SAPIENZA
UNIVERSITÀ DI ROMA

File System con Inode

Facoltà di Ingegneria dell'informazione, informatica e statistica
Dipartimento di Ingegneria informatica, automatica e gestionale
Corso di Laurea in Ingegneria Informatica e Automatica

Candidato

Alessandro Garbetta
Matricola 1785139

Relatore

Prof. Giorgio Grisetti

Anno Accademico 2020/2021

File System con Inode
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: garbetta.alessandro@gmail.com

A Marisa, mia nonna,
ed Alessandra, mia zia ...

Indice

Capitolo 1

Introduzione

A partire dal XX secolo l'informatica ha avuto una crescita esponenziale; lo sviluppo dei computer, basato sul modello della macchina di Turing e l'architettura di Von Neumann, ha portato numerosi vantaggi nella vita di tutti i giorni, diventando un elemento imprescindibile delle nostre vite. Alla base del funzionamento dei computer moderni c'è il sistema operativo, un particolare software che, con tutte le sue componenti, gestisce e organizza i vari compiti degli elaboratori di calcolo. Un ruolo centrale nei sistemi operativi è quello del file system, che ha il compito di organizzare e rendere disponibili i dati e tutte le informazioni che vengono salvate. È proprio su questa componente che si sofferma il mio progetto, il quale ha l'ambizione di simulare il funzionamento di un file system in una forma più semplice e basilare, ma con gli stessi obiettivi e funzionalità.
Nel Capitolo 2, per rendere più comprensibile lo sviluppo del progetto, si sono trattati in maniera generale il sistema operativo e le sue componenti, soffermandosi, in particolar modo, sul file system. Non si è potuto fare a meno di introdurre un'altra componente fondamentale per il suo funzionamento: la memoria, con un focus su quella secondaria. Si è fornita come esempio la struttura di un disco rigido (HDD - Hard Disk Drive), nonostante questa tecnologia stia perdendo mercato a favore di unità di memoria allo stato solido (SSD - Solid State Drive), il suo funzionamento resta comunque attuale ed esplicativo dato il largo uso su molte macchine e da milioni di utenti come "contenitore" di dati. Il capitolo prosegue analizzando tutte le componenti del file system, dagli elementi cardine, come i file e le cartelle, ai modi in cui questi sono gestiti.
Nel Capitolo 3, invece, l'attenzione si è spostata sulla fase di progettazione e di scrittura del codice, scritto in linguaggio C, delle strutture e delle varie funzioni, facendo un'analisi generale delle scelte progettuali.
Nel Capitolo 4, si sono mostrati esempi pratici delle varie operazioni che si possono svolgere, spiegando gli output e dando un'immagine concreta al programma creato.

Capitolo 2

File System

2.1 Introduzione al File System

Il sistema operativo (SO) è il programma principale in esecuzione su un elaboratore che ha il compito di essere l'intermediario tra l'utente e l'hardware sottostante. È costituito da un insieme di programmi che gestiscono le risorse hardware e software della macchina, fornendo, in questo modo, un ambiente dove l'utente può eseguire i programmi applicativi in maniera efficiente.
Per svolgere i suoi compiti, il sistema operativo è composto da vari sottosistemi, tra cui: il kernel, lo scheduler, il file system, i gestori della memoria e delle periferiche, ed altri ancora.

Il file system (FS) è una delle componenti principali dei moderni sistemi operativi, risiede nella memoria secondaria e si occupa della gestione dei file e delle directory (o cartelle). Di questi ultimi definisce gli aspetti principali, dalla struttura e l'organizzazione, alle modalità di lettura e scrittura. Inoltre, permettendo l'interfaccia tra l'utente e la memoria, fornisce un accesso efficiente al disco memorizzando i dati in modo da poter essere recuperati facilmente.
All'interno di un dispositivo (sistema) possono coesistere diversi file system, ognuno con delle peculiarità di gestione in base al tipo di dati che salvano. La presenza di diversi file system è permessa grazie all'uso di un Virtual File System (VFS), un software che, fornendo un'interfaccia, nasconde le singole implementazioni delle operazioni e rende disponibili le chiamate di sistema di un file system.
Le componenti principali sono:

- File: elemento principale di memorizzazione di dati di vario genere.

- Directory: collezione di file e/o altre directory.

- Collegamenti (links): puntatori di un file presente nel sistema; possono essere fisici o logici.

- Punto di montaggio (mount point): la cartella che contiene il file system del dispositivo.

2.2 La Memoria

La memoria è la componente di un elaboratore che ha il compito di mantenere in maniera persistente i dati e le istruzioni dei programmi. Le diverse implementazioni fisiche danno vita ai vari supporti di memorizzazione esistenti. La memorizzazione di informazioni e il loro recupero sono operazioni di fondamentale importanza all'interno dei processi di elaborazione dei dati. In maniera astratta la memoria può essere vista come una sequenza finita di celle, ognuna contenente una sequenza finita di bit, in generale otto, detti byte. Ogni posizione è individuata da un indirizzo di memoria univoco, generalmente espresso con un numero esadecimale positivo.
Le principali operazioni che vengono eseguite in memoria sono: l'inizializzazione, la scrittura e la lettura.
In base alla sua natura, la memoria si distigue in: memoria centrale o primaria e memoria di massa o secondaria. La memoria centrale è collegata alla scheda madre e contiene i dati e le istruzioni prelevati inizialmente dalla memoria di massa e, successivamente, dal microprocessore il quale li elabora. Quando si parla di memorie centrali ci si riferisce, nella maggior parte dei casi, a memorie RAM e memorie cache, le quali possono essere incorporate nella scheda CPU o direttamente nel chip.
La memoria di massa viene chiamata anche memoria secondaria poichè si aggiunge alla primaria per aumentare le capacità di memorizzazione dell'elaboratore. Questo tipo di dispositivi non è collegato direttamente al processore ed è non volatile, perciò, al contrario della primaria, non necessita di elettricità per mantenere i dati che, dunque, persistono allo spegnimento della macchina.
Un'altra divisione può essere fatta in base al tipo di accesso alla memoria: quello sequenziale, dove le memorie possono essere lette o scritte solamente all'indirizzo immediatamente successivo all'indirizzo a cui è avvenuto un accesso precedente; quello diretto, dove possono essere lette e scritte a qualunque indirizzo, con tempo di accesso variante dall'indirizzo di memoria a cui è avvenuto l'accesso precedente e quello casuale, dove possono essere lette e scritte a qualunque indirizzo con stesso tempo di accesso.
Un disco di memoria, dal punto di vista logico, per essere utilizzato viene suddiviso idealmente in blocchi (porzioni più o meno ampie la cui dimensione è gestita dal sistema operativo, che può essere fissa o variabile). Nei blocchi si scrivono, e successi-

vamente si leggono, i dati e le informazioni utili allo svolgimento di un compito. Ogni volta che si deve fare un accesso in memoria, un'allocazione o una de-allocazione, questo si farà per blocchi.
Un problema che si può riscontrare quando viene utilizzata la memoria è quello della frammentazione, il quale deve essere gestito per far si che si verifichi il meno possibile.
La frammentazione avviene quando esiste in memoria spazio libero ma non è disponibile per soddisfare le richieste di un programma (che sia una richiesta del SO, o dell'utente). Esistono due tipi di frammentazione: interna ed esterna. La prima si verifica quando è presente spazio inutilizzato all'interno di un blocco precedentemente allocato. Quando si scrive su disco si fa una richiesta di allocazione e si scrive su un blocco di memoria: se la grandezza dell'informazione che si sta scrivendo è più piccola del blocco allocabile più piccolo, lo spazio di memoria non utilizzato non sarà più utilizzabile per allocazioni future, e sarà "perso". La seconda invece, quella esterna, si manifesta quando non si riesce a soddisfare una richiesta di allocazione nonostante ci sia spazio libero disponibile, il quale però non è contiguo in memoria. La frammentazione esterna si può verificare solo nel caso in cui venga richiesta l'allocazione di blocchi di dimensioni diverse: se si allocano solamente blocchi della stessa dimensione, una volta liberati lasciano uno spazio contiguo sempre utilizzabile per soddisfare future richieste di allocazione. Tra le varie categorie di memoria secondaria ci sono: i dischi magnetici, i dischi ottici, i nastri magnetici, le unità di memoria a stato solido, ecc.

