DEVICE DRIVER PER LA GESTIONE DI MESSAGGI UTENTE

Sistemi Operativi Avanzati
Università degli studi di Roma Tor Vergata

Elisa Venditti
matricola 0315909

A.A 2022/23

Indice

1 Struttura generale **1**
1.1 Device driver 1
1.2 Filesystem 2
1.3 Relazione tra filesystem e device driver 3
1.4 Cache 4

2 Gestione dei messaggi utente **5**
2.1 Layout del blocco 5
2.2 Strutture del kernel 5
2.3 Sincronizzazione 6
2.3.1 dev_read 7
2.3.2 get_data 7
2.3.3 put_data 8
2.3.4 invalidate_data 9

3 Considerazioni sulla concorrenza **10**
3.1 Inserimento 10
3.2 Invalidazione 11

4 Codice utente **12**

Capitolo 1

Struttura generale

Il progetto prevede la realizzazione di un device driver per la gestione a livello di blocco di messaggi utente. Le funzionalità offerte dal driver devono essere in parte supportate dal VFS (in modo da accedere al device come se fosse un file) e in parte non supportate dal VFS. Queste ultime riguardano le operazioni di inserimento, lettura ed eliminazione di blocchi del device. Tutte le funzionalità sono state accorpate in un modulo del kernel Linux. Il software di inizializzazione del modulo, quindi, si occupa di registrare:

- le system calls;
- il file system;
- il char device driver con le file operations implementate

oltre ad eseguire l'inizializzazione di strutture dati per la gestione dei messaggi (vedere 2.2).

1.1 Device driver

Il driver permette l'accesso al dispositivo tramite file operations e system calls. Altri driver nel kernel di Linux scelgono di aggiungere delle chiamate di sistema alla loro interfaccia per esporre funzionalità, ad esempio il generatore di numeri randomici:
https://github.com/torvalds/linux/blob/master/drivers/char/random.c.

Ho scelto di procedere in questo modo perché alcune funzionalità da implementare non devono essere supportate dal VFS: non deve esistere un'interfaccia come *write* o *ioctl* per operare sui blocchi. Se esistesse, la system call si limiterebbe a richiamarne le funzionalità utilizzando il VFS, così come avrebbe potuto fare un utente. Con la soluzione proposta, le system calls sono davvero l'*unico* punto di accesso per svolgere le funzionalità richieste sul device. Le system calls implementate sono:

- get_data - legge un certo numero di byte del messaggio relativo al blocco specificato;

- put_data - inserisce un messaggio in un blocco libero;
- invalidate_data - elimina logicamente un messaggio (lo rende disponibile per sovrascritture).

Nella loro implementazione, le syscall utilizzano il sottosistema *buffer-page chache* del VFS (tramite *sb_bread*) per poter agire sui blocchi memorizzati sul block device.

Il driver fornisce anche delle file operations che permettono al codice specifico di essere richiamato da syscall del VFS:

- dev_open;
- dev_release;
- dev_read.

1.2 Filesystem

Il file system è necessario per poter accedere al contenuto del device come se fosse un file. Viene gestito un unico file, dunque il codice presenta delle semplificazioni nell'inizializzazione delle strutture dati. Ad esempio, non sono state implementate le super operations e non è stato inserito/gestito il vettore di i-nodes. Per formattare correttamente il dispositivo con un file system viene utilizzato il codice utente presente in *mkfs_umessage.c*.

Durante la fase di lookup, le file operations esportate dal driver sono state associate all'inode del file: in questo modo, il file ci permette di accedere al dispositivo tramite le funzionalità del driver, come esplicitato nella traccia. Non essendo richiesta nessuna file operation di scrittura per modificare il contenuto del device, il file è stato reso read-only. Infatti nella *dev_open*, è presente il seguente controllo:

```
if(file->f_mode & FMODE_WRITE){
    printk("%s: FORBIDDEN WRITE", MODNAME);
    return -EROFS;
}
```

Il montaggio del file system viene effettuato con il loop device che permette di vedere il file immagine (adeguatamente formattato) come se fosse un dispositivo a blocchi. Una volta effettuato il montaggio la funzione *singlefilefs_mount* controlla:

