SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Ingegneria Informatica, Automatica e Gestionale "Antonio Ruberti" (DIAG)

Shell con File System Persistente in C

Relazione di Progetto

Studente: Michael Ciotti
Matricola: 1956722

Corso: Sistemi Operativi
Docente: Prof. Giorgio Grisetti
Anno Accademico: 2024/2025

Indice

Introduzione

Il progetto presentato in questa relazione ha come obiettivo la realizzazione di una **shell interattiva** in linguaggio **C**, integrata con un **file system persistente** memorizzato su file immagine. L'intento è quello di riprodurre, in ambiente controllato, le principali funzionalità di un sistema operativo reale, offrendo un contesto didattico in cui mettere in pratica i concetti fondamentali di gestione delle risorse e organizzazione dei dati.

Lo sviluppo è stato affrontato con un approccio modulare, suddividendo il progetto in componenti indipendenti per la gestione dei comandi, dei file, delle directory e del file system. Questa scelta architetturale ha permesso di mantenere il codice ordinato e facilmente estendibile, semplificando le attività di manutenzione, test e documentazione.

Dal punto di vista tecnico, l'implementazione si basa sull'uso delle librerie standard e POSIX, con particolare attenzione all'impiego di tipi a larghezza fissa (`uint32_t`, `uint8_t`, ecc.) per garantire portabilità e coerenza nella rappresentazione dei dati. Il file system utilizza una struttura basata su blocchi e inode, con la gestione della persistenza affidata alla mappatura di memoria (`mmap`), che consente un accesso diretto ed efficiente ai dati memorizzati.

Durante la progettazione sono state privilegiate la chiarezza e la robustezza del codice: ogni operazione critica è stata accompagnata da controlli espliciti sugli input, gestione degli errori e procedure di recupero in caso di stati inconsistenti. Il processo di compilazione e testing è stato automatizzato tramite `Makefile` e script dedicati, garantendo riproducibilità e affidabilità dei risultati.

La relazione è organizzata in modo da descrivere progressivamente le varie fasi dello sviluppo: il capitolo dedicato ai *requisiti e strumenti* illustra le librerie utilizzate e le scelte tecnologiche adottate; l'*architettura del sistema* presenta la struttura dei moduli e la loro organizzazione; il capitolo sulle *scelte progettuali* analizza le decisioni implementative più rilevanti, mentre quello successivo descrive in dettaglio le *strutture dati* e le *funzioni principali* del sistema. Infine, la sezione di *testing e risultati* riporta i casi di prova e verifica la correttezza e la stabilità complessiva del progetto.

In questo modo, la relazione intende fornire una visione completa del percorso progettuale e delle soluzioni tecniche adottate, evidenziando come i principi dei sistemi operativi possano essere concretamente applicati alla realizzazione di un sistema funzionante e coerente.

Capitolo 1

Requisiti e strumenti

1.1 Strumenti e librerie utilizzate

Il progetto è stato sviluppato in linguaggio **C** utilizzando un approccio modulare e facendo ampio uso di librerie standard e POSIX. Le principali librerie e intestazioni incluse sono:

- <**stdio.h**> — gestione dell'input/output standard, funzioni come `printf`, `perror` ecc.;
- <**stdlib.h**> — gestione della memoria dinamica (`free`) e funzioni di utilità generale;
- <**string.h**> — manipolazione di stringhe (`strcmp`, `strcpy`, `strlen`, `strtok`, ecc.);
- <**stdint.h**> — tipi di dato interi a lunghezza fissa (`uint32_t`, `int8_t`, ecc.);
- <**errno.h**> — gestione e interpretazione dei codici di errore di sistema;
- <**sys/types.h**> — tipi POSIX usati per chiamate di sistema (`off_t`, `size_t`, ecc.);
- <**sys/stat.h**> — operazioni su file e directory, funzioni come `mkdir`;
- <**sys/mman.h**> — mappatura della memoria (`mmap`, `munmap`) per la gestione del file system persistente;
- <**unistd.h**> — chiamate di sistema POSIX (`close`);
- <**fcntl.h**> — gestione dei file descriptor (`open`) e modalità di apertura (`O_RDONLY`, `O_WRONLY`, `O_CREAT`, `O_RDWR` ecc.);
- <**dirent.h**> — gestione delle directory, lettura di entry tramite `opendir`, `readdir`;
- <**time.h**> — gestione dei timestamp e operazioni temporali (`struct tm`, `localtime_r`, `strftime`);
- <**ctype.h**> — controllo e conversione di caratteri (`isspace`);
- <**readline/readline.h**> — input interattivo da terminale con cronologia dei comandi;

- <**readline/history.h**> — gestione della cronologia dei comandi precedenti;

Le librerie **POSIX** sono state utilizzate per la gestione diretta delle risorse di sistema (file, directory, memoria mappata), mentre le librerie standard C hanno garantito la portabilità del codice e la gestione dei dati in memoria.

