Report Progetto Sistemi Operativi 2018/2019

Progetto:
File System con inode

Componenti:
Alessandro Appolloni 1757950
Romeo Bertoldo 1765456

What

Il File-system è una parte del Sistema Operativo che si occupa di gestire e strutturare le informazioni memorizzate su supporti permanenti. Attraverso il File System il Sistema Operativo fornisce una visione astratta dei file su disco e permette all'utente di accedervi effettuando operazioni ad alto livello.

Questo File System è stato realizzato con allocazione indicizzata dello spazio su disco (Indexed Allocation). Il file system ha dei blocchi chiamati Index Block che hanno il compito di far accedere in maniera diretta ai blocchi del file o della directory in questione. Nel nostro caso è stato scelto il valore di 87 puntatori per il primo blocco index e 126 puntatori per ogni altro blocco index. Inoltre, nel caso di esaurimento dei puntatori, il blocco index punta ad un successivo blocco index con altri 126 puntatori, continuando a cascata.
Il disco è simulato con un file txt (chiamato file_system.txt) diviso in blocchi di grandezza fissa (512 Byte) gestito con il supporto di una bitmap e driver di disco.

How

Nella realizzazione del FIle System abbiamo seguito un ordine ben preciso che ci permettesse di scrivere il codice passo passo e man mano testarlo.
L'ordine è il seguente:

- bitmap.c & bitmap_test.c
- disk_driver.c & disk_driver_test.c
- simplefs.c & simplefs_test.c
- file_system.c

In particolare, il lavoro è stato suddiviso in cinque fasi:

1) Durante la prima parte Alessandro di è occupato dello sviluppo di Bitmap.c e Romeo si è occupato della prima metà di DiskDriver.c;
2) Nella seconda parte Romeo ha terminato DiskDriver.c e Alessandro, dopo aver definito insieme le strutture per il supporto a Inode, ha iniziato a scrivere le prime funzioni di SimpleFs.c;
3) Durante la terza parte abbiamo scritto tutte le funzioni di supporto per inode;
4) Nella quarta parte entrambi ci siamo dedicati allo sviluppo e alla terminazione di SimpleFs.c;
5) Nell'ultima parte entrambi abbiamo sviluppato la shell interattiva in file_system.c e risolto bug;

Al termine di ogni fase abbiamo scritto i rispettivi file di test, abbiamo testato ogni tipo di situazione, risolto bug ed eliminato tutti i memory leak. In ogni caso non è mancata la collaborazione e entrambi ci siamo supportati, soprattutto durante le fasi di test per individuare in maniera precisa ed efficiente gli errori nel codice.

Attraverso bitmap.c possiamo sapere se un blocco è già occupato (bit settato a 1) oppure è libero (bit settato a 0). Per settare i bit, in Bitmap_set, abbiamo utilizzato shift logici e maschere.
Bitmap_get ci ritorna il primo bit, a partire dalla posizione start, con stato "status" (0 o 1).
bitmap_test.c è servito per testare in primo momento le funzioni della Bitmap.

disk_driver.c contiene tutti i driver per poter accedere al disco ed effettuare qualsiasi operazione su di esso, in particolare: possiamo creare un disco da zero o accedere ad uno già esistente, scrivere informazioni su un blocco, leggere informazioni su un blocco e liberare la memoria di un blocco. Il tutto integrato con il supporto della Bitmap.
disk_driver_test.c è servito per testare in primo momento le funzioni dei driver di disco.

In simplefs.c abbiamo realizzato le funzioni più ad alto livello che gestiscono la struttura del File System, creazione e rimozione file/directory e lettura/scrittura su file oltre all'inizializzazione del File System. Spesso ci siamo aiutati con funzioni ausiliarie per cercare di rendere più leggibile il codice.

