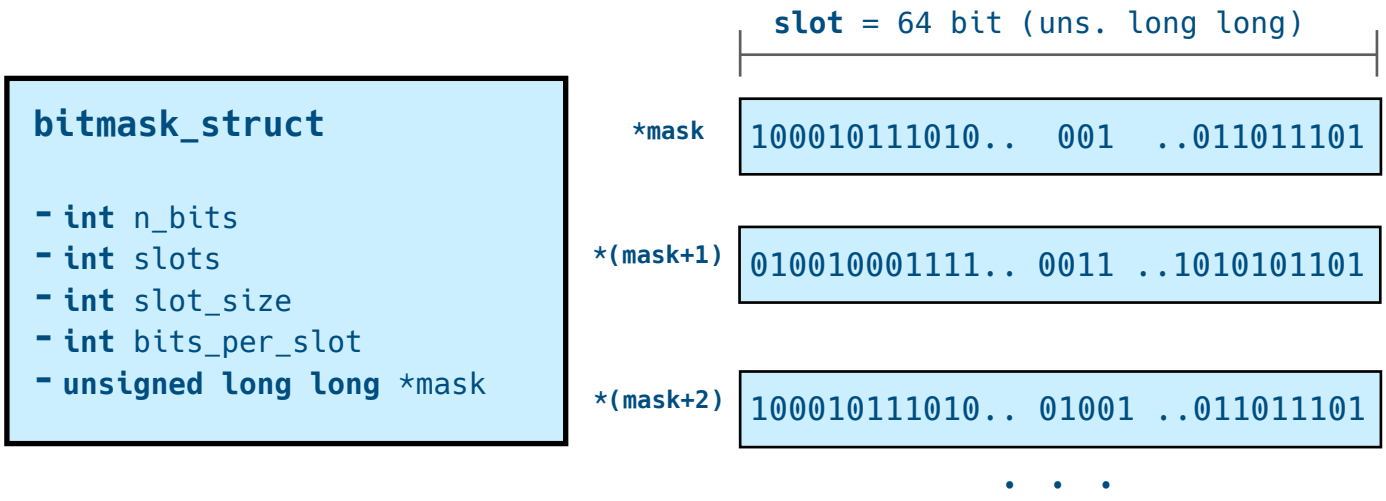
```
typedef struct tag_table_entry_struct {
    int key;
    int tag_key;
} tag_table_entry_t;
```

La **key** corrisponde alla chiave passata alla **tag_get**, ed è il campo su cui viene effettuato l'**hashing** e di conseguenza **l'indicizzazione** delle entry, mentre la **tag_key** corrisponde al **tag_descriptor** associato a quella specifica chiave. Il principale utilizzo dell'hashmap si ha durante una chiamata **tag_get** con il comando di **apertura**: quando un thread vuole aprire (e non creare) un tag service a partire da una chiave **key**, viene effettuata una *get* sull'hashmap la quale restituisce in tempi brevi il **tag descriptor** associato e lo restituisce al thread chiamante.
E' necessario menzionare che l'implementazione dell'hashmap è stata presa dalla seguente **repository[3]:** l'implementazione è stata sufficientemente studiata e testata in maniera approfondita per assicurarsi del corretto funzionamento e che rispetti le specifiche volute. Rispetto al sorgente originale sono state fatte delle leggere modifiche per **implementare un allocatore di livello kernel** e per gestire i **numeri float** (non sempre abilitati per codice livello kernel).

[3] github.com/tidwall/hashmap.c

5.2.2 - Bitmask

La **bitmask** ha la seguente struttura:



La variabile **mask** punta ad un'area di memoria *allocata dinamicamente* di dimensione pari a:

64 bit *(size singolo slot) x* ***numero*** *di* ***slot*** *necessari*.

Il singolo slot è rappresentato da **unsigned long long** ed ogni **bit** in posizione **n-esima** rappresenta se il **tag descriptor n-esimo** è utilizzabile per la creazione di un nuovo **Tag service** (impostato a 0) oppure è occupato (impostato a 1). Quando è necessario avere un tag descriptor per la creazione di un nuovo tag viene invocata la funzione **get_avail_number** (definita in **bitmask.c**) la quale scansiona *slot per slot* finché non trova un *bit impostato a 0*. Una leggera ottimizzazione effettuata è la seguente: se si verifica che *uno slot* è *pieno*, si procede subito al seguente senza controllare quello slot, abbassando di molto le tempistiche per ottenere un *tag descriptor libero;* questa verifica è fatta con il seguente statement:

```
slot_mask = *((bitmask -> mask) + i);

          . . . . .

if(slot_mask == 0xFFFFFFFFFFFFFFFF) continue;
```

5.2.3 - Strutture TAG Service

La definizione **strutture** relative ai **Tag services**, come già esposto precedentemente, sono situate nel file TAG-AOS/tag_module/**tag-struct.h** insieme ad alcune **costanti**, mentre le loro istanze sono dichiarate nel file **module.h**
Le strutture sono le seguenti:

- **tag_table_entry_t**: come precedentemente descritta, è la struttura utilizzata come **entry** nella *hashmap* (stanziata in module.h come **tag_table**)
- **tag_t** è la struttura che identifica il singolo Tag service
- **tag_level_t** è la struttura per la singola istanza **RCU** di un **livello** (stanziata all'interno di **tag_t**)

module.h

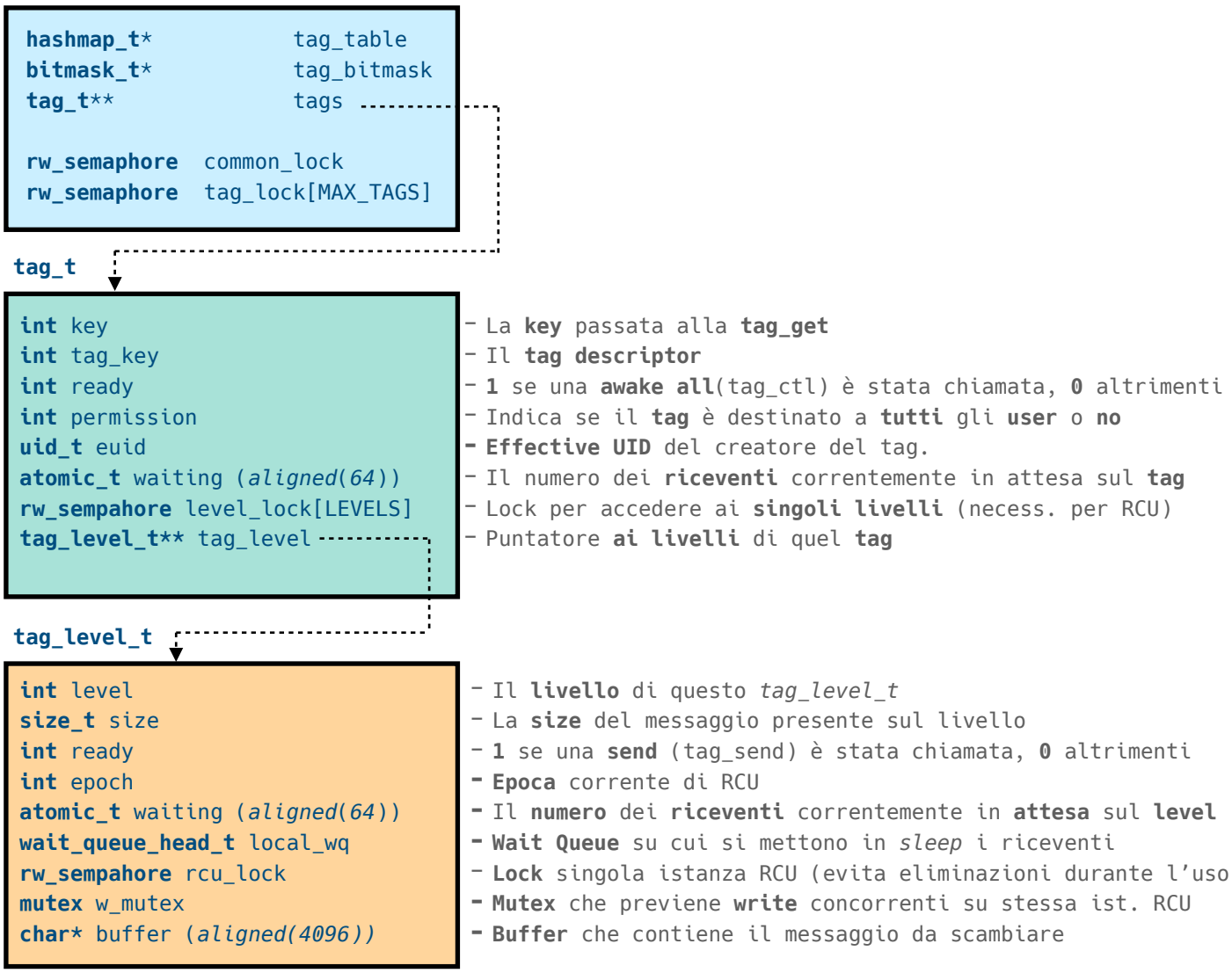
```
extern hashmap_t*           tag_table;
extern bitmask_t*           tag_bitmask;
extern tag_t**              tags;

extern struct rw_semaphore  common_lock;
extern struct rw_semaphore  tag_lock[MAX_TAGS];
```

Per garantire la corretta *lettura/scrittura* sulle varie strutture dati evitando conflitti dovuti alla concorrenza, sono stati usati anche dei **rw_sempaphore**: la scelta di usare questa tipologia di lock viene da due considerazioni principali:

- In un sistema dove *Read e Write* sono operazioni ben differenziate e non sempre accoppiate, avere un lock che distingue queste due operazioni porta un incremento di prestazioni considerevole, soprattutto perché le *Write* sono operazioni molto meno frequenti rispetto alle *Read*, lasciando quindi il sistema libero di performare senza interruzioni per la maggior parte delle istruzioni.
- Utilizzare uno *spinlock* invece che un *semaforo* avrebbe comportato un **degrado prestazionale** poiché quasi tutte le sezioni protette da questi lock presentano delle potenziali *sleep*, rendendo lo spinlock una scelta poco adeguata.

il **common_lock** è utilizzato per proteggere le sezioni di codice che interagiscono con l'**hashmap e la bitmask** (durante la *tag_get* e la *tag_ctl* in modalità *delete*).
Il **tag_lock**[MAX_TAGS] invece protegge le singole *entry* di **tags,** principalmente implementato per evitare un *accesso ad un Tag service* in parallelo ad *un'eliminazione dello stesso Tag*.
Durante la trattazione delle singole system call verrà discusso in maniera più approfondita l'effettivo impiego e cosa ne consegue.

