Progetto Sistemi Operativi Avanzati

Luca Capotombolo

12 Maggio 2023

Contents

1 Introduzione

Il progetto è stato realizzato con lo scopo di ottimizzare la *complessità temporale* e la *complessità spaziale*. Nel progetto è stato utilizzato il *page cache* in modo da poter risparmiare sulla memoria da utilizzare. L'obiettivo è stato quello di poter gestire anche i dispositivi con un elevato numero di blocchi senza dover allocare immediatamente una grande quantità di memoria.

2 Struttura del dispositivo

Il 1° blocco e il 2° blocco del dispositivo sono rispettivamente rappresentati dal *Superblocco* e dal blocco contenente *l'inode* del file.
Dopo il blocco dell'inode, abbiamo una serie di blocchi (almeno 1) che contengono le informazioni relative alla validità dei blocchi di dati. Questi blocchi prendono il nome di *blocchi di stato*. Il numero di blocchi di stato è memorizzato all'interno del campo *num_block_state* del Superblocco. Dopo i blocchi di stato troviamo i blocchi dei dati effettivi del dispositivo.
Il campo *head_sorted_list* rappresenta l'indice del blocco che è in testa alla *Sorted List*. Questo campo verrà utilizzato per gestire l'operazione di inserimento di un nuovo messaggio, l'operazione di invalidazione di un blocco e l'operazione di lettura dei messaggi in base all'ordine di consegna.
La formattazione del dispositivo viene fatta all'interno del file *singlefilemakefs.c*. Inizialmente, si assume che tutti i blocchi del dispositivo siano liberi e, quindi, utilizzabili per memorizzare nuovi messaggi utente.

3 Strutture dati

Nel momento in cui il numero di blocchi da gestire è elevato, potrebbe non essere efficiente mantenere in memoria tutte le informazioni. Tuttavia, se i dati memorizzati all'interno dei blocchi sono presenti in memoria allora si riesce a ridurre i tempi di attesa per l'esecuzione delle operazioni. Di conseguenza, è necessario scegliere in modo opportuno quali e quante informazioni mantenere in memoria.
Le informazioni relative ai blocchi liberi (i.e., l'indice del blocco che permette di identificarlo univocamente all'interno del dispositivo) vengono mantenute *parzialmente* in memoria. Questa scelta è dovuta al fatto che, nel momento in cui si necessita di un blocco libero per scrivere un nuovo messaggio, non esiste alcuna differenza nello scegliere tra due o più blocchi liberi. All'interno di una struttura dati specifica verrà mantenuto solamente un sottoinsieme dei blocchi attualmente liberi che potranno essere utilizzati nel momento in cui un nuovo messaggio dovrà essere scritto. In questo modo, riesco a gestire in modo più efficiente i casi in cui esistono numerosi blocchi liberi all'istante di montaggio che potrebbero occupare una quantità di memoria eccessiva e potenzialmente non necessaria.
Il numero massimo di blocchi liberi che possono essere caricati nella struttura

dati all'istante di montaggio può essere fissato a tempo di compilazione in modo da gestire la memoria in base alle proprie esigenze. Inoltre, è possible fissare il numero di blocchi liberi da caricare all'istante di motaggio purché non superi il valore massimo consentito.
L'idea è quella di caricare progressivamente ulteriori indici di blocchi liberi nel momento in cui i blocchi liberi precedentemente caricati sono stati tutti utilizzati.
Infine, viene utilizzata un'ulteriore struttura dati che ha l'obiettivo di compattare le informazioni relative alla validità dei blocchi del dispositivo. L'idea è quella di utilizzare una quantità minima di memoria per mantenere le informazioni relative alla validità dei blocchi. Queste informazioni consentiranno di velocizzare l'esecuzione delle system calls.