2.2.1 Il disco rigido

Il disco rigido (o hard disk) è uno tra i dispositivi di archiviazione più usato per la memorizzazione a lungo termine dei dati nei personal computer e nei dispositivi elettronici.
Il disco rigido è costituito da uno o più dischi, in rapida rotazione, di materiali come l'alluminio o il vetro e rivestisti di materiale ferromagnetico, da due testine per ogni disco (una per ogni lato del piatto), le quali, spostandosi sul disco leggono o scrivono i dati.
La scrittura dei dati sulla superficie del supporto ferromagnetico avviene attraverso il trasferimento di un verso alla magnetizzazione di un certo numero di domini di Weiss (una piccola area nella struttura cristallina di un materiale ferromagnetico, i cui grani hanno un'orientazione magnetica). Ad un determinato stato (verso) di magnetizzazione è associato il bit di informazione 1 o 0.
I dati, a livello fisico, sono generalmente memorizzati su disco seguendo uno schema ben definito di allocazione fisica, in base al quale si può raggiungere la zona dove

leggere/scrivere i dati.
Un hard disk ha 3 caratteristiche fondamentali che sono di particolare interesse: la capacità, il tempo di accesso e la velocità di trasferimento.
La capacità è espressa in multipli del byte, dal megabyte al terabyte; il tempo di accesso è la variabile discriminante le prestazioni del disco. Esso è il tempo medio di cui necessita il disco per recuperare un'informazione e dipende dal movimento della testina e dalla velocità di rotazione dei piatti (indicata con RPM, Revolution per Minute). La velocità di trasferimento è la quantità di dati che l'hard disk è in grado di leggere e/o di scrivere in un determinato lasso di tempo. Per migliorare questo fattore si possono o velocizzare i piatti, o aumentare la densità di memorizzazione.

2.3 Componenti Principali ed Operazioni

2.3.1 File

I file sono dei "contenitori" digitali di informazioni e/o dati salvati in memoria permanente con spazio di indirizzamento contiguo.
Ogni file può essere letto o scritto, ha delle proprietà, e subisce varie operazioni.
Possono essere di vari formati definiti al momento della loro creazione e in alcuni sistemi sono specificati dall'estensione. Il formato di un file è l'insieme di regole che specificano come decodificare le informazioni e le istruzioni per il loro uso e/o per la loro rappresentazione. Per interpretare correttamente il contenuto del file, sono necessari programmi in grado di decifrare quel particolare formato.
I file possono essere dati (numerici, caratteri, binari) o programmi (eseguibili), e sono composti da vari attributi: il nome (che lo identifica all'interno della directory), la directory che lo contiene, il tipo, la posizione (riferimento allo spazio fisico nel quale si trova), la dimensione, i permessi, il tempo-data-identificazione dell'utente, ed altre informazioni che possono essere utili al sistema o all'utente.
I file hanno un proprietario (owner) e un gruppo (group).
Gli utenti all'interno di un sistema operativo possono essere raggruppati in 3 classi: utente, gruppo e altri (chi non appartiene ai precedenti). Per ogni classe, inoltre, si possono definire 3 permessi: in lettura, in scrittura e in esecuzione.
I permessi possono essere modificati dall'utente e dall'amministratore, ma solamente l'amministratore potrà modificare il proprietario.

2.3.2 Directory

Una directory (o cartella), in Unix, è un file di voci di file ovvero una collezione di nodi contenenti informazioni sui file. Le directory possono diventare un punto di ancoraggio per i nuovi file system che si trovano all'interno del principale.

Il file system può essere visto come una gerarchia di file e cartelle identificabile con l'albero delle directory. Dalla Figura 2.1 si evince come la directory principale sia la

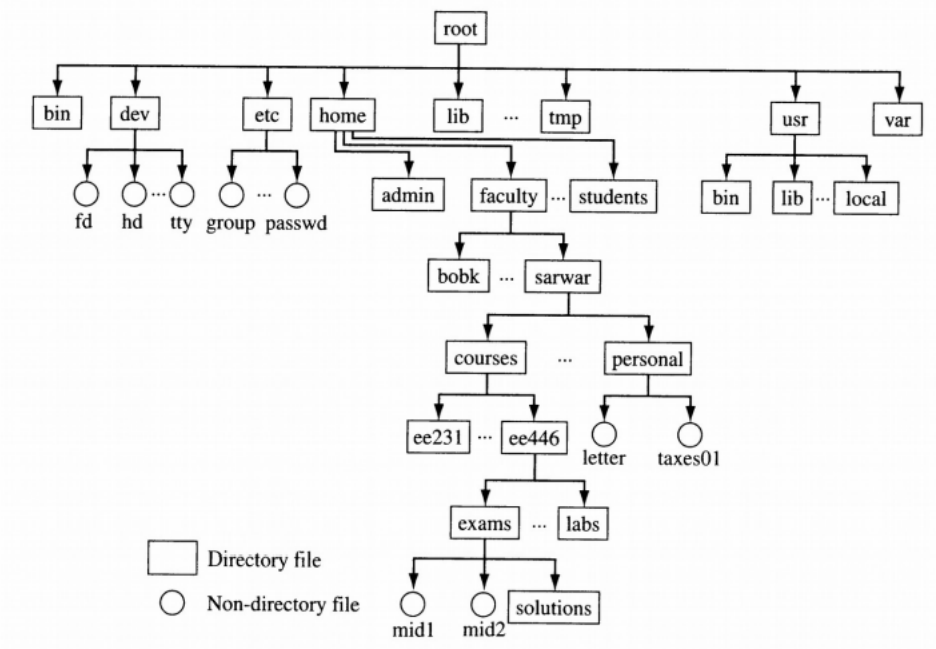

Figura 2.1. Esempio albero di directory e file

root, che indica la radice dell'albero dalla quale sono raggiungibili tutti gli altri file e cartelle. Nell'albero le directory costituiscono i nodi mentre i file le foglie. Ogni elemento è indentificato dal proprio percorso che è univoco e rappresenta tutte le cartelle, che dalla root, occorre visitare per arrivare al nodo che si sta individuando. Per esempio il percorso della directory *students* sarà: *"root/home/students"*. Ogni directory, oltre a salvare le informazioni sui propri file, manterrà informazioni come la propria posizione, la posizione del genitore, la sua grandezza, ecc.

2.3.3 Collegamenti

I collegamenti (link) possono essere hard link (link fisici) o soft link (link simbolici o logici). La creazione di un file è seguita sempre da quella di un link fisico, il quale permette ai programmi di riferirsi ai dati tramite un nome nel file system. Un file senza un collegamento fisico non è accessibile e, di conseguenza, rende di nuovo disponibile lo spazio occupato. In sistemi Unix, Unix-like, i link fisici sono come dei contatori che mantengono il numero di riferimenti allo specifico file e ad ogni eliminazione vengono decrementati; eliminandoli tutti viene rimosso il file fisico dalla memoria.
I link logici, invece, sono puntatori simbolici: sono file che contengono il percorso, o pathname, di un file o di una cartella reale; eliminandone uno non si elimina il

file in memoria ma solo il puntatore, se invece si elimina il file fisico il link rimane pendente.