Uno schema generale delle strutture e di come sono interconnesse tra loro:



Alcune variabili sono **allineate** per motivi di **performance:** nel caso delle *waiting*, è stato necessario allinearle alla *cache line* poiché è un dato che potenzialmente potrebbe essere acceduto in parallelo da molteplici thread, e quindi si è voluto evitare di aver un **false cache sharing** con un conseguente degrado di prestazioni; nel caso del **buffer** invece si vuole garantire che la **lettura/scrittura** avvenga prendendo in esame una **sola pagina di memoria** invece che molteplici.
I **lock** *rcu_lock, level_lock* e *w_mutex* saranno spiegati successivamente durante la trattazione delle system call, ma è opportuno menzionare che ogni lock viene preso in modalità **killable/interruptible** al fine di prevenire un possibile stallo del modulo con conseguente impossibilità di effettuare lo smontaggio.

5.2.4 - System Calls

In questo paragrafo vengono descritte le *system call* implementate nel dettaglio, commentando opportunamente le scelte fatte. Ogni system call, prima di procedere con le istruzioni effettive che le definiscono, presentano alcune righe di codice in comune per tutte:

- **Incrementano** e **decremento il contatore** relativo agli **utilizzi** del **modulo** per evitare che si possa effettuare un cleanup nel mentre si sta utilizzando il modulo
- Effettuano un **controllo** sull'input per verificare che i valori passati siano consentiti.

Le **macro** che definiscono i valori come i *comandi* o i *permessi* sono presenti nell'header *include/* ***tag.h***.

include/**tag.h**

```
#define TAG_OPEN        0
#define TAG_CREAT       1

#define TAG_AWAKE_ALL   0
#define TAG_DELETE      1

#define TAG_PERM_ALL    0
#define TAG_PERM_USR    1
```

Ogni system call **ritorna** un **valore negativo** in caso di **errore,** il cui codice segue lo *standard* di *errno* (EBUSY, EINVAL, ecc..). Per quanto riguarda invece i **permessi**, sono stati implementati in modo che l'utente **root** possa comunque accedere al Tag bypassando il controllo. Le **system call** sono situate nel file **tag-syscall.c** insieme ad alcune procedure di *utility* utilizzate internamente dal file.

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

La system call opera molto diversamente a seconda del **command:**

- **TAG_OPEN** (0):
  Verifica innanzitutto che non si ha una chiave di tipo *IPC_PRIVATE*, poiché in tal caso, la richiesta viene automaticamente respinta. In caso contrario prende il *common_lock* in modalità *read* per accedere all'Hashmap con la *key* passata in input: se nell'hashmap è presente un *mapping* **key - tag desc.** allora viene restituito il **tag descriptor**, altrimenti viene ritornato un errore. A prescindere dall'esito il *lock* viene rilasciato.
- **TAG_CREAT** (1):
  Viene innanzitutto preso il *common_lock* in modalità **write** poiché è necessario accedere alle strutture dati di *Hashmap* e *Bitmask* in maniera **non concorrenziale,** prima per verificare che l'inserimento di *un Tag service* può avvenire, e successivamente per *modificare queste strutture registrando* l'inserimento del nuovo Tag.
  Una volta preso il lock viene quindi verificato se è possibile aggiungere un nuovo Tag, e in caso affermativo viene aggiunta una nuova entry nell'Hashmap e viene settato ad 1 il bit relativo al Tag service che si vuole stanziare nella Bitmask.

Nota: Se la chiave è *IPC_PRIVATE*, non viene aggiunta nessuna entry nell'Hashmap poiché quella mappatura non sarà mai necessario accederla.
Una volta aggiornate le due strutture viene *rilasciato il lock* e si procede all'allocazione del Tag Service con i relativi livelli, per poi restituire il **tag descriptor** al thread chiamante.
Nota 2: durante **l'allocazione** del **Tag** (la sezione più lenta della routine perché include molteplici allocazioni in memoria) *non è necessario avere un lock dato che* non si accede a nessuna struttura comune. Questo ordine delle operazioni permette di poter fare molteplici chiamate a tag_get in parallelo tranne che per una breve sezione critica, lasciando la possibilità alla parte più dispendiosa della system call di operare senza nessuna restrizione dovuta alla concorrenza

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

Anche la *tag_ctl* opera diversamente sul *tag* a seconda del **command**, ma prima di procedere vengono innanzitutto controllati i **permessi** (impostati durante la TAG_CREAT) e viene eventualmente negato/consentito l'accesso.
La system call ritorna 1 in caso di successo, 0 in caso di Tag occupato (caso *delete*) oppure Tag senza nessun ricevente (*caso awake all*), oppure può ritornare un codice di errore se qualche operazione non è andata a buon fine.