1. la dimensione del device - il driver è in grado di supportare un massimo numero di blocchi che è possibile specificare definendo il parametro MAXBLOCKS. La dimensione del file immagine viene calcolata nel seguente modo:

```
filp = filp_open(PATH_TO_IMAGE, O_RDONLY, 0);
// ricavo la dimensione totale del file
size = vfs_llseek(filp, 0, SEEK_END);
// ricavo il numero di blocchi di dati
num_blocks = (size / DEFAULT_BLOCK_SIZE) -2;
```

2. la presenza di altri montaggi - il driver supporta un solo montaggio. Dunque, nel modulo del kernel è presente un intero che viene impostato atomicamente a 1 durante il montaggio e resettato a 0 quando il file system viene smontato.

Passati i controlli e completata la *fill* del superblocco, viene ricavata la struttura *block_ device*. Quest'ultima corrisponde al block device utilizzato dal file system ed è necessaria per ricavare il superblocco. Le file operations chiamate sull'unico file del filesystem, possono accedere al superblocco tramite il puntatore alla *struct file* ricevuto in input. Tuttavia, le syscall non sono file operations e hanno bisogno di un altro meccanismo per poter leggere i blocchi corretti dal *buffer-page cache*:

```
struct buffer_head bh = (struct buffer_head *) sb_bread (
 bdev->bd_super, block_to_read);
```

Infine, viene costruita una rappresentazione in memoria del contenuto del dispositivo utilizzando i metadati (approfonditi in 2.1).

1.3 Relazione tra filesystem e device driver

Le operazioni sul driver devono andare a buon fine solo se è stato montato un file system. Per eseguire questo controllo, tali operazioni verificano che la variabile bdev sia non nulla: bdev verrà resettata a NULL dall'operazione di umount. In questo contesto, però, bisogna prestare attenzione all'interleaving tra le operazioni, perchè potrebbero presentarsi le seguenti situazioni:

- *put_ data*, ad esempio, legge bdev, controlla che sia diverso da NULL e viene deschedulato. Prima che possa continuare, viene eseguita una umount che riporta bdev a NULL. La umount riconosce il target "busy" solo quando è impegnato da file operations. Dunque, serve un meccanismo per comunicare alla mount che c'è un'operazione pendente: uno *usage counter*.

- consideriamo la stessa situazione del punto precedente. Con il contatore si evita lo smontaggio, però il puntatore bdev viene comunque impostato a NULL (in modo da bloccare richieste successive). C'è bisogno, dunque, di una variabile temporanea che memorizzi il puntatore letto per poter operare.

```
// incrementa atomicamente bdev_usage di 1
temp = bdev
if(temp == NULL){
  // decrementa atomicamente bdev_usage
  return
}
// ...
bh = (struct buffer_head *) sb_bread(temp->bd_super,
 block_to_read);
```

Durante lo smontaggio, bdev è riportato atomicamente a NULL; poi si attende su una wait queue il rilascio del contatore da parte di tutti i thread.

1.4 Cache

Le strutture dati strettamente correlate sono state racchiuse in una stessa struttura e allineate alla linea di cache. Ne sono un esempio le variabili introdotte nella sezione precedente: bdev (che indica il block device utilizzato dal file system) e bdev_usage (che indica lo usage counter della variabile bdev) vengono utilizzate sempre insieme, sia dalle funzioni di mount e umount, sia dalle operazioni del device driver. Lo stesso ragionamento è stato seguito per altre strutture dati.

Capitolo 2

Gestione dei messaggi utente

2.1 Layout del blocco

Il blocco sul dispositivo ha una dimensione di 4KB: 32 bit sono metadati; i restanti contengono i messaggi utente. I bit per i metadati sono divisi in:

- 1 bit di validità - indica se il blocco è valido oppure se è stato eliminato logicamente;
- 31 bit per indicare il numero del prossimo blocco - è necessario per dare un ordinamento temporale ai messaggi che vengono inseriti.

I metadati nel dispositivo vengono aggiornati solo durante il montaggio e lo smontaggio del file system: servono per rendere persistente lo stato attuale dei messaggi. Questo rende più snella la gestione delle scritture sulla cache: l'inserimento deve riportare solo i dati, l'invalidazione non deve operare sulla cache. Quando il file system è montato, esistono strutture dati nel kernel per la gestione dei messaggi utente (vedere 2.2).

