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*. Le strutture dati *core* del modulo sono state pensate e implementate con l'obiettivo di risparmiare memoria nel mantenere le informazioni relative ai blocchi del dispositivo, soprattutto per un numero elevato di blocchi, e, al tempo stesso, cercano di evitare un aumento eccessivo del costo temporale nell'esecuzione delle operazioni.

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.
La formattazione del dispositivo viene fatta all'interno del file *singlefilemakefs.c*. Inizialmente, si assume che tutti i blocchi del dispositivo siano liberi.

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.
Per quanto riguarda i dati mantenuti all'interno dei blocchi validi, essi vengono caricati in apposite strutture dati al momento del montaggio. Tuttavia, le informazioni relative ai blocchi liberi (i.e., l'indice del blocco che permette di identificarlo) 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 verrà mantenuto solamente un sottoinsieme dei blocchi liberi che potranno essere usati 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 dei 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 un numero di blocchi liberi qualsiasi da caricare all'istante di motaggio purché non superi il valore massimo consentito.

L'idea è quella di mantenere nelle strutture dati i messaggi memorizzati nei blocchi validi del dispositivo e di caricare progressivamente ulteriori indici di blocchi liberi nel momento in cui i blocchi 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 la ricerca dei blocchi validi all'istante di montaggio e 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 la variabile *head_sorted_list* che viene utilizzata 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.

- **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 inizializzata utilizzando il contenuto di specifici blocchi del dispositivo che prendono il nome di *blocchi di stato*. L'obiettivo della bitmask è quello di mantenere le informazioni relative alla valididtità di tutti i blocchi in modo compatto ed efficiente. Infatti, con un numero piccolo di blocchi riesco a mantenere lo stato di un numero elevato di blocchi di dati nel dispositivo. Da questo punto di vista, risulta essere una soluzione migliore rispetto a quella di mantenere la validità del singolo blocco di dati all'interno del blocco stesso. 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. La variabile che rappresenta la struttura dati della bitmask è la *bitmask*.

- **Tabella Hash con liste di trabocco**: è la struttura dati che mantiene le informazioni relative ai blocchi contenenti messaggi validi. Ogni entry della tabella punta ad una lista singolarmente collegata che mantiene specifici blocchi validi. Il numero delle liste dipende dal numero dei blocchi di dati nel dispositivo. Poiché il numero totale dei blocchi non cambia nel tempo, è possibile computare il numero delle liste in modo da avere in ogni lista al più un numero di blocchi che è logaritmico rispetto al numero totale dei blocchi nel dispositivo. Questa distribuzione dei blocchi può essere fatta semplicmente utilizzando la funzione hash:

  $$entry = i \ mod \ N$$

  dove i è l'indice del blocco mentre N è il numero totale dei blocchi di dati del dispositivo. Avendo le liste con un numero massimo di blocchi che è logaritmico rispetto a N, riesco a ridurre il costo della ricerca di un blocco rispetto a quello che avrei avuto utilizzando un'unica lista singolarmente concatenata. La variabile che rappresenta la tabella hash è *hash_table_valid*.

4 Inizializzazione delle strutture dati

4.1 Bitmask

La dimensione della bitmask dipende dal numero dei blocchi di stato nel dispositivo. La bitmask può essere vista come un array che ha una dimensione pari al numero di blocchi di stato. La entry *i-esima* dell'array punta alle informazioni di stato relative al blocco di stato *i-esimo*.
La funzione *init_bitmask* ha il compito di inizializzare la struttura dati. Le informazioni all'interno della bitmask vengono lette e modificate utilizzando rispettivamente le funzioni *check_bit* e *set_bitmask*.

4.2 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 minore 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 lista.

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 dei 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 lista.
La variabile *pos* consente di ottimizzare la ricerca di nuovi blocchi liberi da inserire nella lista poiché evita la scansione di una porzione della bitmask che sicuramente conterrà informazioni relative ai blocchi validi.
La funzione *init_free_block_list* ha il compito di inizializzare la *Free Block List*.