- **TAG_DELETE** (1):
  Viene fatta una *richiesta* di *try_lock* al **tag_lock[tag]** (entrata *tag-esima* dell'array di rw_sem *tag_lock*) in modalità **write.** L'accesso in Write al lock ha due funzionalità: assicurarsi che nessun thread sta usando/userà durante la sezione critica il Tag in *ricezione/invio* di un messaggio, e anche di assicurarsi che non avvengano due tentativi di rimozione in parallelo.
  La **delete** *non deve essere un'operazione bloccante,* perciò l'uso di un **trylock** permette di ritornare subito il controllo al thread chiamante nel momento in cui non è stato possibile prendere il lock, invece di aspettare che tutti i thread finiscano di operare (l'attesa può anche essere lunga in alcuni contesti con molte interazioni in parallelo con il Tag).
  Una volta preso il lock, l'unica operazione che viene effettuata è quella di levare il riferimento al tag all'interno di **tag_t** tags**: impostando il riferimento a 0 nella entry relativa al tag si evita che un thread possa accedere a quella specifica istanza. Dopo aver tolto il riferimento si rilascia il *tag_lock* e si procede ad *eliminare i riferimenti* al Tag all'interno di Hashmap e Bitmask (proteggendo le interazioni con un *write lock* sul *common_lock*) per poi liberare la memoria dalle pagine allocate per il Tag service.
- **TAG_AWAKE_ALL** (0):
  La prima istruzione prende in modalità *read* il *tag_lock* relativo al *tag* specificato da input per poi rilasciarlo alla fine della routine. Il lock è preso per evitare casi di eliminazione durante una interazione con il tag. Successivamente al lock viene verificato che nessun thread sta attualmente effettuando un *awake_all* in parallelo e che esiste almeno un thread in attesa, se queste condizioni sono verificate viene impostato nell'entry del tag il valore **ready = 1**: questa è una delle 3 condizioni che i thread in attesa di ricezione controllano per poter proseguire.
  Una volta impostato la variabile, si procede a *svegliare* i thread in ricezione su quel tag *di ogni livello*.
  ready = 0 viene reimpostato dall'ultimo thread ricevente, pertanto una volta svegliati i thread è possibile ritornare il controllo all'utente senza ulteriori attese.

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

System call utilizzata per inviare un messaggio al tag **tag** sul livello **level** di dimensione **size** e con il contenuto specificato nel **buffer**. Se la send va a buon fine viene ritornato **1,** se invece il livello è momentaneamente *occupato* o senza *alcun receiver* viene ritornato **0**, o altrimenti un *eventuale* messaggio di errore. Dopo il *check preliminare sull'input e sui permessi,* viene preso il *tag_lock* relativo al tag in esame per poi rilasciarlo alla fine della routine. Successivamente viene fatta una *escalation* di lock in *read*: viene prima acceduto il *level_lock* relativo al livello passato in input per accedere alla struttura di *tag_level,* ed una volta acceduto all'istanza del livello, prendere anche il *rcu_lock*. Questi lock sono necessari per il corretto funzionamento del meccanismo di **RCU** implementato: ogni volta che viene creata una nuova *epoca*, viene sovrascritto il puntatore all'entry del livello in considerazione, e quindi è necessario assicurarsi che la struttura *tag_level* non venga acceduta durante questa sovrascrizione (questo passaggio sarà più chiaro durante la spiegazione della *tag_receive*); mentre il *rcu_lock* ha come uso esclusivo quello di fornire *sincronizzazione* nel momento in cui quell'epoca non ha più riferimenti e deve essere rimossa (anche questo passaggio sarà più chiaro durante la sezione riguardante la *tag_receive*).
Dopo aver preso i lock è possibile accedere al livello e fare eventuali controlli sui parametri delle strutture:

- Controllare se *esiste* qualche ricevente in attesa, *altrimenti* non ha senso procedere con la send
- Controllare se *esiste* una *send* che già sta operando su quel livello

Se questi controlli passano, è presente comunque un ultimo step di sincronizzazione dato da un **mutex,** in particolare il *w_mutex* (visto precedentemente nello schema delle strutture dati): questo mutex è stato introdotto poiché era comunque possibile, in un caso molto raro, che due send operanti in parallelo, vedessero entrambe il livello come *non acceduto* da un'altra send, nonostante ci fosse un'istanza in esecuzione concorrente. Il w*_mutex* viene preso tramite un *trylock*, dove il fallimento viene interpretato come un segnale di concorrenza e quindi necessario ritorno del controllo all'utente. In questo contesto, mettersi in attesa della liberazione del lock al fine di proseguire con la send non ha senso: una volta che il lock viene rilasciato, non ci sarà nessun *receiver* pronto ad ascoltare su quel livello, rendendo quindi l'attesa inutile.
In seguito all'ottenimento del lock viene operata l'effettiva **copia del messaggio**: viene copiato il messaggio contenuto in **buffer** (userspace) nel **buffer di livello,** con conseguente aggiornamento della **size**. Dopo aver completato la copia del buffer, viene impostato il *flag di livello* **ready** a **1,** questo per segnalare sia ad altri sender che il buffer è momentaneamente occupato, sia ad altri receiver che il messaggio è pronto per essere consumato.
Come ultimi *step* vengono svegliati tutti i thread *dormienti* nella **wait quelle** di livello e infine *rilasciati tutti i lock*.
Nota: il buffer viene copiato solo se la size è maggiore di 0, altrimenti vengono solo svegliati i thread in attesa sulla wait queue di livello.

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

Le **receive** è l'unica **system call bloccante** presente, ed è anche l'unica responsabile del meccanismo di **RCU**: a differenza di altre possibili implementazioni, non è la presenza di un nuovo messaggio ad iniziare una nuova epoca (poiché questa epoca, non avendo nessun receiver, porterà la send a ritornare all'utente senza nessuna consegna di messaggio), ma è invece l'ingresso di *un nuovo ricevente* nel *livello* durante la fase di lettura effettuata dai riceventi arrivati precedentemente alla consegna del messaggio.