2.3.4 Punto di Montaggio

Il mount point (punto di montaggio) è il punto nel quale viene posizionato il file system. Attraverso l'operazione di mount si mappa il volume con il proprio file system e si fa in modo che sia possibile vedere qualsiasi periferica o disco che viene collegato al sistema come una semplice cartella/file.
Può essere descritto come la directory principale del file system, attraverso la quale si accede ai dati memorizzati nei dischi rigidi. Nella Figura 2.1, per esempio, *dev* è la directory che individua sotto forma di file le periferiche hardware. Quando una cartella diventa un mount point il contenuto allocato precedentemente non è più accessibile. Solo attraverso l'operazione di unmount, ossia quando viene "smontato" il file system, i file torneranno ad essere disponibili.
L'operazione di mount, per esempio, può essere eseguita automaticamente da un demone in attesa su un USB per aggiungere il dispositivo esterno al file system preesistente sull'elaboratore.

2.3.5 Operazioni

- Operazioni File
  - creazione
  - scrittura
  - lettura
  - riposizionamento (seek) all'interno del file
  - rimozione
  - troncamento
  - apertura
  - chiusura
- Operazioni Directory
  - aggiungere un file
  - cancellare un file
  - visualizzare (list) il contenuto della directory
  - rinominare un file
  - ricercare un file
  - attraversare il file system

2.4 Metodi Allocazione File

Un file, dal punto di visto fisico, è un insieme di blocchi di memoria ed è fondamentale decidere come tenerne traccia e come scegliere i blocchi da assegnare ad un determinato file. Le strategie principali sono 3 e vengono scelte in base alla natura del dispositivo sul quale è montato il file system e in base alla tipologia di dati che verranno salvati e letti. Si può dimostrare che tra le varie strategie la scelta migliore sia quella di creare anche degli ibridi per rendere il sistema il più efficiente possibile. Si distinguono 3 metodi di allocazione:

- allocazione contigua
- allocazione a lista concatenata (linked list)
- allocazione a lista indicizzata (inode)

Allocazione contigua
Ogni file viene memorizzato in sequenze di blocchi di memoria consecutivi ed è definito dal blocco di inizio e dal numero dei blocchi da cui è costituito. Questa tecnica comporta dei vantaggi e degli svantaggi. In primo luogo, la vicinanza spaziale dei blocchi su disco fa lavorare di meno i piatti, la testina si dovrà spostare per leggere il primo blocco e gli spostamenti successivi saranno minimi, rendendo trascurabile il tempo di ricerca dei blocchi. Inoltre, l'allocazione contigua, consente sia l'accesso sequenziale che diretto ai blocchi del file.
Un problema che si verifica è la difficoltà di trovare spazio libero contiguo per l'allocazione dell'intero file. Si possono usare gli algoritmi di first-fit (si utilizza il primo spazio abbastanza grande per contenere il file), o di best-fit (si utilizza lo spazio che approssima meglio la dimensione, per eccesso, del file) per scegliere lo spazio da allocare. Un'altra problematica che si può riscontrare è la frammentazione esterna: ad ogni scrittura di un nuovo file il disco si riempie e restano zone libere contigue sempre minori fino a diventare inutilizzabili. Nel caso in cui lo spazio di memoria libero risulti eccessivamente frammentato, per ridurlo, divengono necessarie azioni di compattazione. La compattazione consiste nello spostare i file su un altro disco, creare una zona contigua di spazi liberi, per poi riportare i file sul disco originale e allocandoli in modo contiguo. Questa tecnica rende lo spazio libero rimasto sul disco ancora utilizzabile. L'operazione di compattazione richiede del tempo e può essere fatta sia off-line che on-line: la prima rende il disco non utilizzabile fino alla fine dell'operazione; la seconda rallenta le prestazioni della macchina. Un altro grande svantaggio è la necessità di fare una stima della dimensione del file ancor prima che questo venga creato e scritto. Occorre stimare quanto

spazio riservare per la sua allocazione. Quando un file esaurisce il suo spazio disponibile bisogna spostarlo in una zona contigua di memoria libera di maggiori dimensioni.

Allocazione a lista concatenata

Ogni file è composto da una lista concatenata di blocchi sparsi all'interno del disco ed è identificato dal puntatore al primo blocco e all'ultimo. Ogni blocco contiene il puntatore al blocco successivo. Quando si crea un file si aggiunge un elemento alla directory con il puntatore al primo blocco settato a null e dimensione pari a 0. Quando si scrive il file si cerca un blocco libero e si effettua la scrittura; successivamente viene concatenato all'ultimo blocco del file (se presente) e si aggiornano i puntatori della directory al primo e all'ultimo blocco. Per la lettura, invece, si scorre la sequenza dei blocchi seguendo la concatenazione.

Anche questa tecnica ha dei "pro" e dei "contro". Tra i "pro", per esempio, c'è l'assenza dei problemi dell'allocazione contigua tra cui quelli della frammentazione e della compattazione, e la stima della dimensione del file al momento della creazione non è più necessaria. Un "contro" è che solo l'accesso sequenziale dei blocchi sarà efficiente; a causa dei puntatori non sarà possibile sapere dove è locato uno specifico blocco senza prima accedere a tutti i precedenti. Questo richiede più lavoro da parte del disco, il quale dovrà leggere i blocchi sparsi, eventualmente, su tutta la memoria. Inoltre, i puntatori occupano spazio nella struttura consumando lo spazio libero e la loro gestione potrebbe creare dei problemi, come per esempio puntare ad un blocco vuoto o a quello di un altro file. Per risolvere questo problema si può utilizzare una lista doppiamente concatenata.

Esiste una variante della linked list, l'allocazione a lista concatenata FAT (File Allocation Table, o Tabella di Allocazione File), la quale presenta nella prima parte del disco una tabella. Quest'ultima contiene tanti elementi quanti sono i blocchi di memoria, i quali vengono ordinati in base al numero del blocco (in senso crescente). La directory contiene per ogni file il puntatore al primo elemento nella FAT, ed ogni elemento successivo contiene il numero del prossimo blocco del file. L'ultimo elemento presenta un valore che indica la fine della lista, quindi del file (o un valore NULL o -1). Nella FAT, gli elementi che non sono stati allocati, e che quindi sono liberi, contengono il numero 0. Questa modalità di allocazione per essere più efficiente della precedente, deve avere la tabella dei file caricata in RAM, o in una memoria cache. La sua assenza porterebbe la testina del disco a spostarsi ogni volta tra l'inizio del disco e la locazione del blocco successivo. In ogni caso, l'accesso diretto ai blocchi del file migliora, poichè, la testina scorrerà al massimo tutta la FAT per sapere dove è allocato un determinato blocco e non tutto il disco come sarebbe accaduto precedentemente.