Le *strutture dati core* utilizzate nel modulo e che mantengono le informazioni descritte precedentemente sono le seguenti:

- **Sorted List**: è la lista ordinata singolarmente collegata contenente i blocchi del dispositivo che mantengono i messaggi validi. Il puntatore alla testa di questa lista è il campo *head_sorted_list* del Superblocco che viene utilizzato per eseguire la scansione della lista, la rimozione di un blocco e l'inserimento di un nuovo blocco. Inoltre, questa lista viene acceduta dalla operazione di *read* per l'accesso al contenuto del device file in base all'ordine di consegna dei dati.
  Il collegamento tra i blocchi all'interno della Sorted List avviene sfruttando il campo *next* della struttura dati *soafs_block*. Per determinare la fine della Sorted List (oppure per capire se la lista è attualmente vuota) si utilizza il valore *num_block* per il campo *next* (o per il campo *head_sorted_list* del Superblocco) poiché è un indice di un blocco che non esiste nel dispositivo.

- **Free Block List**: è la lista singolarmente collegata contenente le informazioni relative ai blocchi che sono attualmente liberi e che possono essere utilizzati per l'inserimento di un nuovo messaggio utente. Il puntatore alla testa di questa lista è la variabile *head_free_block_list* che viene utilizzata per la rimozione e l'inserimento dei blocchi liberi.
  La dimensione della *Free Block List* può essere inizialmente regolata specificando un opportuno valore per la macro *SIZE_INIT*. L'obiettivo è quello di non avere inizialmente una quantità eccessiva di informazioni in memoria relative ai blocchi liberi. Ad esempio, se inizialmente non esiste alcun blocco valido allora potrebbe essere eccessivo mantenere gli indici di tutti i blocchi all'interno di questa struttura dati.
  Il numero degli indici dei blocchi liberi da inserire in fase di inizializzazione della lista potrebbe essere determinato in base anche allo studio della tipologia di applicazioni che utilizzano il dispositivo.

- **Bitmask**: è la struttura dati che mantiene le informazione relative alla validità dei blocchi nel dispositivo. La bitmask viene memorizzata nel dispositivo all'interno di specifici blocchi che prendono il nome di *blocchi*

di stato. L'obiettivo della bitmask è quello di mantenere le informazioni relative alla validititità di tutti i blocchi in modo compatto ed efficiente. Infatti, con un numero piccolo di blocchi di stato riesco a mantenere lo stato di un numero elevato di blocchi di dati nel dispositivo. Poiché la validità di un blocco è un'informazione che può essere codificata utilizzando un solo bit, ogni bit della bitmask codifica lo stato di uno specifico blocco.

4 Inizializzazione delle strutture dati

4.1 Free Block List

Il Superblocco mantiene l'array di indici *index_free* che memorizza i primi blocchi liberi del dispositivo. La dimensione di questo array è pari a *SIZE_INIT* che è un valore tipicamente inferiore rispetto al numero totale dei blocchi di dati nel dispositivo. Più precisamente, *SIZE_INIT* rappresenta la dimensione massima dell'array di indici mentre la dimensione effettiva è pari a *actual_size*. Il valore di *actual_size* è uno dei campi del Superblocco.
Osservo che il valore di *actual_size* è inferiore o uguale al valore di *SIZE_INIT*. In questo modo, posso stabilire con una maggiore flessibilità la quantità di informazioni relative ai blocchi liberi che dovranno essere caricate all'interno della Free Block List.
L'array *index_free* mantenuto all'interno del Superblocco insieme alla sua dimensione effettiva, consente di recuperare i primi blocchi liberi per l'inizializzazione della lista senza dover scandire la bitmask. In questo modo, posso direttamente determinare gli indici e caricarli nella struttura dati.
L'inserimento dei blocchi liberi viene fatto in testa alla lista poiché non mi interessa mantenere alcun ordine specifico per questi blocchi. Infatti, nel momento in cui devo recuperare un blocco per inserire un nuovo messaggio, non si ha alcuna differenza tra lo scegliere il blocco X oppure il blocco Y, con X diverso da Y.
Una volta inseriti i blocchi liberi nella lista, vengono aggiornate le due variabili *pos* e *num_block_free_used* che rappresentano rispettivamente la posizione nella bitmask a partire dalla quale bisogna cercare nuovi blocchi liberi e il numero dei blocchi liberi all'istante di montaggio che sono stati caricati all'interno della Free Block List.
La variabile *pos* consente di ottimizzare la ricerca di nuovi blocchi liberi da inserire nella Free Block List poiché evita la scansione di una porzione iniziale della bitmask che sicuramente conterrà informazioni relative a blocchi validi.