La condizione che effettivamente fa da t*rigger* alla creazione di una **istanza di livello RCU** è il flag **ready**: se un ricevente arriva mentre il flag è impostato ad 1 significa che sta già avvenendo una lettura di un messaggio già consegnato, e quindi non destinato al ricevente in questione. Se questa condizione si avvera, il thread alloca una nuova istanza di *tag_level_t* e **sostituisce in *tag_t*** il puntatore al livello con quello della nuova epoca creata, evitando che altri thread possano accedervi. In questa prima fase viene esclusivamente levato il *riferimento*, quindi i thread che stanno operando sull'epoca ormai precedente, possono continuare a farlo senza alcun problema, mentre i nuovi thread che vorranno interagire con questo livello, vengono instradati verso l'epoca appena creata.
Il **level_lock** è necessario proprio per serializzare la modifica al puntatore del livello in *tag_t*.
Successivamente, a prescindere se si è creata una nuova epoca o meno, si procede con l'inserimento dello stesso thread all'interno della *wait queue* di livello con una **wait_event_interruptible** dove si hanno **3 condizioni** di risveglio:

- il **flag ready** di **livello** (indica che è stato consegnato un messaggio)
- il **flag ready** di **tag** (indica che è stata chiamata una *awake all*)
- un segnale di **interrupt**

La scelta di avere *una wait queue* per ogni singola istanza RCU di livello è stata fatta per avere una *selettività* maggiore per quanto riguarda "quali *thread* svegliare".
Una volta risvegliati per mezzo di una *send* o un *ctl*, si controlla quale delle 3 condizioni è stata verificata: nel caso di *ctl* o *interrupt* la routine è la stessa e viene impostato il valore di ritorno pari a 0 (in modo da segnalare all'applicativo userspace che il buffer passato in input non presenta messaggi), mentre se ci si è svegliati per mezzo di una *send* vengono eseguiti degli *step extra* per effettuare la **copia del messaggio** memorizzato nel **buffer di livello** (precedentemente scritto da una *send*).
Come già espresso, il compito di fare **reset** del **livello/tag** dopo una **send/ctl** (*reimpostare a 0 il flag ready di livello/tag e pulire i campi modificati*) è *delegato* al receiver: nel momento in cui il ricevente verifica di essere l'ultimo thread su quel livello/tag, azzera i campi e rende il livello riutilizzabile per interazioni future; ma nel caso in cui, durante questa fase di *cleanup*, viene riscontrato che **esiste uno stesso livello** ad un **epoca successiva** provvederà a **rimuovere** il **livello** liberando lo spazio in memoria. Il lock **rcu_lock** serve infatti per questo scopo: una volta verificato che si è l'ultimo thread ricevente su quel livello/epoca, questo lock viene preso in modalità **write** per assicurarsi che nessun altro thread sta cercando di accedere o sta ancora finendo di operare, ed una volta preso effettua il *reset* o la *deallocazione* a seconda del caso.
Il thread ricevente si affida ad un **contatore atomico** per sapere se è l'ultimo *thread ricevente* sul livello/tag: il campo **waiting** di **tag_t/tag_level_t,** questi contatori vengono incrementati ogni volta che si entra in *ricezione* e *decrementati* all'uscita.
La scelta di implementare una *versione propria* di RCU è stata puramente dettata dal voler sperimentare nella creazione di questo meccanismo invece di usare le API RCU di Linux. Inoltre, era possibile implementare un numero fisso di epoche (cioè istanze di livelli) e far *ciclare* le nuove istanze tra questo *pool* di possibili livelli, ma si è invece preferito non imporre questo limite e cercare una soluzione più versatile ed adatta anche a casi di alto carico.

5.3 - Char Device Driver

Il **char device driver** è l'ultima funzionalità **esposta** dal modulo **TAG**.
In **/dev** viene creato un **char device** con il nome **tag_info** il quale ritornerà lo **status** dei tag nell'istante in cui si è fatta una **read**. L'implementazione del **char device** è presente nel file **tag-dev-driver.c**. Di seguito è possibile visionare un *esempio di output* dello *pseudofile* (opportunamente minimizzato).

```
$sudo cat /dev/tag_info
```

```
|   KEY |  EUID | LEVEL |  WAIT |
+-------------------------------+
|  1337 |  1000 |     0 |    12 |
|  1337 |  1000 |     1 |   102 |
|       |       |       |       |
|  .  . | .  .  |  .  . |  . .  |
|       |       |       |       |
|  1337 |  1000 |    30 |     0 |
|  1337 |  1000 |    31 |    51 |
+-------------------------------+
|  4510 |  3000 |     0 |     0 |
|  4510 |  3000 |     1 |    37 |
|       |       |       |       |
|     .  .  .  .  .  .  .     |
```

Il char device ritorna lo status di **tutti i Tag** attualmente presenti (stanziati e non ancora eliminati), e si è scelto di far riportare anche i livelli che hanno *0 riceventi* attualmente in attesa. Tra un tag e un altro, viene messa una linea di separazione che può risultare particolarmente comoda per dividere visualmente più tag dichiarati con IPC_PRIVATE (quindi con stessa chiave pari a 0).

L'**implementazione del char device** si riduce nella sola definizione dei metodi di **read** e **llseek**, le operazioni rimanenti come *open, release, ctl* e *write* non sono state implementate poiché non vi era necessità (specialmente la Write dato che il file è stato assunto come *Read Only*).
La **read**, insieme all'ottenimento di *opportuni lock*, otterrà le informazioni necessarie scorrendo tra tutte le istanze di *tag_t* e *tag_level_t,* precedentemente descritte, per costruire un *buffer* che sarà poi consegnato al thread *user* con una opportuna chiamata a **copy_to_user.** Invece di creare uno *snapshot* all'apertura del file (tramite *open*) e restituire poi allo user la porzione del buffer interessata, si è adottato un metodo dove ogni *read* ritorna una versione aggiornata dello status.
Dato l'**offset** come **parametro** della **read,** è possibile da questo valore **scartare** a priori alcuni *tag* che si è certi non verranno riportati nella read, riducendo significativamente le tempistiche di questa operazione.

