Relazione Progetto SOA A.A.19-20

***Abstract*—La possibilità di compilare parti del kernel Linux sotto forma di modulo, anche a run-time, permette agli sviluppatori di mantenere il sistema sempre aggiornato e ai neofiliti di interagire e sperimentare nuove soluzioni.**

***Keywords*—** *Kernel, Linux, Modules.*

1. STRUTTURA DEL MODULO

Il modulo CoreMess rappresenta l'implementazione di un device file per lo scambio di messaggi tra processi (o thread) a livello user viaggiando attraversando il livello kernel. I messaggi rappresentano l'unità minima di informazione e vengono materializzati nel dispositivo secondo l'ordine FIFO:

- i *readers* ricevono il primo messaggio disponibile in testa alla coda;
- i *writers* postano un nuovo messaggio in coda agli altri.

1.1 Parametri

Il device file è caratterizzato da due parametri di configurazione, i cui valori possono essere modificati accedendo in modalità root ai corrispondenti pseudo file esposti nel file system \sys. Infatti i permessi associati sono **S_IRUGO|S_IWUSR**, che permettono l'accesso in scrittura e lettura al super user e l'accesso in sola lettura da parte di tutti gli altri utenti.

All'interno del codice sorgente del modulo sono definiti come:

```
static unsigned long max_message_size =
    MAX_MESSAGE_SIZE;
module_param(max_message_size, ulong,
    PERM_MODE);

static unsigned long max_storage_size =
    MAX_STORAGE_SIZE;
module_param(max_storage_size, ulong,
    PERM_MODE);
```

I valori di defualt sono definiti nel file *config.h*.

1.2 Montaggio e smontaggio

In fase di montaggio del modulo, viene invocata la funzione *__init* in cui vengono inizializzate le strutture globali del device file che rappresentano una singola istanza del dispositivo. Il numero massimo di istanze che è possibile creare coincide con il valore del *Minor Number*, un parametro non configurabile a run-time definito sempre all'interno del file *config.h*.
Al termine della creazione delle strutture, viene assegnato dinamicamente il valore del *Major Number*.

In fase di smontaggio, invece, viene invocata la funzione *__exit* in cui vengono deallocate tutte le strutture associate alle istanze. Infine, invocando la funzione *unregister_chrdev* il modulo viene completamente rimosso.

1.3 File Operations

La struttura delle *file_operations* permette di definire le funzionalità del modulo che verranno esposte agli utenti per l'accesso ai dati e la modifica dello stato del device file. Oltre alle classiche (apertura, chiusura, lettura, scrittura), è possibile invocare dei segnali di controllo per comunicare con il device file.

```
static struct file_operations fops = {
        .owner = THIS_MODULE,
        .open = dev_open,
        .release = dev_release,
        .read = dev_read,
        .write = dev_write,
        .unlocked_ioctl = dev_ioctl,
        .flush = dev_flush,
};
```

- **Open**:
  1. viene allocata e inizializzata la struttura che rappresenta una nuova sessione;
  2. il valore del puntatore alla nuova sessione viene assegnato al pointer *private_data* della *struct file*;
  3. la nuova sessione viene aggiunta nella linked_list delle sessioni relative ad uno specifico *Minor Number*.
- **Release**:
  1. vengono deallocate le strutture dati e i puntatori relativi ai campi della sessione;
  2. la sessione viene rimossa dalla linked_list delle sessioni attive relative ad uno specifico *Minor Number*;
  3. viene deallocata la sessione stessa.
- **Write**:
  1. viene allocato un *message_t*;
  2. viene invocata la funzione **copy_to _user** e aggiornata la struttura appena creata con i relativi dati;
  3. viene controllato il valore del write_timer: se è maggiore di zero è stata richiesta dall'utente una deferred write. Altrimenti viene direttamente invocata la funzione statica *__add_new_message* in cui viene aggiornato lo stato globale dell'istanza del device:
     - incrementando la size globale del device;
     - incrementando il contatore atomico corrispondente al numero di messaggi disponibili;
     - aggiunto il nuovo messaggio alla linked_list dei messaggi salvati nel device, rendendolo disponibile alla lettura.

  L'acquisizione del valore del timer avviene in sezione critica per evitare inconsistenze dovute ad eventuali accessi concorrenti: un thread potrebbe invocare una IOCTL e accedere alla variabile già in utilizzo da parte di un altro thread appartenente alla stessa sessione.

  Il valore di ritorno corrisponde al numero di byte scritti.