Durante il montaggio, viene eseguita una scansione per inizializzare le strutture dati nel kernel e per trovare l'*head* (unico elemento valido per cui non è stato trovato il predecessore) da cui partire per la lettura. L'operazione duale viene eseguita durante lo smontaggio.

2.2 Strutture del kernel

Come accennato in precedenza, la gestione a livello di blocco dei messaggi utente viene effettuata dal kernel tramite delle strutture dati:

- *struct block_ node* - metadati di gestione mantenuti dal kernel. La struttura è definita come segue:

```
struct block_node {
  struct block_node *val_next;
  int num;
};
```

L'intero *num* indica il numero di blocco a cui si riferisce la struttura. Come si nota dalla presenza di *val_ next*, le strutture *block_ node* verranno collegate e ordinate tramite una lista. È su questa lista che andranno ad agire le operazioni del device driver. Il puntatore *val_ next* non può utilizzare il bit più a sinistra perché questo indica la validità del blocco. Sapere se un blocco è valido o no è necessario per eseguire alcune operazioni di *compare-and-swap*. Ho introdotto alcune macro che agiscono su *val_ next* per: (1) ricavare il puntatore; (2) ricavare il bit di validità; (3) invertire la validità del blocco. Tali macro funzionano sui puntatori validi non nulli, dunque è stata prestata particolare attenzione per l'ultimo elemento della lista (con *val_ next*= VALID_NULL).

- *struct block_ node *valid_ messages* - lista di elementi validi. Possono esistere elementi validi che non sono presenti su questa lista: è una temporanea inconsistenza dovuta all'invalidazione. Pertanto, solo i blocchi in questa lista vengono considerati come effettivamente validi. Per evitare la gestione della lista vuota è stata inserita una *head* permanente con *head->num* = -1.

- *struct block_ node block_ metadata[MAXBLOXKS]* - array che memorizza tutti i blocchi (anche invalidi) ordinati con il campo *num*.

2.3 Sincronizzazione

Per sincronizzare i thread che operano sulle strutture della sezione 2.2 è stato seguito l'approccio RCU. I lettori non attendono le scritture, mentre le invalidazioni vengono sequenzializzate tramite l'uso di lock in scrittura. Infine, gli inserimenti non fanno uso di lock, ma eseguono le operazioni in modo atomico seguendo un approccio all-or-nothing. Se ci sono problemi di consistenza viene ritornato -EAGAIN in modo da far capire al codice utente che l'operazione non è stata portata a termine ma è possibile riprovare.

Le invalidazioni devono impedire il riuso delle aree di memoria invalidate finchè esistono lettori nel grace period (che hanno segnalato la loro presenza sulla lista ma che non hanno ancora completato). Per realizzare questi meccanismi è stata utilizzata la seguente struttura:

```
struct counter{
  // contatori rilasciati nell'epoca corrente
  unsigned long pending[2];
  // lettori nell'epoca corrente (bit a sinistra=epoca)
  unsigned long epoch;
  // indice per accedere a pending[] nella prossima epoca
  int next_epoch_index;
  // write lock sulla lista
  struct mutex lock;
};
```

Il contatore viene innalzato dai lettori su *epoch*, e poi viene rilasciato sul campo di *pending* relativo all'epoca corrente. L'invalidazione cambia epoca ed è l'unico thread a poterla modificare (in quanto possiede il write lock). Dopo aver cambiato epoca si mette su una wait queue per attendere che i lettori dell'epoca vecchia abbiano rilasciato il contatore su *pending*.

Durante l'inizializzazione del modulo viene creato un kernel thread che si occupa di aggiornare l'epoca periodicamente (per evitarne l'overflow).