4.3 Tabella Hash e Sorted List

Il numero delle entry della Tabella Hash dipende dal numero dei blocchi di dati del dispositivo. Infatti, l'obiettivo è quello di ridurre il tempo di ricerca dei blocchi validi mantenendo in ogni lista (al più) un numero logaritmico di blocchi validi.
Per recuperare i blocchi validi viene utilizzata la bitmask. Più precisamente, viene recuperato il contenuto dei blocchi che hanno il bit nella bitmask settato a *1*.
Dato un blocco valido B, viene determinata la lista in cui inserire tale blocco, utilizzando il numero delle entry della tabella hash e l'indice del blocco stesso, per poi effettuare un inserimento in testa all'interno della lista individuata. Infine, il blocco valido viene inserito all'interno della lista contenente i blocchi ordinati in base al valore di un metadato del blocco. Questo metadato rappresenta la posizione del blocco valido all'interno della lista *Sorted List*.

5 Smontaggio del FS

Il processo di smontaggio deve comprendere il flush su dispositivo delle informazioni necessarie per avere un'esecuzione corretta al montaggio successivo. Più precisamente:

- Il contenuto della bitmask deve essere riportato nei blocchi di stato corrispondenti per aggiornare correttamente lo stato dei singoli blocchi. In questo modo, al montaggio successivo sono in grado di determinare i blocchi validi e i blocchi liberi.
- Il contenuto del Superblocco deve essere aggiornato correttamente. Viene computato il numero totale di blocchi liberi e vengono determinati i primi blocchi liberi che verranno utilizzati nel montaggio successivo per l'inizializzazione della *Free Block List*.
- I messaggi dei blocchi validi devono essere riportati nei corrispondenti blocchi del dispositivo. Inoltre, deve essere aggiornato correttamente il metadato che rappresenta la posizione del blocco all'interno della lista *Sorted List*.

Infine, è necessario deallocare le strutture dati core che sono state utilizzate per il motaggio corrente.

6 System Calls

6.1 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 se è possibile lavorare sui blocchi del dispositivo e per verificare la validità del valore dei parametri passati dall'utente. Viene determinata la lista nella tabella hash che dovrebbe contenere il blocco richiesto dall'utente. Questo passaggio viene fatto sfruttando l'aritmetica modulare:

$$i = offset \ \% \ x$$

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. La bitmask consente di evitare l'accesso alla lista dei blocchi nel caso in cui in quell'istante di tempo il blocco richiesto risulti non contenere dei dati validi. In questo modo, ottimizzo l'operazione di lettura evitando la ricerca del blocco nella struttura dati. Se il blocco richiesto dall'utente dovesse essere valido in quell'istante di tempo, allora è possibile effettuare la ricerca di tale blocco all'interno della lista precedentemente determinata. Tuttavia, è necessario osservare che la ricerca del blocco target potrebbe non concludersi con successo. Infatti, durante l'esecuzione dell'operazione di lettura, un altro thread potrebbe invalidare il blocco target cercato dal reader. L'utilizzo della bitmask

non assicura che i dati vengano trovati ma permette di ottimizzare la ricerca del blocco.
Al termine dell'esecuzione della funzione *read_data_block*, si avvisano gli altri thread che l'operazione di lettura è stata completata per poi risvegliare i thread all'interno della wait queue. In questo modo, l'eventuale invalidatore che era in attesa della conclusione delle operazioni di lettura può deallocare con sicurezza il blocco invalidato.
Per quanto riguarda un'analisi della concorrenza, possiamo considerare i seguenti scenari:

- **Esecuzione concorrente lettura-lettura**: ogni lettore verifica la validità del blocco richiesto dall'utente e, se valido, effettuerà la ricerca del blocco nella corrispondente lista della tabella hash.
- **Esecuzione concorrente lettura-invalidazione**: un blocco *B* invalidato da un thread *T'* non viene deallocato fino a quando il grace period non è terminato. In questo modo, sono in grado di gestire l'eventuale riferimento al blocco *B* che è stato ottenuto da un altro thread *T*.
- **Esecuzione concorrente lettura-inserimento**: l'inserimento di un nuovo blocco nella lista della tabella hash avviene in testa. Un problema che è stato affrontato per questo scenario di concorrenza è il seguente: un thread reader termina la scansione dei blocchi validi all'interno della lista della tabella hash e dichiara di non aver trovato il blocco *B* richiesto quando in realtà questo blocco è presente all'interno della lista. Questo scenario si può verificare nel seguente modo:

  1 Il thread *T* reader verifica la validità del blocco *B* e osserva che è valido per cui ha senso effettuare la ricerca del blocco all'interno della lista.