- **Read**:

  1. viene controllato il valore del read_timer: se è maggiore di zero è stata richiesta dall'utente una deferred read. Altrimenti viene direttamente invocata la funzione statica *__send_first_message*, in cui:
     - viene selezionato il primo messaggio in testa alla coda;
     - viene inviato all'utente il testo del messaggio della dimensione richiesta;
     - viene aggiornato lo stato globale dell'istanza del device:
       * rimuovendo dalla coda dei messaggi il *message_t* appena inviato;
       * decrementando la size globale del device;
       * decrementando il contatore atomico corrispondente al numero di messaggi disponibili.

  Anche in questo caso, il valore del timer viene acceduto in sezione critica.

  Il valore di ritorno corrisponde al numero di byte letti.

- **Unlocked_ioctl**:
  in base alla MACRO invocata vengono:

  - settati in sezione critica i valori dei timer dei reader e dei writer per la sessione corrente (SET_SEND_TIMEOUT e SET_RECV_TIMEOUT);
  - eliminati tutti i messaggi pendenti su uno specifico minor (REVOKE_DELAYED _MESSAGES);
  - eliminati tutti i messaggi non ancora materializzati e sbloccati tutti i reader ancora in attesa su uno specifico minor (FLUSH_DEF_WORK).

- **Flush**: Facendo riferimento ad uno specifico minor:

  1. vengono sbloccati tutti i reader ancora in attesa dello scadere del proprio *read_timer*;
  2. viene invocata una flush sulla *work_queue*;
  3. vengono rimossi i messaggi pendenti con le corrispondenti *pending_defwrite_structs*.

1.4 Deferred Operations

- **Deferred Write**:
  L'implementazione delle deferred write sfrutta le *work_queue*, in particolare la *queue_delayed_work*, che permette di schedulare l'esecuzione di un task allo scadere di un intervallo di tempo espresso in *jiffies*. Infatti, ogni task è rappresentato all'interno della coda attraverso la struttura (*struct delayed_work*) contenente i dati che verranno utilizzati durante la sua stessa esecuzione.

  Il deferred work che deve essere eseguito è definito staticamente (*__deferred_add_new_message*) e tutti i dati sono racchiusi all'interno della struttura *pending_write_t* incapsulata all'interno della *struct delayed_work*.
  Ogni volta che viene invocata una write con un *write_timer* diverso da zero:

  1. viene allocata e fillata una struttura *pending_write_t*;
  2. viene associato ad un deferred work il puntatore all'area di memoria che corrisponde al codice della funzione da eseguire al risveglio;
  3. viene aggiunto il nuovo task alla *work_queue*.

- **Deferred Read**:
  L'implementazione delle deferred read si basa sulla *wait_event_interruptible_hrtimeout* che permette di invocare una sleep sul thread in esecuzione per un intervallo di tempo in *jiffies* espresso in *ktime_t*. Infatti ogni reader con *reader_timer* diverso da zero viene "messo a dormire" sulla coda in attesa di un evento di risveglio come:

  - num_pending_read > 0, è stato materializzato un nuovo messaggio ed è disponibile per la lettura;
  - flush_me == true, è stato invocato l'**IOCTL** di tipo **FLUSH_DEF_WORK** per sbloccare tutti i reader ancora in attesa;
  - timer expired, è scaduto il timer e, nel frattempo, non è stato materializzato nessuno nuovo messaggio.

  La scelta della tipologia della coda è legata alla facilità di gestione dei vari flussi di esecuzione in base ai possibili valori di ritorno.

1.5 IOCTL

La comunicazione tra gli utenti e il device file può avvenire attraverso l'utilizzo delle seguenti MACRO invocando la system call **ioctl**:

- **SET_SEND_TIMEOUT**, che permette di definire il valore dell'intervallo di tempo da attendere prima di materializzare nel device file un nuovo messaggio, rendendolo disponibile alla lettura, generando una deferred write;
- **SET_RECV_TIMEOUT**, che permette di definire il valore massimo dell'intervallo di tempo che un reader deve attendere per poter leggere un nuovo messaggio, generando una deferred read;
- **REVOKE_DELAYED_MESSAGES**, che permette di rimuovere tutti i messaggi non ancora materializzati per uno specifico minor;
- **FLUSH_DEF_WORK**, che permette di interrompere l'attesa dei reader e rimuovere evenutali messaggi ancora in attesa di materializzazione per uno specifico minor.