2.3.1 dev_read

Algorithm 1 read device's content as a file

```
if filesystem is mounted then
    increment atomically the reader's epoch-counter
    read access to first block
    while element is not NULL do
        move to the block to read
        read from cache
        read access to the next block
    end while
    release atomically the reader's epoch-counter
end if
```

L'accesso in lettura ad un blocco non si rintraccia nella lettura vera e propria dalla cache, ma nel momento in cui si legge il puntatore al *block_ node* corrispondente al prossimo blocco. Quando mi trovo sul blocco N:

- se il blocco N+1 viene invalidato prima di averne preso il puntatore (accesso in lettura) allora il lettore non ne vedrà il contenuto;
- se il blocco N+1 viene invalidato dopo averne preso il puntatore (accesso in lettura) allora il lettore ne vedrà il contenuto. Questa lettura non tiene conto della validità del blocco: N+1, essendo sganciato dalla lista, non è più valido. Tuttavia, il lettore aveva effettuato l'accesso quando il blocco era valido, perciò deve continuare a vederlo.

2.3.2 get_data

Algorithm 2 read a block's message

```
if filesystem is mounted then
    increment atomically the reader's epoch-counter
    if block is valid then
        read from cache
    end if
    release atomically the reader's epoch-counter
end if
```

Non era richiesto l'uso dei contatori di lettura in quanto *get_ data* non scandisce la lista di elementi validi, ma accede ad un singolo blocco. Tuttavia, il contatore è stato comunque incrementato per avere la sicurezza che il blocco che si vuole leggere non verrà invalidato e riutilizzato (prima che questo lettore completi l'acquisizione del messaggio). Poichè la scrittura sulla cache che inserisce un nuovo messaggio non avviene atomicamente, *get_ data* potrebbe notare inconsistenze nella lettura e ho preferito evitare questo scenario.

2.3.3 put_data

Algorithm 3 insert message in a free block

if filesystem is mounted **then**
  get an invalid block to overwrite
  validate atomically the block
  update data in cache
  get the tail of the valid blocks
  insert atomically the new element
end if

L'inserimento avviene in coda alla lista in modo da fornire sempre un ordinamento dei messaggi. Tale system call opera secondo un approccio all-or-nothing: la validazione di un blocco e l'inserimento in coda avvengono atomicamente con delle *compare-and-swap*. In questo modo, si riesce a rilevare qualsiasi inconsistenza e a disfare le operazioni (*undo*). Le considerazioni sulla concorrenza verrano approfondite nel capitolo 3.

La scrittura sulla cache viene effettuata con una *strncpy* sul campo *b_ data* della struttura *buffer_ head*. Se viene definita la macro FORCE_SYNC, l'inserimento si occupa di forzare in modo sincrono la scrittura dalla cache al dispositivo; altrimenti se ne occupa il demone del *buffer-page cache*.

```
 // update data
 if (bh->b_data != NULL){
   strncpy(bh->b_data, message, DATA_SIZE);
   mark_buffer_dirty(bh);
 }
 // force synchronous update
#ifdef FORCE_SYNC
 if(sync_dirty_buffer(bh) == 0){
   printk("%s: SUCCESS IN SYNCHRONOUS WRITE", MODNAME);
 } else
   printk("%s: FAILURE IN SYNCHRONOUS WRITE", MODNAME);
#endif
```

2.3.4 invalidate_data

Algorithm 4 invalidate block

```
if filesystem is mounted then
    get write lock
    get the block N to invalidate
    if block is valid then
        get predecessor P
        change atomically P.next to N.next
        move atomically to a new epoch
        wait pending readers
        invalid N
    end if
    release write lock
end if
```

L'operazione di eliminazione di un blocco si materializza tramite il cambiamento atomico di P.next. Non c'è concorrenza tra le invalidazioni ma possono esistere conflitti con le scritture, che non prendono il lock (capitolo 3). Non si opera con la cache perché, come già detto, i metadati nel device vengono aggiornati solo durante lo smontaggio del file system.

Capitolo 3

Considerazioni sulla concorrenza

È importante considerare i possibili scenari di concorrenza in quanto gli scrittori non si comportano nello stesso modo: le invalidazioni si sincronizzano tra di loro con i lock, ma gli inserimenti procedono senza utilizzare lock e possono andare in conflitto con altri inserimenti o con le invalidazioni. Tutti gli scenari presentati di seguito sono stati testati.

3.1 Inserimento

Nei prossimi esempi ci si riferisce con T1 al thread che procede con l'inserimento. T1 deve: prendere un blocco da riutilizzare S, validarlo, selezionare il blocco in coda T e agganciare S a T. Durante questi passaggi può essere deschedulato e interrotto da un thread concorrente T2.