Allocazione indicizzata

Con questo metodo di allocazione i blocchi della memoria si dividono in blocchi data e blocchi indice. Le componenti principali che rappresentano i file sono gli inode, delle strutture che mantengono varie informazioni sui file e sul loro indirizzamento all'interno del disco. Ogni file possiede almeno un blocco indice, il quale contiene un array con le posizioni dei blocchi sul disco che rappresentano i vari blocchi data del file. L'i-esimo elemento dell'array punta all'i-esimo blocco del file. La directory memorizza per ogni file un puntatore al suo blocco indice. Utilizzando questo metodo si riesce a risolvere il problema dell'accesso diretto che si presenta con l'allocazione a lista e si evita anche la frammentazione esterna. Quando si crea un file per prima cosa si crea un blocco indice con tutti gli elementi settati a null (o -1, per indicare che non stanno puntando a nessun blocco) e si aggiunge un elemento alla directory con puntatore al blocco indice. Per scrivere sul file si cerca il primo blocco libero, si effettua la scrittura dei dati e si aggiorna il blocco indice. Infine, per leggere il file non serve altro che scorrere in sequenza i blocchi data seguendo l'array del blocco indice.
Un problema può verificarsi quando, scrivendo file di piccole dimensioni, non viene utilizzato l'intero blocco indice, lasciandolo per lo più vuoto e portando così frammentazione interna. Per ovviare a questa problematica si devono usare blocchi più piccoli in modo da ridurne le celle non utilizzate. Questo può comportare difficoltà nella gestione dei file di grandi dimensioni. Ci sono due possibili soluzioni: lo schema concatenato e quello a più livelli. Con il primo si usano blocchi più piccoli e, nel caso in cui il file sia troppo grande da essere indicizzato con un solo blocco indice, l'indice dei blocchi sarà composto da più blocchi indici concatenati. Con il secondo, invece, si usano sempre blocchi più piccoli, ma si ha un indice di primo livello che non conterrà la posizione del blocco dati del file. Quest'ultimo conterrà la posizione di un altro blocco indice di secondo livello, il quale avrà l'array che punta ai blocchi data del file. Questo meccanismo può estendersi fino a 3/4 livelli. La Figura 2.2 mostra un esempio della struttura rappresentante i file nei sistemi Unix , in cui viene usato uno schema combinato, nel quale ogni First File Block contiene 15 elementi per indicizzare un file. I primi 12 sono puntatori diretti ai blocchi data del file, il 13° punta ad un blocco indice di primo livello, il 14° ad uno di secondo ed infine il 15° ad uno di terzo livello. In questo modo si riesce a salvare file con una dimensione massima di $(12 + 256 + 256^2 + 256^3)$*grandezza del blocco (ipotizzando che ogni blocco indice abbia al suo interno 256 elementi).

Come è stato già esposto in precedenza, la scelta del metodo di allocazione non è

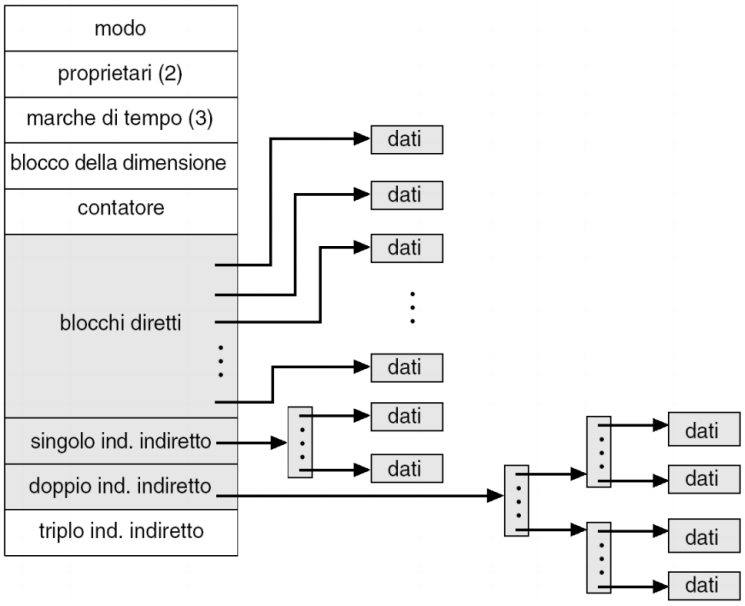

Figura 2.2. Esempio di un Inode

univoca e deve considerare principalmente due fattori: l'efficienza di memorizzazione e il tempo di accesso ai dati. La scelta è influenzata dalla modalità di accesso al file. La prima tecnica, quella dell'allocazione contigua, richiede un solo accesso al disco, mentre, la seconda, ovvero quella concatenata, è efficiente per l'accesso sequenziale ma non per quello diretto. Le prestazioni dell'allocazione indicizzata, diversamente, dipendono molto dalla dimensione del file, che va a ricadere sulla profondità dell'indice. In base ai vantaggi e alle problematiche elencate precedentemente si conclude la scelta del metodo di allocazione prendendo in considerazione le metriche che si vogliono massimizzare con il sistema in uso. Esistono anche dei sistemi, per esempio quelli ibridi, nei quali si decide di utilizzare l'allocazione contigua per i file di piccole dimensioni e quella indicizzata per i file più grandi.

2.5 Implementazione Directory

Le directory, com'è stato già ampliamente specificato, non sono altro che collezioni di file ed altre directory, chiamate colloquialmente sotto-cartelle.
Esistono due possibili modalità di implementazione: la lista concatenata (linked-list) o la tabella di hash (hash table). La prima modalità è una lista concatenata dei nomi dei file con i puntatori ai loro blocchi data: è facilmente programmabile ed ha tempo di consultazione lineare. Volendo fare delle ottimizzazioni si può gestire la lista come un B-Tree e ordinare i file alfabeticamente. La seconda opzione, invece, salva i file attraverso una struttura dati hash, che da un lato permette una maggiore efficienza

della ricerca, ma dall'altro può generare delle collisioni, che si riscontrano nella maggior parte delle tabelle di hash. La collissione tra due nomi si verifica quando due file diversi ricadono nella stessa posizione della tabella. Questa problematica è facilmente risolvibile anche se comporta una maggior gestione della tabella e dei cali a livello prestazionale.

2.6 Gestione spazio libero

Il file system si occupa della gestione efficiente della memoria, per rendere la creazione e l'eliminazione di un file il più veloce possibile. Questo porta ad avere sempre traccia dei blocchi di memoria non allocati. Ci sono diversi modi per renderlo possibile: attraverso un vettore di bit (bitmap), una lista concatenata e attraverso la tecnica del raggruppamento o del conteggio.

Bitmap
La bitmap è un vettore che viene utilizzato per rappresentare lo stato dei blocchi di memoria del disco sul quale è montato il file system. La lunghezza dell'array deve essere pari al numero dei blocchi di memoria e lo stato di ogni blocco è rappresentato da un bit: quando è 1 il blocco sarà libero, quando è 0 sarà occupato.
La bitmap è di semplice implementazione ed efficiente nella ricerca del primo o di n blocchi liberi. Inoltre, a suo svantaggio c'è la richiesta di uno spazio di memoria aggiuntivo per salvare il vettore. Questo può risultare problematico con dischi di grandi dimensioni poichè la tecnica risulta vantaggiosa solo se la mappa di bit si trova in memoria centrale.

Lista concatenata
Questa tecnica gestisce i blocchi liberi tramite una lista concatenata: ci sarà un puntatore al primo blocco libero, che a sua volta sarà concatenato al successivo.
In questo modo si avrà sempre a disposizione il primo blocco libero. Nel momento in cui quest'ultimo verrà richiesto per essere allocato, si aggiornerà il puntatore al blocco successivo. Se verrà chiesto di scorrere l'intera lista l'operazione sarà poco efficiente, poiché i molti accessi in memoria comportano una diminuzione delle prestazioni.

Raggruppamento
In memoria è contenuto l'indirizzo del primo blocco libero, nel quale sono presenti gli indirizzi degli altri blocchi liberi. L'ultimo conterrà l'indirizzo del successivo blocco di indirizzi di blocchi liberi.