6 - Istruzioni

Per montare i **moduli** ed eseguire i **test** sviluppati (presentati nella prossima sezione) è stato creato un *apposito script* che automatizza tutto il processo: `TAG-AOS/start.sh`.
Analogamente è stato creato uno *script* che fa invece il **cleanup** dell'ambiente eliminando i *binari* e *smontando* i moduli: `TAG-AOS/clean.sh`.
start.sh esporta e usa delle **variabili di ambiente** per gestire quelli che sono le diverse opzioni offerte, è infatti necessario eseguire lo script con la seguente istruzione *bash*:

terminale bash

```
$ . start.sh
```

In questo modo, le variabili d'ambiente **saranno visibili** anche ai vari **Makefile** per differenziare le diverse modalità di compilazione. Le variabili di ambiente sono le seguenti:

- **TEST_FUNC**: Se impostata ad **1**, invece di montare i due moduli, verrà semplicemente eseguito un test **funzionale** sulle strutture dati di Hashmap e Bitmask, mentre se impostato a 0 questo test non verrà eseguito e si procederà con l'installazione dei moduli
- **TEST_SYSCALL:** Se impostata ad **1** verrà montato **esclusivamente** il **modulo di SCTH** e verrà installata una **syscall dummy** contenete altri **test funzionali** su Hashmap e Bitmask, con un annesso applicativo che chiama la system call in questione; mentre se impostata a 0, verrà montato il modulo SCTH senza nessuna system call dummy e si procede con il montaggio del modulo di **TAG** e l'**esecuzione** dei **test** sui Tag services
- **MOD_DEBUG**: come già espresso nell'introduzione, se questa variabile è impostata ad 1 vengono attivate delle *print extra* utilizzate in fase di sviluppo, altrimenti vengono riportate esclusivamente stampe di carattere critico.

7 - Test funzionali e prestazionali

I test sono stati parte **integrante dello sviluppo**: non solo vengono usati come **benchmark** e **dimostrazione** del funzionamento delle funzionalità implementate, ma sono stati utilizzati anche durante tutta la fase dello sviluppo per provare che ad ogni modifica il sistema reagisse con dei comportamenti aspettati.
Non essendo parte integrante del **core** del progetto, non è stata riposta particolare attenzione allo **stile** del codice dei test: si è cercato di renderli **più funzionali e leggibili** piuttosto che **ridurne** le linee di codice **accorpando** istruzioni simili in **routine separate**.
Sono stati condotti due tipologia di **test:**

- Test relativi alle strutture di **Hashmap e Bitmask**
- Test relativi all'**interazione con i moduli**

I test possono essere trovati nella cartella TAG-AOS/**test** e presentano un sistema semplice di **asserzione automatica** che verifica, tramite il ritorno delle funzioni chiamate, se il comportamento atteso è stato rispettato.

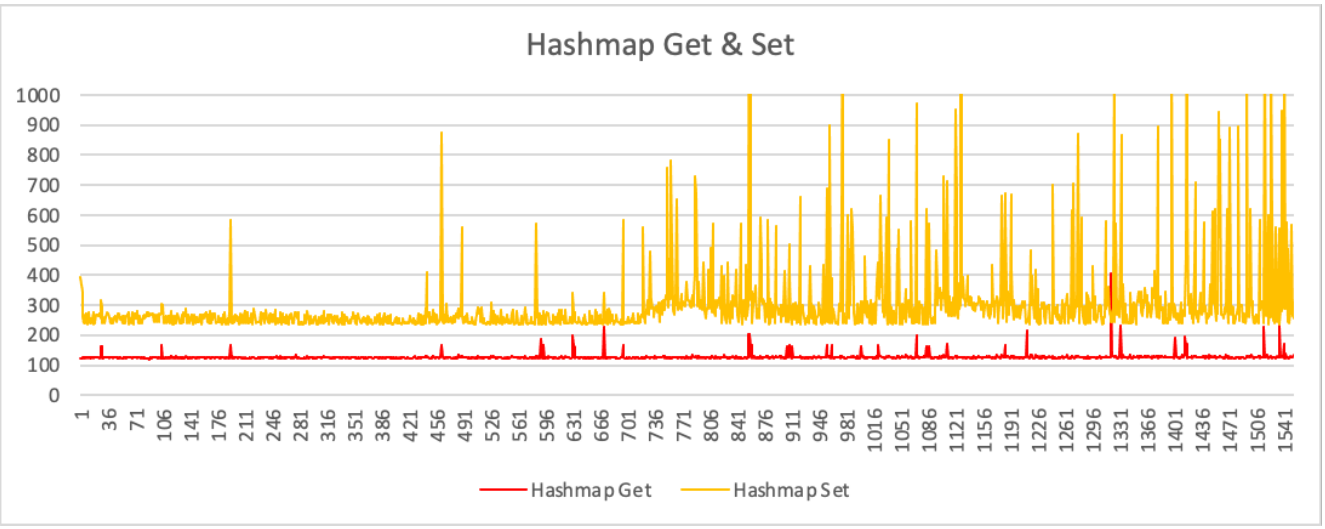
7.1 - Hashmap e Bitmask

Per assicurarsi del corretto funzionamento di queste due strutture, sono stati eseguiti dei test scrupolosi al fine di provare sia utilizzi *standard* che *casi limite.* I test sono implementati nel file **test_func.c** e nella **dummy system call:** il test è stato duplicato sia in ambiente *userspace* che in *kernelspace* per verificare che non ci fossero problematiche inaspettate nell'impiegare queste strutture in ambito **kernel**, inoltre il test sulla **dummy system call** è stato utilizzato anche nelle fasi iniziali dello sviluppo per verificare che **l'installazione** di una system call proceda senza errori.