I valori scelti per identificare i nuovi comandi devono essere univoci e non possono essere scelti casualmente, poichè potrebbero coincidere con altri già presenti all'interno del kernel associati ad altri moduli. Per questo si utilizza la funzione *_IO(a,b)* per calcolare il parametro, dove *a* e *b* possono essere numeri o caratteri combinati in modo da evitare l'overlap con valori già registrati.

```
1 //ioctl commands
2 #define IOC_BASE_NUM 'k'
3 #define SET_SEND_TIMEOUT
4     _IO(IOC_BASE_NUM, 0)
5 #define SET_RECV_TIMEOUT
6     _IO(IOC_BASE_NUM, 1)
```

```
#define REVOKE_DELAYED_MESSAGES
    _IO(IOC_BASE_NUM, 2)
#define FLUSH_DEF_WORK
    _IO(IOC_BASE_NUM, 4)
```

2. ARCHITETTURA

2.1 Strutture aggiuntive

Per mantenere le informazioni relative alle diverse istanze del dispositivo e gestire gli accessi concorrenti degli scrittori e dei lettori, sono state create le seguenti strutture, definite all'interno del file *new_structures.h*.

2.1.1 message_t

```
typedef struct message_t{
    char* text;
    size_t len;
    struct list_head next;
}message_t;
```

L'unità minima di storage del device file è rappresentata da un messaggio implementato attraverso questa struttura contentente le informazioni di base che lo descrivono, ovvero il testo del messaggio e la sua dimensione.

2.1.2 read_subscription_t

```
typedef struct read_subscription_t{
    bool flush_me;
    struct list_head next;
}read_subscription_t;
```

Tutti i readers che intendono effettuare una deferred read allocano una struttura di tipo *read_subscription_t* per notificare la loro presenza all'interno dell'istanza del device ed essere rintracciabili quando devono essere svegliati in caso di flush o FLUSH_DEF _WORK: il booleano contenuto all'interno della struttura per la sottoscrizione è la condizione di risveglio in caso di attesa sulla *wait_queue*.

2.1.3 pending_write_t

```
typedef struct pending_write_t{
  int minor;
  message_t* pending_message;
  struct delayed_work the_deferred_write_work
  struct list_head next;
}pending_write_t;
```

Ogni work_queue utilizza una struttura dati specifica, di tipo *struct work_struct*, in cui è possibile racchiudere tutti i dati necessari per completare il deferred_work, in questo caso inglobati a loro volta all'interno della *pending_write_t*. Questa struttura è unica per ogni invocazione di una deferred write.

- *minor* è lo spiazzamento nell'array delle istanze del device, che identifica la singola istanza del device;
- *message_t* è il puntatore al messaggio che verrà materializzato allo scadere del timer;
- *struct delayed_work* è il puntatore all'indirizzo di memoria in cui è stato definito staticamente il codice che verrà eseguito allo scadere del timer.

2.1.4 single_session

```
typedef struct single_session{
  struct mutex operation_mutex;
  unsigned long write_timer;
  ktime_t read_timer;
  struct list_head pending_defwrite_structs;
  struct workqueue_struct* pending_write_wq;
  struct list_head next;
}single_session;
```

Ogni nuova sessione viene creata in fase di *open* del device e contiene:

- un *mutex* necessario per la sincronizzazione dei thread appartenenti allo stesso processo che utilizzano lo stesso file descriptor;
- la variabile associata al *write_timer*;
- la variabile associata al *read_timer*;
- il puntatore alla linked_list delle strutture dati utilizzate dalle deferred write;
- una *workqueue* utilizzata per l'accodamento delle deferred write.

2.1.5 device_instance

```
typedef struct device_instance{
  unsigned long actual_total_size;
  struct list_head stored_messages;
  struct list_head all_sessions;
  struct mutex dev_mutex;
  long num_pending_read;
  wait_queue_head_t deferred_read;
  struct list_head readers_subscriptions;
}device_instance;
```

Ogni *Minor Number* corrispone ad una possibile istanza del device file, contentente:

- la size totale del device file;
- un puntatore alla linked_list dei messaggi effettivamente salvati nel device;
- un puntatore alla linked_list delle sessioni attualmente attive;
- un mutex per la sincronizzazione degli accessi;
- un contatore atomico utilizzato come condizione di risveglio dei readers quando è diponibile un nuovo messaggio;
- una wait_queue su cui dormono i reader con read_timer diverso da zero;
- un puntatore alla linked_list delle sottoscrizioni dei readers.