Il progetto utilizza inoltre la libreria `readline`, che semplifica l'interazione con l'utente fornendo funzionalità di editing e cronologia dei comandi all'interno della shell.

Capitolo 2

Architettura del sistema

2.1 Struttura delle directory

```
├── img/
├── include/
│   ├── commands.h
│   ├── dir_util.h
│   ├── file_util.h
│   ├── fs.h
│   └── gen_util.h
├── src/
│   ├── commands.c
│   ├── dir_util.c
│   ├── file_util.c
│   ├── fs.c
│   ├── gen_util.c
│   └── shell.c
├── LICENSE
├── Makefile
├── README.md
└── SO_FS_Project_Report.pdf
```

2.2 Organizzazione modulare

Ogni file sorgente ha una funzione ben definita:

- **shell.c**: gestione dell'interfaccia utente da riga di comando e del ciclo di esecuzione;
- **commands.c** / **commands.h**: implementazione dei comandi disponibili;
- **fs.c** / **fs.h**: definizione e gestione del file system;
- **dir_util.c** / **dir_util.h**: funzioni di supporto per directory;
- **file_util.c** / **file_util.c**: funzioni per lettura/scrittura di file;
- **gen_util.c** / **gen_util.c**: funzioni ausiliarie utili per diverse operazioni legate ai comandi.

Capitolo 3

Scelte progettuali

3.1 Tipi interi a larghezza fissa

Per garantire portabilità e coerenza nella rappresentazione dei dati all'interno del file system, sono stati utilizzati tipi interi a larghezza fissa come `uint32_t`, `uint8_t` e affini, definiti in `<stdint.h>`. Questi tipi assicurano che, indipendentemente dall'architettura hardware o dal compilatore, il numero di bit allocati per ogni variabile rimanga invariato. Ad esempio, `uint32_t` garantisce un intero senza segno di 32 bit, utile per contatori, puntatori diretti o dimensioni di blocchi, mentre `uint8_t` è impiegato per flag o campi che devono occupare un solo byte. In questo modo si evita l'incertezza propria dei tipi generici (come `int` o `unsigned int`), il cui numero di bit può variare a seconda della piattaforma.

3.2 Struttura modulare del sistema

La suddivisione del progetto in moduli distinti (shell, gestione comandi, file system, utilità) risponde al principio della "separazione delle responsabilità". Ogni modulo è stato progettato per svolgere una funzione ben precisa, riducendo le dipendenze incrociate e facilitando la manutenzione futura. Questa architettura modulare ha semplificato il testing, la lettura del codice e l'estensibilità del sistema nel suo complesso.

3.3 Allocazione e gestione dello spazio

Per semplificare la gestione dello spazio e rendere il progetto adatto all'ambito didattico, è stata adottata una struttura di allocazione basata su blocchi di dimensione fissa (ad esempio 4096 byte) e su un numero definito di puntatori diretti (`DIRECT_PTRS = 8`) per ogni inode. Questa scelta garantisce un equilibrio fra semplicità implementativa e comportamento realistico.

3.4 Gestione degli errori e consistenza

La robustezza del sistema è stata perseguita introducendo verifiche sistematiche sugli input, controlli sull'allocazione e deallocazione delle risorse e procedure di recupero in caso di stato inconsistente. Ogni operazione critica (creazione di file, rimozione ricorsiva di directory, chiusura del filesystem) è accompagnata da controlli espliciti e messaggi di errore chiari. Inoltre, l'uso di tipi a larghezza fissa contribuisce a evitare comportamenti indefiniti dovuti a overflow o interpretazione diversa dei dati.

Nel capitolo seguente verranno illustrate in dettaglio le strutture dati e l'implementazione dei comandi che compongono il sistema.