5 Montaggio del FS

La variabile *is_mounted* viene utilizzata per gestire la concorrenza nel montaggio e come meccanismo di sicurezza per evitare che un thread esegua le proprie operazioni quando il FS non è stato montato. Questa variabile è settata inizialmente al valore *0* poiché non abbiamo alcuna istanza di FS montata. In questo modo, i threads che desiderano eseguire le system calls vengono immediatamente bloccati.
Il thread che esegue il motaggio del FS setta atomicamente al valore *1* la variabile *is_mounted* solamente se in quel momento essa vale *0*. Più precisamente, il thread inizierà ad eseguire il montaggio solamente se il FS non è attualmente montato o non esiste alcun thread che in quel momento sta provando a montarlo.
Nel codice delle system calls viene eseguito un controllo sul valore della variabile *is_mounted*. Tuttavia, l'utilizzo della sola variabile *is_mounted* non è sufficiente per gestire il montaggio. Infatti, potrebbe accadere che il thread che sta montando il FS abbia settato la variabile al valore *1* ma non abbia ancora completato l'operazione. Per risolvere questo scenario, ho deciso di inserire una nuova variabile, che chiamo *stop*, la quale fa partire il thread solamente se il montaggio è stato concluso con successo.
Per la fase di montaggio viene utilizzata la variabile *is_mounted* per gestire la concorrenza tra differenti thread e la variabile *stop* per garantire una corretta esecuzione dei servizi da parte dei thread.

6 Smontaggio del FS

Per quanto riguarda il processo di smontaggio, viene utilizzata la variabile *is_mounted* in modo da gestire la concorrenza tra più thread che voglio smontare il FS. Il thread tenta di modificare atomicamente la variabile *is_mounted* settandola al valore *0*. Più precisamente, un thread esegue lo smontaggio del FS solamente se il FS non è stato già smontato in precedenza e non ci sono tentativi di smontaggo in corso.
Inoltre, lo smontaggio del FS *non* può avvenire finché esistono dei thread che stanno lavorando sul FS. Per gestire questo scenario, viene utilizzata la variabile *num_threads_run* che è incrementata di una unità ogni volta che un thread è intenzionato a lavorare sul dispositivo. Nel momento in cui il thread ha terminato l'operazione, oppure si è accorto che è in corso un tentativo di smontaggio, allora decrementa di una unità il valore della variabile. Lo smontaggio potrà avvenire solamente quando la variabile *num_threads_run* assumerà il valore *0*.
Il processo di smontaggio deve comprendere il flush su dispositivo delle informazioni necessarie per avere un'esecuzione corretta al montaggio successivo. Infine, è necessario deallocare le strutture dati core che sono state utilizzate per il motaggio corrente.

7 System Calls

7.1 get_data

La system call *get_data* consente di leggere il messaggio contenuto all'interno di uno specifico blocco valido. Inizialmente, vengono eseguiti dei controlli per verificare la validità del valore dei parametri passati dall'utente e per verificare se è possibile lavorare sui blocchi del dispositivo. A questo punto, il thread *reader* avverte gli altri thread dell'inizio della sua operazione di lettura. Per verificare la validità del blocco richiesto, si accede al contenuto della bitmask. Se il blocco richiesto dall'utente dovesse essere valido in quell'istante di tempo, allora è possibile effettuare la lettura del messaggio utente contenuto al suo interno; altrimenti, l'esecuzione della system call termina con errore.

7.2 put_data

La system call *put_data* consente di inserire un nuovo messaggio utente all'interno di un blocco libero. Inizialmente, eseguo dei controlli per verificare la validità del valore del parametro passato dall'utente e per verificare se è possibile lavorare sui blocchi del dispositivo. A questo punto, poiché per inserire un nuovo messaggio necessito di un blocco libero, devo verificare se tutti i blocchi del dispositivo sono stati già utilizzati. Nel caso in cui non ci siano blocchi liberi da utilizzare, evito di eseguire la ricerca dell'indice di un blocco libero. Questo scenario si verifica sotto la seguente condizione:

```
1: if ((head_free_block_list == NULL) and (num_block_free_used ==
   num_block_free)) then
2:     return -ENOMEM
3: end if
```