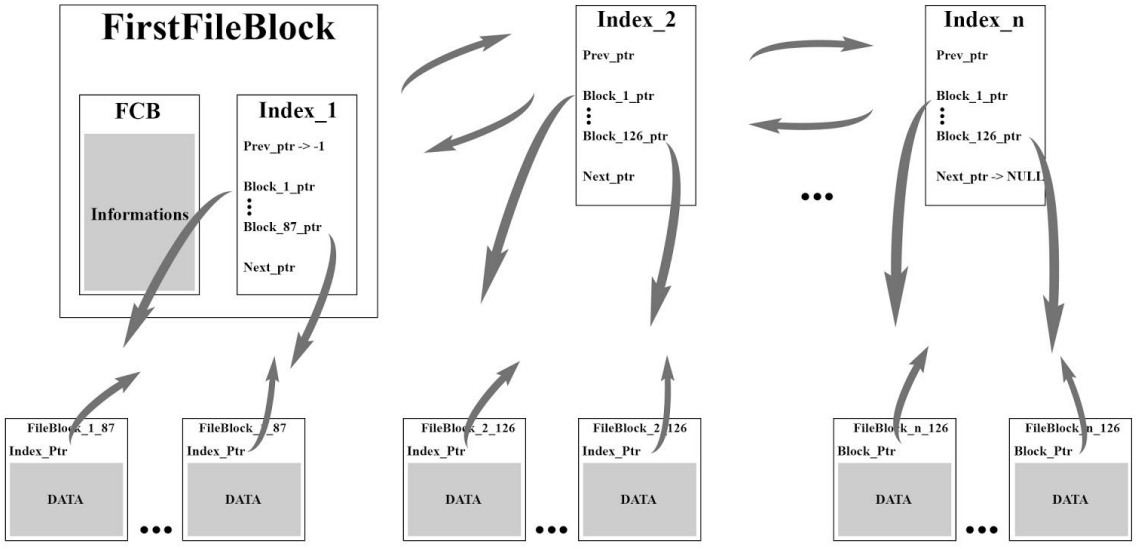
Simplefs_test.c è servito per testare, in maniera statica, tali funzioni prima di poter realizzare il programma interattivo.

Ogni elemento, sia file che directory, memorizza un FileControlBlock, una struttura che immagazzina tutte le informazioni fondamentali del file/directory in questione tra cui nome, dimensione, posizione sul disco.

I file vengono identificati attraverso due strutture: i FirstFileBlock e i FileBlock:

- **FirstFileBlock:** è il primo blocco fisico di un file, contiene il FileControlBlock e il primo blocco index;
- **FileBlock:** blocchi che contengono i dati scritti su file;

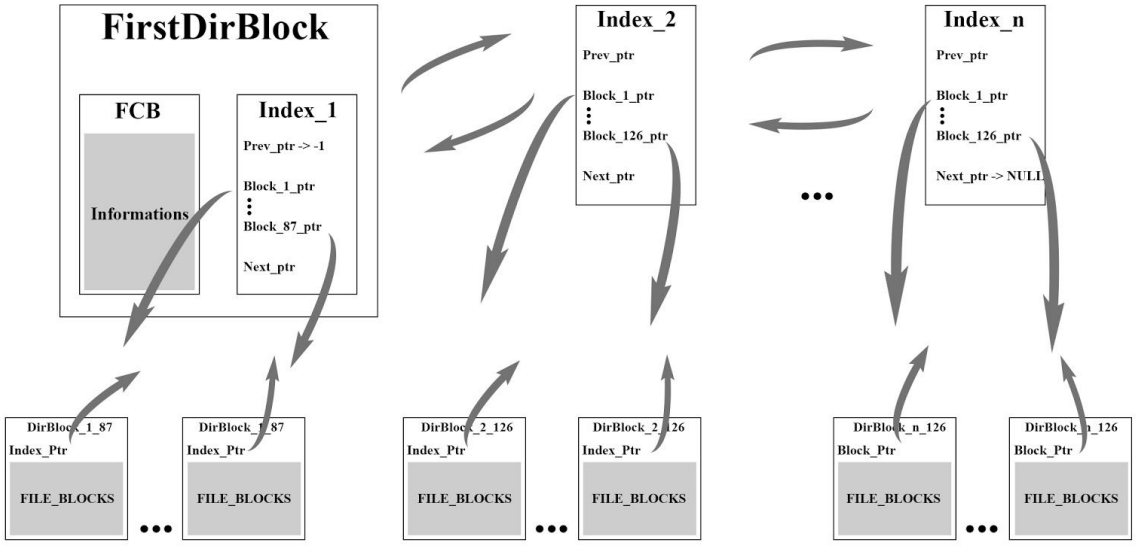
Di seguito un esempio di un file:



Le directory vengono identificate attraverso due strutture: i FirstDirectoryBlock e i DirectoryBlock:

- **FirstDirectoryBlock:** è il primo blocco fisico di una directory, contiene il FileControlBlock e il primo blocco index;
- **DirectoryBlock:** blocchi che contengono i file contenuti nella directory ;

Di seguito un esempio di una directory:



Sia file che directory hanno come supporto altri blocchi di tipo index, in caso di esaurimento del primo.

Per tenere traccia di ogni file o directory aperti all'interno del file system sono state utilizzate le strutture **FileHandle** e **DirectoryHandle**.

FileSystem.c è il programma interattivo, una shell simulata in cui l'utente può effettuare le classiche operazioni da terminale per interagire con il File System. I comandi sono:

- mkdir per creare una directory;
- touch per creare un file;
- write per scrivere su file;
- more per leggere da file;
- rm per rimuovere un file o una directory;
- cd per cambiare directory;
- ls per mostrare i contenuti della cartella;
- info per ottenere informazioni sul disco;
- help per la lista dei comandi;
- quit or CTRL+C per uscire dalla shell;

How-to-Run

Per testare ed utilizzare il nostro file system occorre seguire i seguenti step:

Prima di tutto bisogna digitare da terminale **make** per poter compilare in maniera corretta ogni parte del nostro file system.

Per testare la Bitmap digitare **valgrind ./bitmap_test** ; Dopo aver visualizzato i test effettuati, otterrete il seguente messaggio da parte di valgrind:

```
==4677==
==4677== HEAP SUMMARY:
==4677==     in use at exit: 0 bytes in 0 blocks
==4677==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==4677==
==4677== All heap blocks were freed -- no leaks are possible
==4677==
==4677== For counts of detected and suppressed errors, rerun with: -v
==4677== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Per testare i Driver del Disco digitare **valgrind ./disk_driver_test** ; Dopo aver visualizzato i test effettuati, otterrete il seguente messaggio da parte di valgrind:

```
==4836==
==4836== HEAP SUMMARY:
==4836==     in use at exit: 0 bytes in 0 blocks
==4836==   total heap usage: 11 allocs, 11 frees, 6,168 bytes allocated
==4836==
==4836== All heap blocks were freed -- no leaks are possible
==4836==
==4836== For counts of detected and suppressed errors, rerun with: -v
==4836== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Per testare le funzioni di supporto a inode digitare **valgrind ./index_test** ; Dopo aver visualizzato i test effettuati, otterrete il seguente messaggio da parte di valgrind:

```
==4880==
==4880== HEAP SUMMARY:
==4880==     in use at exit: 0 bytes in 0 blocks
==4880==   total heap usage: 97 allocs, 97 frees, 22,668 bytes allocated
==4880==
==4880== All heap blocks were freed -- no leaks are possible
==4880==
==4880== For counts of detected and suppressed errors, rerun with: -v
==4880== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Per testare le funzioni di appoggio al File System, in SimpleFS, digitare **valgrind ./simplefs_test** ; Dopo aver visualizzato i test effettuati, otterrete il seguente messaggio da parte di valgrind:

```
==4921==
==4921== HEAP SUMMARY:
==4921==     in use at exit: 0 bytes in 0 blocks
==4921==   total heap usage: 437 allocs, 437 frees, 119,523 bytes allocated
==4921==
==4921== All heap blocks were freed -- no leaks are possible
==4921==
==4921== For counts of detected and suppressed errors, rerun with: -v
==4921== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Per utilizzare la nostra shell digitare **valgrind ./file_system** ; Dopo aver effettuato le operazioni necessarie, valgrind confermerà l'assenza di memory leak.