Concatenamento

Quest'ultima tecnica sfrutta il fatto che i blocchi liberi sono spesso contigui: in memoria viene salvato un array dove ogni elemento contiene l'indirizzo del primo blocco libero e il numero di blocchi liberi contigui.

Capitolo 3

Progetto di File System con Inode

3.1 Analisi e Progettazione

La realizzazione di questo progetto ha avuto inizio con una prima analisi di tutte le varie componenti.
L'obiettivo è stato quello di creare un file system ispirato a quelli esistenti, in grado di eseguire le comuni operazioni da riga di comando in modo il più efficiente possibile, senza però copiarne la reale implementazione.
Oltre a sviluppare i codici delle varie funzioni, è stata creata una shell client-friendly per rendere il funzionamento intuitivo e funzionale.Il primo step è stato quello di decidere quali tecniche di allocazione e gestione dello spazio libero utilizzare.
Per la prima si è optato per l'allocazione indicizzata con inode, con una indicizzazione fino a 3 livelli; per la seconda la bitmap.
Il programma è stato scritto per strati, con un approccio bottom-up, iniziando con la scrittura della bitmap, l'implementazione del disco, per permettere la lettura e la scrittura dei blocchi di memoria, ed infine, le funzioni del file system, nelle quali si sono implementate le operazioni di alto livello.
La shell è stata sviluppata per ultima, implementando una grafica interattiva per rendere fruibile l'intero progetto.

3.2 Strutture

Per l'implementazione delle varie componenti sono state realizzate delle struct, strutture, che costituiscono l'intero scheletro dell'implementazione, necessarie per lo svolgimento delle varie funzioni da svolgere.

3.2.1 Bitmap

La bitmap viene vista come un insieme di celle contigue che rappresenta la memoria del dispositivo. Può essere rappresentata tramite un array, dove ogni elemento è una parola di n bit. Ogni bit (0, 1) indica lo stato di memoria di un singolo blocco. La Figura 3.1 riporta come esempio la stuttura di una bitmap, messa in parallello al disco. Viene usato l'1 per indicare il blocco occupato, e lo 0 per quello libero. Sulla bitmap si svolgono le operazioni di lettura e scrittura del bit, che sono permesse solo tramite una corretta gestione dell'indice dell'array che la rappresenta. È stata necessaria la creazione di due strutture. La prima è la BitMap, che contiene i dati del vettore: il numero di bit e il puntatore all'array. Si è scelto di usare parole di 8 bit, quindi ogni cella è grande un byte, e viene rappresentata con un char. Con ogni cella dell'array vengono rappresentati 8 blocchi di memoria contigui. Questo è un vantaggio, ottimizzando la memorizzazione delle informazioni, ma implica un lavoro di maschere e operazioni bit a bit per modificare un singolo bit della tabella. La seconda struttura, la BitMapEntryKey, aiuta proprio nel suddetto compito e contiene anch'essa due informazioni: l'indice del vettore della bitmap, e l'offset del bit all'interno del byte; è una entry della mappa.
Infine, sono state implementate le varie funzioni: la conversione dall'indice del blocco a quello dell'array e viceversa, la get e la set di un bit e la print della bitmap usata durante i test del programma.

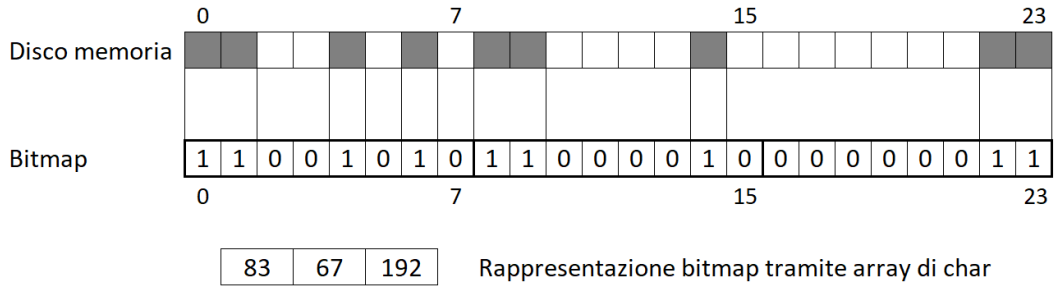

Figura 3.1. Esempio Bitmap di una memoria costituita da 24 blocchi

3.2.2 Disk Driver

La seconda parte dello sviluppo si è basata sull'implementazione del disco di memoria. Non si è lavorato fisicamente con un hard disk, ma se ne è simulato il funzionamento attraverso l'uso di un file (che può essere un file binario, o semplicemente un file di testo) sul quale vengono svolte le operazioni di lettura e scrittura. Il file è stato idealmente diviso in blocchi fissi da 512 byte, che vanno a contenere le

informazioni di file e directory.
Quando si lavora con un file si può mappare in memoria principale, per renderlo più velocemente accessibile e fare in modo che il memory mananger ne gestisca la cache e la scrittura (i file mappati in memoria sono disponibili nei sistemi posix, con la chiamata a funzione mmap, in questo modo il file è visto come un array e viene salvato nello spazio di indirizzamento del processo che lo apre). Le strutture necessarie sono state la DiskHeader e la DiskDriver.
La prima contiene le informazioni base per l'uso del disco, tra cui: il numero di entries della bitmap, il numero di blocchi del disco, il numero dei blocchi liberi e il primo blocco libero, da utilizzare nella allocazione successiva. La seconda, invece, contiene l'header, l'array della bitmap e il filedescriptor del file.
Le funzioni implementate sono state quelle di inizializzazione del disco, nel quale si crea il file e si inizializzano le strutture, di lettura e scrittura di un blocco, la quale si occupa anche dell'aggiornamento della bitmap con un controllo dello stato di memoria del blocco sul quale si sta scrivendo (in modo da non permettere la sovra scrizione), ed infine, la liberazione di un blocco e l'individuazione del primo blocco libero.
Inoltre, durante la scrittura del codice è stata necessaria la creazione di un'altra funzione, che "copia" la scrittura del blocco, che è stata utilizzata per l'aggiornamento di un blocco già scritto, non comportando un'ulteriore modifica della bitmap.
Dato che la bitmap e il DiskHeader sono mmappati in memoria, è stata scritta anche l'operazione di flush, per scrivere i dati rimasti nel buffer ed essere sicuri che la scrittura sia avvenuta nel file.