Per verificare le effettive performance su queste strutture sono stati fatti dei **test prestazionali** usando sia **rdtsc** che **jiffies**.

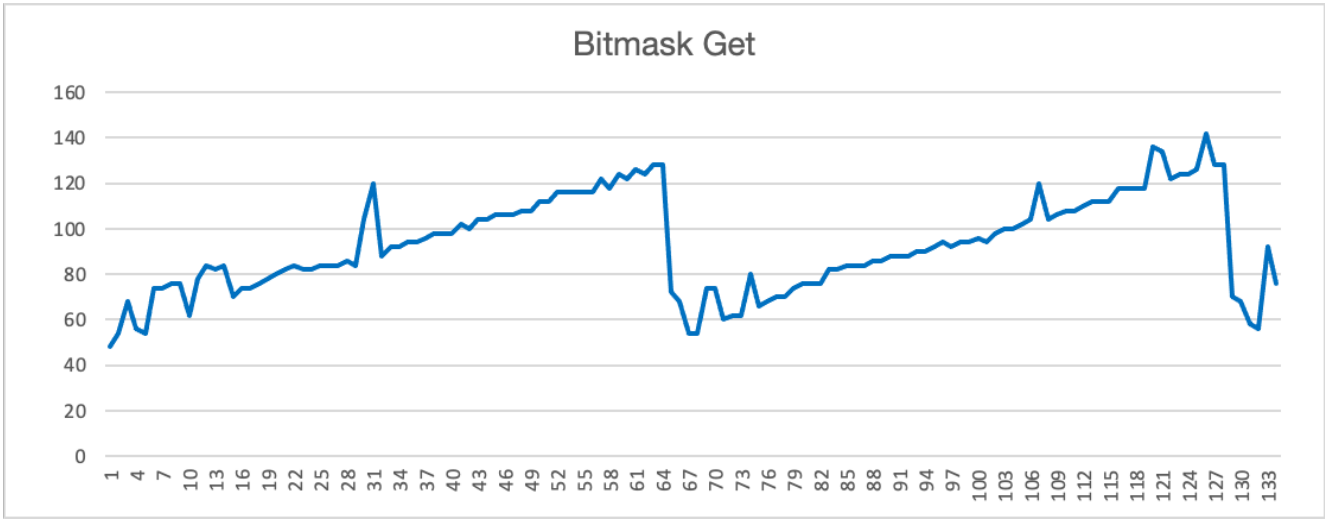
Per quanto riguarda l'**Hashmap** le tempistiche misurate sono risultate molto promettenti: tramite **rdtsc** è stato verificato che il tempo per *accesso e scrittura* è di fatto **O(1)**, non è infatti presente una crescita in questi tempi all'aumentare degli elementi.

Tramite **jiffies**, non essendo indicato per tempistiche inferiori ai millisecondi, il tempo riportato per effettuare 1600 **get** e **set** sull'hashmap è circa *di 1 jiffy* (circa 1 msec)
E' comunque possibile estrapolare da **rdtsc** un tempo di esecuzione approssimativo grazie alla conoscenza che **il processore** utilizzato per effettuare i test ha una frequenza di **4.9 GHz** *costanti*.



Nonostante le oscillazioni presenti, soprattutto nella **set**, è evidente come al crescere della dimensione dell'Hashmap non è presente un pattern di incremento del tempo necessario ad effettuare le due operazioni, ma solo di una maggiore instabilità, pur sempre rimanendo in un range di **timestamp counter** che non supera le 2000 unità (eccetto alcuni *outlier*).

Situazione analoga per quanto riguarda la **Bitmask**: il tempo misurato tramite *jiffies* non è sufficientemente grande da poterlo ritenere preciso (*sotto il singolo jiffy*) e si è pertanto preso in esame solo il tempo con **rdtsc**. L'ottimizzazione che permette di saltare di slot in slot se questi sono già pieni, ha permesso di mantenere il tempo per effettuare l'operazione di *get* abbastanza costante, portandolo ad un minimo se il primo bit di uno slot è 0 fino a raggiungere un massimo quando la maschera è piena eccetto che per l'ultimo bit.



Il grafico mostra sull'asse x *la chiamata get n-esima* e sull'asse y il *timestamp counter*. L'andamento a salire per poi riascendere è dato dal meccanismo a *slot* come già descritto.