Questa struttura è unica per ogni coppia di (Major, minor), ma può essere utilizzata da diverse sessioni contemporanee. Le strutture utilizzare per rappresentare i deferred readers sono state posizionate all'interno della struttura che rappresenta il minor per evitare di dover acquisire i mutex delle singole sessioni nel caso venga invoca la flush attraverso una ioctl con MACRO **FLUSH_DEF_WORK** oppure indirettamente chiudendo un file descriptor associato a quel minor.

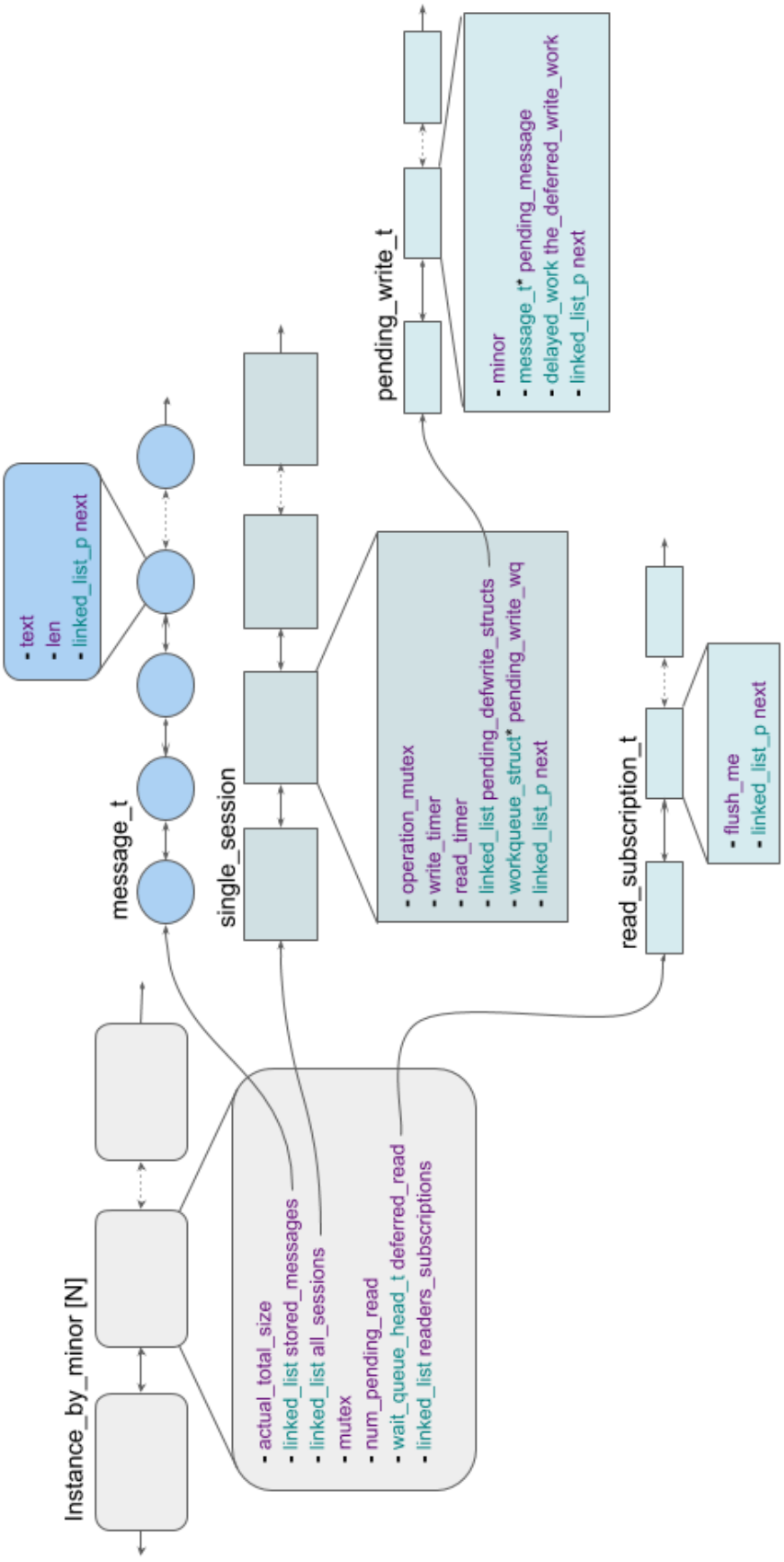

Figure 1. Rappresentazione dell'architettura dell'implementazione del modulo.