3.2.3 Simple File System

Il Simple File System è il vero fulcro dell'applicazione. In questa sezione sono state sviluppate le strutture e le funzioni che costituiscono il file system e la gestione dei suoi elementi. Sono state create le strutture dei file e delle cartelle sviluppando un sistema con allocazione indicizzata, emulando la struttura degli inode nei sistemi Unix-like. In questi sistemi ogni elemento è visto come un file, come anche le cartelle, con la differenza che non contengono dati ma "indici" di altri file e cartelle. Si sono volute creare due strutture molto simili, ma la differenza si trova negli oggetti puntati dai blocchi indici: quelli dei file puntano a blocchi data, mentre quelli delle directory punteranno ai primi blocchi dei file o delle cartelle contenute.
Le strutture sono state divise idealmente in due categorie: le strutture scritte sul disco e le strutture per la gestione dei blocchi. Le prime sono: il BlockHeader, il FileControlBlock, il FirstDirectoryBlock, il FirstFileBlock, il FileBlock e l'IndexBlock.
Il BlockHeader è la struttura presente all'inizio di ogni blocco allocato, contiene

le informazioni base di ogni singolo blocco come il numero che occupa all'interno del disco, il numero sequenziale del blocco del file e un flag per indicare il tipo del blocco, ovvero data o indice.
Il FileControlBlock (FCB) è la struttura che contiene tutte le informazioni di un file: il nome, la grandezza in bytes ed in blocchi, la sua cartella genitore ed il suo blocco; infine, è presente un flag, is_dir, che indica se il FileControlBlock appartiene ad una directory o ad un file. Quando viene creata la root, la cartella principale, il suo puntatore alla cartella genitore sarà NULL.
Il FirstDirectoryBlock e il FirstFileBlock sono le strutture che rappresentano i file nella loro totalità, ispirate agli inode, occupano tutto il primo blocco di un file e al loro interno ci sono le strutture definite precedentemente, utilizzando la tecnica dell'ereditarietà. Per primi conterranno un BlockHeader e un FileControlBlock, poi i vettori che contengono gli indici dei blocchi. Un primo array, il direct_blocks, contiene gli indici dei blocchi indirizzati direttamente ed è stato scelto di definirne 12 (valore che si può facilmente modificare tramite una macro). Il secondo, invece, indirect_blocks, contiene gli array dei blocchi indicizzati indirettamente, definendo fino al terzo livello di indirizzamento. Le due strutture differiscono per tre campi.

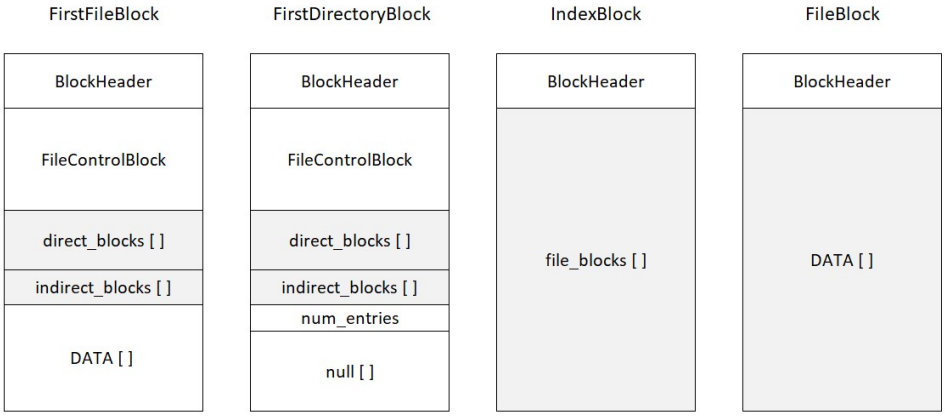

Figura 3.2. Struttura dei FirstBlock, IndexBlock e FileBlock

Nel FirstDirectoryBlock ci sarà un campo per indicare il numero di elementi presenti nella cartella ed un vettore di char, chiamato null, che verrà inizializzato a zero; è stato necessario inserirlo come padding, riempimento, per poter scrivere la struttura su disco. Bisogna ricordarsi che la memoria è scritta e letta per blocchi e, dato che il FirstDirectoryBlock è più piccolo del blocco definito (512 byte), occorre che i dati da scrivere che non portino informazioni aggiuntive. Il FirstFileBlock, al posto del vettore null, conterrà un array di char, DATA, che sarà il primo campo dove verranno inseriti i dati da scrivere all'interno del file. Le ultime due strutture cardine, che permettono il funzionamento di tutta l'opera, sono l'IndexBlock e il

FileBlock, entrambe contengono il BlockHeader. La prima contiene un array di interi che servirà ad indicizzare gli altri index block e i data block (in base di quale struttura è collegato lo specifico index block), mentre, la seconda, un array di char che conterrà i dati che verranno scritti sui file.
Nella seconda parte di strutture ci sono quelle di gestione, utili alle esecuzioni del programma. Il SimpleFS contiene il puntatore al DiskDriver, tramite il quale si ottengono tutte le informazioni del disco e della bitmap. Il DirectoryHandle e il FileHandle sono i riferimenti ai file, contengono il puntatore al SimpleFS, i puntatori al FirstDirectoryBlock e al FirstFileBlock. Il FileHandle contiene inoltre un intero, pos_in_file, che indica la posizione del cursore nel file e viene utilizzato principalmente per la scrittura e la lettura del file. Il DirectoryHandle, invece, avrà due interi: uno per la posizione del cursore nella directory e l'altro per la posizione relativa del cursore nel blocco.
Le funzioni che sono state implementate simulano le principali operazioni del filesystem, tra cui: la mkdir (make directory), la ls (list), la cd (change directory), la rm (remove), la open, la create e la read di un file. Per poter utilizzare il file system si dovranno simulare l'inizializzazione della root e la format del disco, la quale "pulisce" il file del diskdriver e le strutture. C'è stata una grande modularizzazione ed una frequente ricorrenza delle funzioni, poichè molte di queste hanno una fondamentale parte di ricerca dell'esistenza di un file. Queste funzioni ricevono come parametri degli interi che, funzionando da flag, fanno eseguire o meno specifiche istruzioni inerenti alla funzione che la chiama. Sono state create anche funzioni ausiliarie per la creazione e l'inizializzazione delle varie strutture.

3.3 Shell

Il codice della shell è molto lineare e semplice. Tutto si sviluppa all'interno di un main loop, il quale è in attesa di un'istruzione in input dall'utente. Per prima cosa sono state settate e inizializzate le funzioni messe a disposizione dalla libreria linenoise, importata e scaricata tramite la piattaforma gitHub come: l'auto completamento delle istruzioni attraverso il tasto <TAB>, i suggerimenti dei parametri da passare alle singole operazioni e il caricamento del file che conterrà la cronologia delle istruzioni eseguite precedentemente, per poter essere richiamate attraverso l'uso delle freccette da tastiera.
Al suo interno, vengono inizializzati il disco e il simple file system, stampando a schermo la riuscita delle funzioni. La libreria, inoltre, mette a disposizione un'operazione che permette di visualizzare a schermo una specifica stringa ogni volta che il loop si ripete: questa rappresenterà la linea di comando del nostro terminale.

Solitamente, un prompt presenta il percorso della directory attuale ed un cursore lampeggiante che indica l'attesa del comando. Per creare la stringa del percorso delle directory è stata scritta la funzione concat, libera ispirazione della strncat, funzione definita nella libreria C string.h e quest'ultima concatena due stringhe. La nostra versione, oltre ad unire due stringhe, che nel nostro caso sono il nome della directory padre e quella del figlio nella quale ci spostiamo, deve anche separare i nomi con uno "/" (slash), e terminare la stringa con un ">", che indica la fine del percorso e l'inizio dello spazio soggetto al comando dell'utente.
Ogni volta, all'interno del ciclo, viene salvata l'operazione appena scritta e ne viene fatto il parse, il quale divide l'istruzione dai suoi parametri e si eseguono a cascata una serie di if dove si confrontano il nome dell'input e il nome delle operazioni possibili. Appena ci sarà un match verrà chiamata la funzione del SimpleFS che svolgerà quel compito, in caso contrario, se nessuna operazione corrisponderà, si stamperà a schermo una stringa di non riconoscimento dell'operazione richiedendone un'altra all'utente.
Di particolare interesse è stato lo sviluppo nella shell delle operazioni di lettura e scrittura di un file. Quando, nel main loop, viene invocata la funzione *openFile* verrà chiamata una funzione ausiliaria composta anch'essa da un loop, la quale prenderà il "controllo" del programma. Questa, inoltre, stamperà a video dei suggerimenti e sarà in attesa delle operazioni di scrittura, lettura e seek che sono possibili fare su un file. Con il comando close si uscirà dal controllo del file e si tornerà nel main loop, in attesa di altre operazioni.
Per rendere il lavoro più comprensibile e user-friendly si è cercato di dare un maggior impatto visivo al terminale attraverso l'uso di colori in contrasto tra loro. Per esempio, la list di una cartella presenterà il nome della directory genitore, dei file e delle sotto-cartelle di colori diversi.