Infatti, se la lista dei blocchi liberi risulta essere vuota, allora tutti i blocchi che erano validi all'istante di montaggio e tutti i blocchi che erano liberi all'istante di montaggio e i cui indici sono stati inseriti all'interno della lista risultano essere utilizzati. Se oltre a questa condizione, tutti gli indici dei blocchi liberi all'istante di montaggio sono stati caricati nella *Free Block List*, allora sono sicuro che non esiste alcun blocco libero da utilizzare per l'inserimento di un nuovo messaggio.
Nel caso in cui esiste almeno un blocco libero, allora è necessario gestire la concorrenza nel recupero dell'indice di un blocco libero da utilizzare per l'inserimento del messaggio richiesto. Infatti, è possibile avere un numero arbitrario di thread concorrenti che eseguono l'inserimento di un nuovo blocco. Il recupero dell'indice di un blocco libero viene fatto tramite una rimozione in testa dalla lista *Free Block List* eseguita dalla funzione *get_freelist_head*.
Nel caso in cui esiste almeno un blocco libero, la condizione *AND* mostrata precedentemente non è soddisfatta e, di conseguenza, è possibile avere i seguenti scenari:

- La *Free Block List* non è vuota ed esistono dei blocchi che erano liberi al tempo di montaggio il cui indice non è stato ancora caricato nella lista.
- La *Free Block List* non è vuota ma gli indici di tutti i blocchi che erano liberi al tempo di montaggio sono stati caricati nella lista.
- La *free Block List* è vuota ma esistono dei blocchi che erano liberi al tempo di montaggio il cui indice non è stato ancora caricato nella lista.

L'obiettivo è quello di arrivare ad eseguire la funzione *get_freelist_head* avendo un'alta probabilità di riuscire a recuperare l'indice di un blocco libero.
Nel caso in cui la *Free Block List* è vuota ma esistono dei blocchi liberi il cui indice può essere caricato nella lista (i.e., *num_block_free_used* minore di *num_block_free*), viene implementato un meccanismo di *retry* con l'obiettivo di recuperare un insieme di indici e di inserirli all'interno della *Free Block List*. Il meccanismo di *retry* è necessario poiché gli inserimenti eseguono in concorrenza e bisogna gestire delle situazioni particolari. Per evitare di caricare un numero eccessivo di indici all'interno della *Free Block List*, la funzione *get_bitmask_block* viene eseguita all'interno di una sezione critica. Di conseguenza, se ci dovessero essere più thread che si ritrovano a dover eseguire questa funzione, il caricamento degli indici nella lista viene fatto solamente quando la *Free Block List* risulta essere effettivamente vuota. Nella funzione *get_bitmask_block* vengono fatti dei controlli per capire se caricare gli indici nella lista. Se la *Free Block List* risulta essere non vuota, allora significa che degli indici sono stati caricati in precedenza. Questo può accadere per via delle invalidazioni o per via del fatto che un altro thread in precedenza ha eseguito in sezione critica questa funzione. Se la *Free Block List* è vuota, allora bisogna controllare se esistono degli indici di blocchi liberi al tempo di montaggio che non sono ancora stati caricati nella lista. Ovviamente, se lista non è vuota oppure se non ci sono indici da caricare nella lista, allora il thread non deve analizzare il contenuto della bitmask alla ricerca di nuovi indici.
Nel caso in cui si debba caricare gli indici nella lista, il numero massimo di indici che possono essere caricati è pari a *update_list_size*. Il thread esegue la scansione del contenuto della bitmask a partire dal valore della variabile *pos*. Infatti, tutti i blocchi precedenti risultano essere occupati e, di conseguenza, sarebbe solamente uno spreco di risorse. La ricerca degli indici termina se il numero di blocchi liberi rimanenti è zero oppure se ho caricato all'interno della lista il numero massimo di indici consentito. Per via della concorrenza, al termine dell'esecuzione della funzione *get_bitmask_block* la lista potrebbe essere nuovamente vuota. A questo punto, se non esistono più blocchi liberi da caricare, viene resituito il codice di errore all'utente; viceversa, il thread prova nuovamente a recuperare gli indici dei blocchi liberi. Se il numero massimo di tentativi consentiti per il recupero di un blocco libero non dovesse essere sufficiente, allora verrebbe restituito all'utente un codice di errore.
A questo punto, viene invocata la funzione *get_freelist_head* per recuperare l'indice di un blocco libero. Se la funzione ha successo, si procede con l'inserimento del nuovo blocco all'interno della Sorted List; viceversa, viene restituito all'utente