  2 Il thread *T'* esegue l'invalidazione del blocco B con la conseguente deallocazione del blocco stesso. A seguito della deallocazione, l'indice del blocco *B* viene inserito all'interno della *Free Block List* e potrà essere usato per nuovi inserimenti.

  3 Il thread *T* dichiara l'inizio della sua operazione di lettura e inizia la scansione dei blocchi nella lista.

  4 Il thread *T"* inserisce un nuovo messaggio all'interno del blocco *B* che a sua volta viene inserito in testa alla lista.

  5 Il thread T non è in grado di leggere il blocco *B* poiché ha già iterato sulla testa della lista.

  Questo scenario si verifica poiché il reader dichiara l'inizio della sua operazione di lettura a seguito del controllo sulla validità del blocco *B*. Di conseguenza, nell'intervallo di tempo che va dal controllo della validità alla dichiarazione dell'inizio dell'operazione di lettura, è possibile che il blocco venga invalidato e deallocato. Per risolvere la problematica in questo

scenario di concorrenza, è possibile dichiarare l'inizio dell'operazione di lettura prima del controllo sulla validità. Questo è il motivo per cui nell'implementazione l'incremento del campo *epoch_ht* avviene prima del controllo sulla validità del blocco. In questo modo, anche se il blocco B dovesse essere invalidato, tale blocco non verrà deallocato e inserito nella *Free Block List* fino a quando (almeno) il reader non ha terminato la scansione della lista. Ovviamente, poiché il thread ha iniziato l'esecuzione della funzione *read_data_block*, il blocco B risultava essere valido in precedenza e, quindi, il suo indice non poteva essere presente all'interno della *Free Block List*.

6.2 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 se è possibile lavorare sui blocchi del dispositivo e per 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 delle due liste (i.e., Tabella Hash e lista ordinata); viceversa, viene restituito all'utente un codice di errore. Se l'inserimento del blocco ha successo, allora viene settato il corrispondente bit nella bitmask in modo da renderlo valido.
Nella funzione *insert_hash_table_valid_and_sorted_list_conc* 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 con la conseguente 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 lista della tabella hash. Se l'inserimento nella tabella hash fallisce, allora l'indice del blocco che si stava utilizzando viene inserito nuovamente nella *Free Block List*, vengono svegliati i thread in attesa degli inserimenti e viene restituito all'utente un errore. Nel caso in cui l'inserimento nella lista della tabella hash ha successo, allora si tenta di inserire il blocco nella lista *Sorted List*. Se l'inserimento fallisce, allora viene eseguito il rollback rimuovendo il blocco dalla tabella hash precedentemente inserito, viene inserito l'indice del blocco nella *Free Block List* e vengono risvegliati i thread che sono in attesa.

6.3 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 se è possibile lavorare sui blocchi del dispositivo e per verificare la correttezza del valore del blocco richiesto dall'utente. 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 si evita la ricerca del blocco nelle due liste. Se il blocco risulta essere valido, allora viene eseguita la funzione *invalidate_block* per procedere con l'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 coomunicarlo agli altri thread tramite l'aggiornamento

della variabile *sync_var*, allora viene eseguita la funzione *remove_block* che ha il compito di rimuovere effettivamente il blocco da entrambe le liste. Bisogna osservare che durante la fase di invalidazione di un blocco è possibile avere in concorrenza solamente dei thread lettori. Nella funzione *remove_block* può accadere che la lista della tabella hash sia vuota o che il blocco richiesto per l'invalidazione non sia presente all'interno della lista. Questo può accadere a causa del fatto che più invalidazioni possono essere richieste in parallelo ma verranno servite solamente una alla volta.

7 Device Driver

7.1 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 lista dei blocchi ordinati partendo dalla testa della lista che si recupera tramite la variabile *head_sorted_list*.

8 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*.

8.1 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.

8.2 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.

8.3 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.