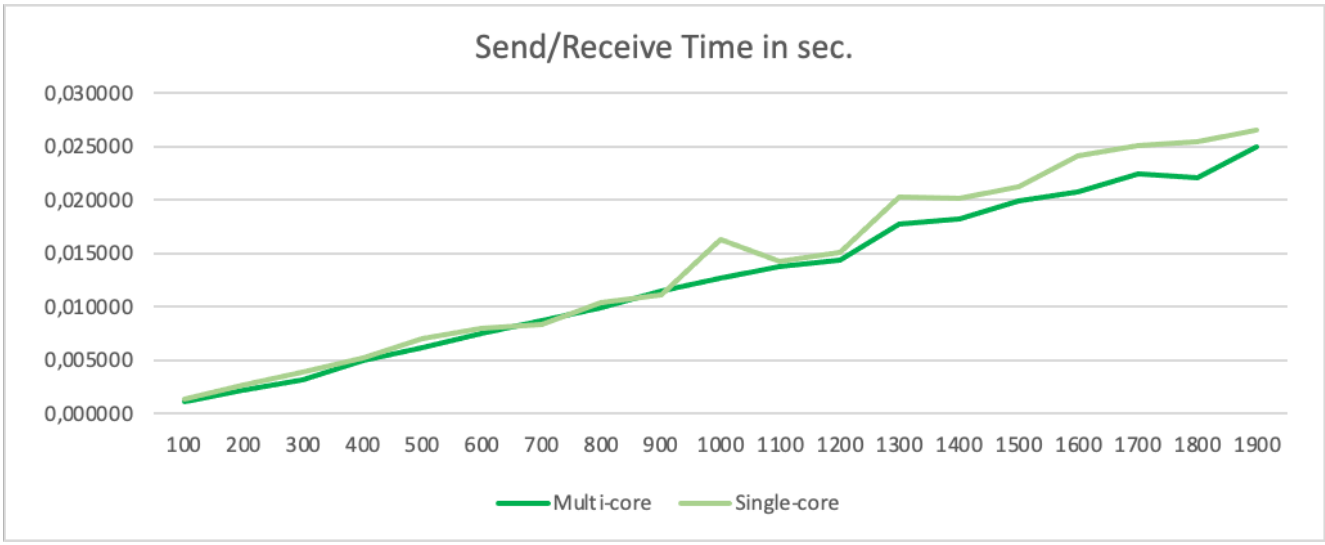
7.2 - Test TAG Service

Per provare che il modulo Tag con le relative system call funzioni correttamente con tutte le possibili variazioni nei casi d'uso e con un alto livello di concorrenza, sono stati fatti dei test che cercano di provare quante più casistiche limite e di stressare il sistema alla ricerca di potenziali **deadlock** e malfunzionamenti. A riguardo dei **deadlock**, lo stesso test sul TAG viene eseguito in due modalità: una con il **multi core** abilitato e una con modalità **single core** (ottenuta tramite *taskset*).
Il test è definito in **test_tag.c** ed è eseguito in parallelo ad un **altro test**: **test_char_dev.c,** il quale consente la lettura dello pseudofile in maniera iterativa fino all'arrivo di un *segnale di interrupt*.
Lo *script di start,* una volta arrivato alla fase di testing, apre un nuovo terminale ed esegue il **test_char_dev.c** in modo da poter eseguire in parallelo sia i test funzionali sulle 4 system call, sia di mostrare in tempo reale il funzionamento del char device con il relativo status dei Tag stanziati.
I test prevedono un minimo di interazione con lo user e forniscono su *console* una descrizione delle operazioni che stanno per essere eseguite.
Ogni test, alla fine della sua esecuzione, *pulisce l'ambiente* eliminando tutti i Tag services stanziati durante l'esecuzione. **All'interno** dei test sono stati verificati anche dei **piccoli sottocasi** per provare più casistiche possibili (es. ricezione/invio su un livello maggiore del massimo consentito oppure l'invio/ricezione con size diverse/pari a 0)

I test che vengono eseguiti sono i seguenti:
- Chiamate a **tag_get** in modalità **open** e **create** insieme a **chiamate** di **delete.**
- Creazione e apertura di **Tag privati** misti a **Tag publici**
- Test di chiamate **Ctl** ed invio di segnali di **interrupt** su un **tag occupato** da uno o più **riceventi**
- Test di scambio **messaggi** tra più **riceventi** e più **sender**

- Test sui **permessi** variando l'EUID del thread chiamante
- **Stress test** che sottopone il modulo Tag a molteplici richieste di *send/receive* in parallelo sullo stesso livello e tag
- **Time test** per misurare il tempo nel caso di 1 sender e N receivers

Le performance misurate nel **Time test**, al variare del numero dei receivers, è riportato nel grafico sottostante, sia per l'esecuzione in contesto *multi-core*, sia in contesto *single-core*.



Sull'asse x sono visualizzati il numero di thread *riceventi*, mentre sull'asse y il *tempo* (espresso in secondi) registrato per il completamento di tutte le *receive* successive ad una singola *send*.

8 - Considerazioni finali

In questo ultimo paragrafo vengono presentate brevemente alcune idee per possibili migliorie o sviluppi futuri del sistema.

- Utilizzare meno livelli o allocarli su richiesta per utilizzare meno memoria e rendere la creazione di Tag services più veloce
- Utilizzare un approccio *wait-free* o *lock-free* per lo sviluppo delle system calls con conseguenti miglioramenti di prestazioni
- Misurare empiricamente la differenza di prestazioni tra Hashmap/Bitmask e supporti diversi
- Utilizzare delle **wait queue** con *timeout* definibile dall'utente
- Bitmask con slot di dimensione più piccola per migliorare performance
- Diminuire *memory footprint* con strutture dati più compatte
- Estendere il modulo per versioni del Kernel precedenti a 4.17.0