un codice di errore.
Nella funzione *insert_new_data_block* si ha la necessità di gestire la concorrenza con le invalidazioni dei blocchi. Infatti, è possibile avere più inserimenti in parallelo ma non è possibile avere un inserimento quando è in atto l'invalidazione di un blocco. Questo scenario di concorrenza tra invalidazione e inserimento deve essere evitato poiché si potrebbero avere delle problematiche relative al raggiungimento in memoria dei blocchi a causa della rottura della lista. Per garantire l'esistenza di molteplici inserimenti e la gestione delle invalidazioni, viene utilizzata la variabile *sync_var* che è strutturata nel seguente modo:

- Il bit più significativo codifica l'assenza o la presenza di una invalidazione in corso.
- I rimanenti bit rappresentano il numero di inserimenti in corso.

L'accesso alla variabile viene fatto all'interno di una sezione critica in modo da leggerne il valore e da aggiornarla correttamente. Se non è in atto alcuna invalidazione, allora il thread segnala la sua presenza per l'inserimento di un nuovo blocco e tenta di inserire il blocco nella Sorted List. Se l'inserimento nella Sorted List fallisce, allora l'indice del blocco che si stava utilizzando viene inserito nuovamente nella *Free Block List*. Nel caso in cui l'inserimento nella Sorted List ha successo, allora si setta correttamente il corrispondente bit di validità nella bitmask.

7.3 invalidate_data

La system call *invalidate_data* consente di invalidare i dati contenuti all'interno di uno specifico blocco. Inizialmente, eseguo dei controlli per verificare la correttezza dell'indice del blocco richiesto dall'utente e per verificare se è possibile lavorare sui blocchi del dispositivo. Successivamente, si accede alla bitmask per verificare lo stato del blocco richiesto in quell'istante di tempo. Nel caso in cui il blocco richiesto risulta essere *non valido*, allora la system call termina immediatamente. Se il blocco risulta essere valido, allora viene eseguita la funzione *invalidate_data_block* per procedere con l'operazione di invalidazione.
Le invalidazioni dei blocchi vengono eseguite una alla volta. Di conseguenza, è necessario acquisire un *lock* per sequenzializzare le operazioni di invalidazioni. Ovviamente, una volta eseguita l'invalidazione del blocco, si ha il rilascio del lock per permettere l'esecuzione delle successive invalidazioni.
Similmente a quanto detto per la system call *put_data*, per evitare problematiche relative all'esecuzione concorrente di una invalidazione e di un inserimento, è necessario gestire correttamente la variabile *sync_var*. Il controllo del valore della variabile e il relativo aggiornamento vengono effettuati all'interno di una sezione critica.
Nel momento in cui non esistono inserimenti in corso e il thread che deve fare l'invalidazione riesce a comunicarlo agli altri thread tramite l'aggiornamento della variabile *sync_var*, allora viene eseguita la funzione *remove_data_block* che

ha il compito di rimuovere effettivamente il blocco dalla Sorted List. Se la rimozione ha avuto successo, allora viene settato il corrispondente bit di validità all'interno della bitmask.
Al termine del grace period vengono eseguite due differenti operazioni:

- Il blocco che è stato precedentemente invalidato e rimosso dalla Sorted List viene completamente scollegato dalla lista. Infatti, il blocco era stato precedentemente scollegato dalla lista ma continuava a puntare al blocco successivo. Questo è necessario poiché un eventuale thread che aveva precedentemente acquisito il riferimento a tale blocco, e che deve continuare a scorrere la Sorted List, deve essere in grado di poter recuperare l'elemento successivo. Se così non fosse, il thread penserebbe che la Sorted List sia stata completamente attraversata anche se in realtà ci potrebbero essere successivamente degli altri elementi.

- Il blocco che è stato precedentemente invalidato e rimosso dalla Sorted List è diventato libero. Di conseguenza, il suo indice deve essere inserito all'interno della Free Block List. L'inserimento del suo indice all'interno della Free Block List deve essere effettuato a seguito della terminazione del grace period. Se così non fosse, si potrebbe verificare lo scenario in cui un thread che sta eseguendo la lettura del blocco vada in conflitto con un thread che sta scrivendo un nuovo messaggio all'interno dello stesso blocco.