Capitolo 4

Test ed Esempi

È stato definito un MakeFile per eseguire e testare le varie componenti del codice. Sono stati effettuati numerosi test per verificare la presenza di errori e la corretta gestione di casi limite durante l'esecuzione dei comandi. Sono stati creati molti file e cartelle, lavorando con inode più piccoli, per controllare, per esempio, il comportamento del programma quando un file aveva raggiunto la sua grandezza massima, si è provato ad aprire cartelle per poterci scrivere, o a scrivere file non creati, ecc. Ogni volta i controlli delle funzioni rispettavano i vincoli, e non si sono verificate operazioni invalidanti.
Con il fine di comprendere fino in fondo il lavoro fatto, vengono presentati alcuni esempi pratici con una spiegazione del codice, mostrando gli output e le modifiche che le operazioni comportano sul disco.
Si è scelto di prendere in esame tre casi:

- List directory
  Il comando "ls" nei sistemi operativi Unix e Unix-like elenca le informazioni sui file e il contenuto delle directory. Basterà quindi digitare il comando sul nostro terminale che sarà stampato su schermo l'elenco di tutti gli elementi presenti nella cartella nella quale ci si trova. La Figura 4.1 è un semplice esempio di esecuzione del comando: il nome dei file è colorato di blu, quello delle cartelle di violetto e in rosso a sinistra è ripetuto il nome della cartella genitore. In alcuni sistemi, come per esempio in Linux, non è obbligatorio esplicitare il formato del file con l'estensione; leggendo solamente il nome di un file, si potrebbe non sapere a priori se questo sia un file o una cartella. Usando colori diversi, si riesce a distinguere più facilmente la natura degli elementi, per esempio nella Figura 4.1 si nota che *elenco_esami* non è una direcotry come *esami_vecchi*.
  Una volta inviato il comando, dalla shell sarà invocata la funzione *int Sim-*

*pleFS_readDir(char** names, DirectoryHandle* d)* che, dopo un controllo sui parametri in input, inizierà a visitare i blocchi indici e a salvare sull'array *names* i nomi degli elementi della cartella puntata da *d*. Nel codice sono state scritte le funzioni ausiliarie *cotrol_direct_block*, *control_single_block*, *control_double_block* e *control_triple_block*. Queste funzioni sono molto simili tra

```
mysmpFS://utente/ale>cd Documenti
mysmpFS://utente/ale/Documenti>ls
Documenti : slides
Documenti : esami_vecchi
Documenti : progetto.txt
Documenti : relazione.txt
Documenti : elenco_esami
mysmpFS://utente/ale/Documenti>
```

Figura 4.1. Esempio dell'esecuzione del comando "ls"

loro ed hanno tutte lo stesso compito. Per prime controllano l'array dell'indice e continuano la scansione dei blocchi fino a quando non incontrano una cella a -1. Le funzioni, come si intuisce dal nome, sono state scritte per ogni livello di indirizzamento. Le funzioni di livello di indirizzamento più alto al loro interno chiamano quelle del livello inferiore. Le funzioni sono costituite principalmente da cicli while e controlli sugli indici.
Per semplicità i codici ausiliari non sono stati scritti solo per la *SimpleFS_readDir*, ma tramite l'uso di flag e appropriati if, le funzioni sono state riutilizzate per la ricerca della open del file, della create, della changedir e della mkdir, ognuna con specifici moduli di codice. Per esempio, nella mkdir si usa lo stesso controllo della *SimpleFS_readDir*; con la differenza che nel momento in cui non viene trovato un elemento con lo stesso nome della cartella che stiamo creando allora verranno chiamate le funzioni sull'allocazione di un nuovo blocco e tutte le operazioni che comportano l'aggiornamento degli indici e delle grandezze.

- Scrittura di un file
  Un file per poter essere scritto deve essere prima aperto. Il compito verrà svolto dalla funzione *FileHandle* SimpleFS_openFile(DirectoryHandle* d, const char* filename)*. Per prima cosa controlla i parametri in input ed inizia a fare una ricerca sui blocchi, ricercando il file (*filename*) da aprire (usa le stesse funzioni di controllo usate per la *SimpleFS_readDir*); qualora non si trovi è stata adottata la policy creazione ed apertura. Nel momento in cui il file viene aperto, come già anticipato, il controllo della shell passa dal main loop principale ad un altro loop dedicato alla modifica dei file. Attraverso la Figura 4.2 si può notare che dopo la open viene modificato anche il layout della shell, presentando alcune righe che spiegano come lavorare con i file. La

```
mysmpFS://utente/ale/Documenti>open progetto.txt
To write use form: write text
To read use form: read n_bytes to read,if n_bytes == all  rea
d file until the end
To move the seek use form: seek n_of_bytes to move the cursor
 to n bytes, if n_bytes == all move cursor until the end
progetto.txt>write Il progetto del File System con Inode è un
 progetto che vuole simulare un moderno file system, saranno
disponibili le varie operazioni: tra le quali la ls, la mkdir
, la mkfile, la cd, la open, la rm, ecc.
progetto.txt>ls
Unreconized command
progetto.txt>cd
Unreconized command
progetto.txt>read all

Il progetto del File System con Inode è un progetto che vuole
 simulare un moderno file system, saranno disponibili le vari
e operazioni: tra le quali la ls, la mkdir, la mkfile, la cd,
 la open, la rm, ecc.
progetto.txt>
```

Figura 4.2. Apertura di un file esistente, seguita da un'operazioni di write e read

write potrà essere eseguita in append (a fine file) o in sovrascrizione ed ogni invocazione della *read* eseguirà in automatico una *seek* grande quanto i byte letti. Con il comando *seek* si può scrivere in qualsiasi punto del file, in input riceverà un intero che rappresenterà l'offset del cursore all'interno del file.
La funzione che viene invocata per scrivere è *int SimpleFS_write(FileHandle* f, void* data, int size)*, i dati che devono essere scritti, passati come parametro nella shell, vengono contenuti nel vettore *data*. Fondamentalmente la *SimpleFS_write* non farà altro che una copia delle informazioni puntate da *data*, sul vettore DATA presente nelle strutture FirstFileBlock e FileBlock del file puntato da *f*. Il punto cruciale dell'operazione è la gestione degli array e dell'offset in base alla posizione del cursore; bisogna garantire che le operazioni in overwriting e in append non corrompino sezioni di file diverse da quelle desiderate dall'utente. La scelta dell'allocazione dei blocchi indicizzata è molto di aiuto; come spiegato nel paragrafo 2.4, si può scrivere in qualsiasi punto del file in maniera veloce ed efficacie. In base al valore della seek si può facilmente risalire al numero del blocco sul quale si vuole scrivere facendo delle semplici operazioni di divisione intere, sottrazioni e divisioni con resto. In questo modo si conosce il numero del blocco indice sul quale posizionarsi e su quale cella si scriverà l'indice successivo.
La *SimpleFS_write* al suo interno invoca la funzione per la scrittura di un blocco del disco, la *int DiskDriver_writeBlock(DiskDriver* disk, void* src, int block_num)* che modificherà il file (rappresentante la memoria) e la bitmap. Se si apre il file di testo che simula il disco si può vedere (Figura 4.3) l'esecuzione