Caso 1: seleziono il blocco S e T2 mi interrompe

Operazioni di T2	Conseguenze
Inserimento che seleziona S	Se T2 riesce a validare S, il campo S.val_next ha bit di validità a 1: T1 può riconoscerlo quando tenta di validare S
Invalidazione di S	T2 fallisce perchè S al momento è invalido

Caso 2: seleziono S, lo valido e T2 mi interrompe

Operazioni di T2	Conseguenze
Inserimento in T.val_next	Non causa problemi perché T1 deve ancora cercare T. Inoltre T2 non potrà selezionare il blocco S perché già valido
Invalidazione di S	T2 non riconosce S come valido perché è scollegato dalla lista dei blocchi validi

Caso 3: seleziono la coda T e T2 mi interrompe

Operazioni di T2	Conseguenze
Inserimento in T.val_next	T2 cambia T.val_next quindi T1 riconosce la modifica quando tenta di agganciare S
Invalidazione di T	T2 cambia la validità in T.val_next e T1 può riconoscere la modifica quando tenta di agganciare S

3.2 Invalidazione

Nei prossimi esempi si analizzano i conflitti possibili durante l'invalidazione. T1 è il thread che tenta di invalidare il blocco N; P è il predecessore di N, e S è il successore. T1 deve: prendere il lock in scrittura, selezionare il blocco N da invalidare, prendere il predecessore P, scambiare P.val_next con N.val_next, cambiare l'epoca e attendere i lettori pendenti. Infine deve invalidare il blocco N e rilasciare il lock in scrittura. Durante questi passaggi può essere deschedulato e interrotto da un thread concorrente T2.

Se il blocco da invalidare non è la coda, T1 può essere interrotto dagli inserimenti senza entrare in conflitto con essi (perché operano sulla coda). Le altre invalidazioni non sono un problema perché c'è il lock in scrittura. Procediamo ad analizzare le invalidazioni della coda.

Caso 1: prima di sganciare N, T2 mi interompe

T2 tenta di inserire un blocco in coda e può farlo senza problemi. Andrà ad agire su N.val_next: è necessario che la compare-and-swap di T1 che andrà a sganciare N prenda il valore più recente di N.val_next.

Caso 2: prima di invalidare N, T2 mi interompe

Potrebbe essere problematico se T2 tentasse di riutilizzare N prima che tutti i lettori abbiano finito di leggere. Questo scenario non può presentarsi perché T1 renderà definitivamente invalido N solo dopo averlo sganciato dalla lista e aver atteso il completamento dei lettori.

Capitolo 4

Codice utente

Il codice sviluppato può essere testato (con *user.c*) digitando il codice relativo all'operazione che si vuole eseguire: oltre alle system calls, è possibile montare e smontare il filesystem nella directory corrente. Se viene definita la macro TEST, la lettura di tutti i blocchi validi del device viene ritardata di 10 secondi non appena si accede in lettura al terzo blocco. Questo è utile per simulare la concorrenza naturale che potrebbe presentarsi.

Caso di test

- definire la macro TEST;
- su una shell eseguire in ordine: make, make create-fs, sudo make mount-mod, sudo make mount-fs;
- su una seconda shell eseguire il codice utente;
- sulla prima shell avviare la lettura del file *"the-file"* (che ritarderà di 10 secondi);
- prima che la lettura termini, avviare l'opzione 6 sulla shell 2 (con il codice utente). Questo comando farà partire un certo numero di chiamate a tutte le system calls.

Una volta completate le operazioni, consultare i messaggi del kernel con sudo dmesg.

1. Lo scrittore che tenta di invalidare il blocco 0, non rilascia il blocco finchè il lettore non ha terminato.
2. Uno scrittore che tenta di inserire un messaggio, non trova nessun blocco libero finchè il lettore non termina. Infatti, l'unico invalido è il blocco 0 ma esiste almeno un lettore pendente.
3. Quando il lettore termina, continua a vedere il blocco 0 perchè era valido nel momento in cui aveva eseguito l'accesso in lettura. Lo scrittore che stava tentando di invalidare 0, può rilasciare il lock in scrittura e il blocco può essere riutilizzato per un inserimento.