8 Device Driver

8.1 Read

L'operazione di read consente di accedere al contenuto del device file in base all'ordine di consegna dei dati. Il thread che esegue la *read* comunica agli altri thread l'inizio della sua operazione di lettura. A questo punto, viene eseguita la scansione della Sorted List partendo dalla testa della lista che si recupera tramite il campo *head_sorted_list* del Superblocco.

9 Compilazione

Per eseguire il processo di compilazione è necessario prima capire il significato di specifiche variabili all'interno del *Makefile* e all'interno dei file *header*.

9.1 Variabili di compilazione

- **NBLOCKS** *(file_system.h)*: Questa macro rappresenta il numero massimo di blocchi del dispositivo che possono essere gestiti. Questo numero tiene in considerazione anche i blocchi di stato.
- **NBLOCKS_FS** *(Makefile)*: Questa variabile rappresenta il numero totale di blocchi esclusi i blocchi di stato. Di conseguenza, ingloba al suo interno anche il *Superblocco* e il blocco dell'*inode*. Non è necessario computare a mano il numero dei blocchi di stato poiché sarà il programma *parametri* che stamperà su console il numero totale di blocchi, inclusi i blocchi di stato.
- **UPDATE_LIST_SIZE** *(Makefile)*: Questa variabile rappresenta il numero massimo di blocchi che può essere caricato ogni volta che si necessita di nuovi indici di blocchi liberi. Nel momento in cui la lista *Free Block List* è vuota ed esistono dei blocchi liberi al tempo di montaggio di cui è possibile caricare gli indici nella lista, allora vengono caricati al più *UPDATE_LIST_SIZE* indici di blocchi liberi.
- **SIZE_INIT** *(file_system.h)*: Questa macro rappresenta la dimensione massima dell'array *index_free* che mantiene gli indici dei blocchi liberi con cui inizializzare la *Free Block List* all'istante di montaggio.
- **ACTUAL_SIZE** *(Makefile)*: Questa variabile rappresenta la dimensione effettiva dell'array *index_free* che verrà utilizzato all'istante di montaggio.

I valori di queste variabili non possono essere arbitrari e, all'atto della compilazione, devono rispettare specifiche condizioni. Comunque sia, bisogna osservare che, anche se queste condizioni non dovessero essere rispettate dall'utilizzatore del modulo, il programma che esegue la formattazione effettua dei controlli sui valori di queste variabili. Qualora vengano riscontrati degli errori nelle relazioni tra queste variabili, si verificherebbe un errore durante l'esecuzione della formattazione.

9.2 Settaggio delle variabili di compilazione e del modulo

Per eseguire correttamente la compilazione e l'installazione del modulo è necessario prima settare il valore delle variabili in modo che rispettino le seguenti condizioni:

- I valori della *ACTUAL_SIZE* e della *UPDATE_LIST_SIZE* devono essere al massimo pari al valore della *SIZE_INIT*.

- Il valore della *SIZE_INIT* deve essere al massimo pari al numero dei blocchi di dati.

9.3 Lista dei comandi per la compilazione

La lista dei comandi da eseguire è la seguente:

0 Installare il modulo *the_usctm.ko* che si trova all'interno della cartella *Linux-sys_call_table-discoverer*. Dopo aver installato il modulo, posizionarsi nuovamente sulla cartella principale del progetto (i.e., uscire dalla cartella Linux-sys_call_table-discoverer).

1 **make clean**

2 **make all**

3 **make get-param**: per computare il valore del parametro PARAM per l'azione *create-fs*. Il valore del parametro viene stampato nel seguente formato:

Numero totale di blocchi inclusi i blocchi di stato da utilizzare per il device: X

4 **make create-fs**: Una volta ottenuto il valore *X*, eseguire il seguente comando:

make create-fs PARAM=X

5 **make mount-module && make mount-fs**: A questo punto, è possibile installare il modulo e montare il FS:

sudo make mount-module
make mount-fs

6 **make umount-fs**: per smontare il FS.

7 **make umount-module**: per smontare il modulo.