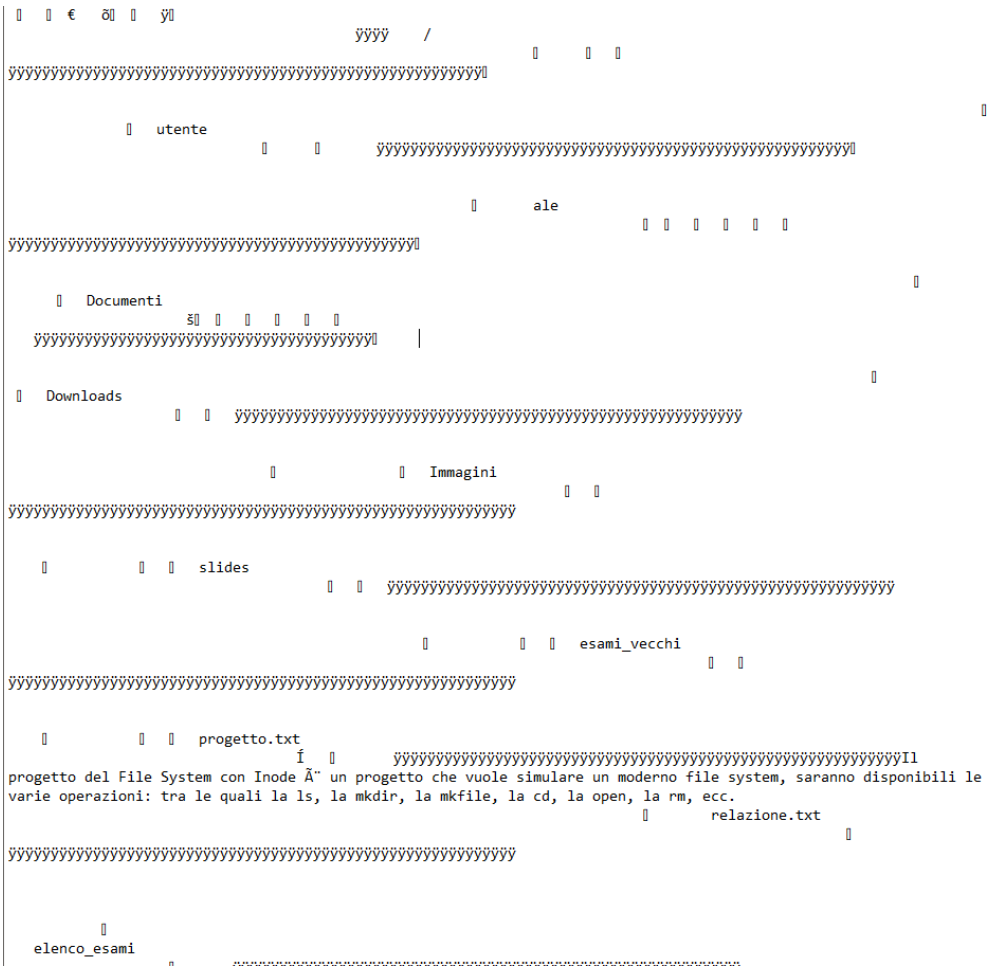

Figura 4.3. Visione del disco di memoria

della scrittura; in questo esempio è stato usato esclusivamente il campo dati del FirstFileBlock. Questa immagine del disco, inoltre, mostra i blocchi occupati, usando un editor di testo, ci saranno alcuni elementi delle strutture che non si riescono a decodificare, però il testo contenuto è leggibile, il che rende possibile comprendere la scrittura per blocchi dei file.

- Remove ricorsiva di file e cartelle
  La remove è sempre preceduta dalla ricerca del file da eliminare, nel caso in cui il controllo non avesse esito positivo viene mostrata una stringa di errore. La *int SimpleFS_remove(DirectoryHandle* d, char* filename)* è stata scritta in modo da poter funzionare sia con i file che con le cartelle. La remove di un file è costituita da un ciclo che elimina tutti i FileBlock e gli IndexBlock relativi a quel file, per poi eliminare il FirstFileBlock; ogni volta vengono aggiornati i campi della FirstDirectortBlock ed eventualmente il primo blocco libero. La remove di una cartella invece è implicitamente ricorsiva e si è voluto mantenere

questa caratteristica anche in questa versione. Durante l'eliminazione dei suoi elementi, ogni volta che si incontra una cartella, si chiama la funzione *SimpleFS_remove* ricorsivamente, la quale inizia di nuovo ad eliminare i file e le cartelle presenti nella stessa.
L'eliminazione di un file all'interno della directory che lo contiene comporta la modifica dell' IndexBlock, questo perché non si vuole che ci siano buchi con -1 all'interno del suo array. Quest'ultimi vengono sostituiti con l'ultimo elemento presente nell'ultimo IndexBlock in modo da non creare ambiguità. La remove,

Figura 4.4. Riproduzione del disco dopo aver eliminato la directory Documenti

come la write, si occupa della modifica del disco.
Nella figura 4.4 si può notare come tutti gli elementi della cartella Documenti siano stati eliminati: sia i file vuoti che quelli scritti, che le altre directory presenti.

Capitolo 5

Conclusioni

L'obiettivo di questo progetto è stato quello di comprendere e sviluppare le principali operazioni che vengono svolte da un file system ogni volta che viene creato o modificato un file sui nostri computer. Dopo un'introduzione generale della teoria, le strutture e le componenti sono state analizzate da un punto di vista logico e fisico. Si è lavorato con un file rappresentate la memoria, cercando di utilizzarlo nel modo più efficiente possibile, usando i concetti dell'allocazione indicizzata e della bitmap, per avere allocazioni e de-allocazioni veloci e sicure, senza corrompere i dati che venivano salvati. Sono stati fatti numerosi test di prova per verificare la robustezza del codice e tutte le possibili richieste dell'utente. Si è testata l'allocazione di più di 1000 elementi tra file e cartelle, controllando la gestione dei blocchi indici e l'aggiornamento delle varie strutture.
Infine, dopo la scrittura di tutte le strutture, delle funzioni e dei test, è stata scritta la shell. Quest'ultima è stata fondamentale per dare un'immagine al progetto, rendendolo fruibile ed intuitivo, terminando in questo modo il programma.

Bibliografia

[1] Silberschatz, Galvin, Gagne, *Sistemi Operativi. Concetti ed Esempi*, Pearson Ottava edizione, 2009

[2] Giorgio Grisetti, *Operating System - What is an Operating System*, Dispensa corso Sistemi Operativi, 2020

[3] Giorgio Grisetti, *Operating System - FileSystem*, Dispensa corso Sistemi Operativi, 2020

[4] Quaglia, Demetrescu, *Programmazione in Ambiente Unix*

[5] Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces, Chapter: Hard Disk Drives*, 2014, https://pages.cs.wisc.edu/ remzi/OSTEP/file-disks.pdf

[6] Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, *Operating System: Three Easy Pieces, Chapter: File System Implementation*, 2014, https://pages.cs.wisc.edu/ remzi/OSTEP/file-implementation.pdf

[7] Patterson, Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 1971

[8] *Operating Systems 600.418 The File System*, Department of Computer Science Johns Hopkins University, https://www.cs.jhu.edu/ yairamir/cs418/os7/sld001.htm [ultimo accesso 15/09/2021]

[9] *5.10. Filesystem*, https://tldp.org/LDP/sag/html/filesystems.html [ultimo accesso 10/09/2021]