Capitolo 4

Strutture dati del File System

4.1 Superblock

Il **superblock** contiene informazioni globali sull'intero file system:

```
typedef struct {
    uint32_t signature;     // Firma identificativa "MCFS"
    uint32_t version;       // Versione del FS
    uint32_t block_size;    // Dimensione di un blocco (4096 byte)
    uint32_t total_blocks;  // Numero totale di blocchi
    uint32_t inode_count;   // Numero totale di inode
    uint32_t root_inode;    // Numero dell'inode della root
    uint32_t bitmap_blocks; // Numero di blocchi occupati dalla bitmap
    uint32_t inode_blocks;  // Numero di blocchi occupati dalla tabella degli inode
} Superblock;
```

4.2 Inode

Ogni **inode** rappresenta un file o una directory:

```
typedef struct {
    uint8_t  type;              // Macro INODE_FREE=0, INODE_FILE=1, INODE_DIR=2
    uint32_t size;              // Dimensione del file
    uint32_t parent;            // Numero dell'inode del padre
    uint32_t direct_ptrs[8];    // Puntatori diretti ai blocchi dati
} Inode;
```

4.3 Directory entry

Le directory contengono coppie nome/inode:

```
typedef struct {
    int32_t inode;                  // Numero dell'inode
    char name[MAX_NAME];            // Nome della entry
} DirEntry;
```

4.4 Struttura FS

La struttura principale del sistema è la **struct FS**, che rappresenta un file system montato:

```
typedef struct {
    int fd;                             // File descriptor del file immagine
    char fs_filename[MAX_NAME];         // Nome del file immagine (MAX_NAME=56)
    void *base;                         // Puntatore base all'area mmap
    Superblock *sup_b;                  // Puntatore al superblock
    uint8_t *bitmap;                    // Puntatore alla bitmap dei blocchi
    Inode *inode_tab;                   // Puntatore alla tabella inode
    uint8_t *data;                      // Inizio dei blocchi dati
    uint32_t cwd_inode;                 // Numero di inode della cwd
} FS;
```

Quando un file system viene aperto, viene allocata una struttura `FS` che mappa tutte le sue componenti, rendendo possibile l'accesso diretto alla memoria persistente tramite offset.

Capitolo 5

Descrizione delle funzioni principali

5.1 fs.c / fs.h

- **fs_bind(FS *fs)**: associa i puntatori interni (superblock, bitmap, inode, data) alla base mmap;
- **alloc_block()**: trova un blocco libero nella bitmap e lo marca come occupato;
- **alloc_inode(InodeType t, uint32_t parent)**: alloca un nuovo inode di tipo file o directory;
- **free_inode_blocks(Inode *inode)**: libera tutti i blocchi associati a un inode e aggiorna la bitmap;
- **inode_ensure_block(Inode *inode, int slot)**: alloca un nuovo blocco dati se necessario;
- **block_ptr(uint32_t block)**: calcola il puntatore reale a un blocco dati.

5.2 commands.c / commands.h

- **cmd_open(const char *path)**: apre un file immagine e lo mappa in memoria;
- **cmd_close()**: chiude il file system e "unmappa" la memoria;
- **cmd_format(const char *path)**: inizializza un nuovo FS (superblock, bitmap, root);
- **cmd_mkdir(const char *path)**: crea una nuova directory;
- **cmd_touch(const char *path)**: crea un file vuoto;
- **cmd_append(const char *path, const char *text)**: scrive i dati di **text** nel file in **path**;
- **cmd_cat(const char *path)**: stampa il contenuto di un file;
- **cmd_ls(const char *path)**: elenca le entry di una directory;

- **cmd_rm(const char *path, const char *flag)**: elimina file o directory (ricorsivo con -r, ricorsivo forzato con -rf: in tali casi all'interno della funzione viene chiamata la funzione statica **cmd_rm_recursive(int32_t inode, int force)** che prima di eliminare la cartella indicata in input, svuota ed elimina ricorsivamente le altre cartelle e i file presenti al suo interno);
- **cmd_images()**: mostra i file immagine disponibili;
- **cmd_help()**: mostra l'elenco dei comandi.

5.3 dir_util.c / dir_util.h

- **dir_append_entry(int dir_inode, const char *name, int target_inode)**: aggiunge una nuova entry alla directory identificata da dir_inode;
- **dir_find(Inode *dir, const char *name, int *out_slot, DirEntry *out)**: cerca un file in una directory (utile per mostrare un messaggio di errore in caso di operazione che creano duplicati);
- **path_to_inode_n(const char *path, int parent_req, char *name)**: partire da un path calcola il numero dell'inode corrispondente (in particolare se parent_req=1 restituisce il numero dell'inode del padre e in name il nome della cartella, utile per mkdir; invece se è 0, restituisce il numero dell'inode dell'elemento in fondo al path, utile per cd, ls);
- **list_dir_entries(int32_t inode, char **entries, int max_entries)**: legge tutte le entries di una directory e ne inserisce il nome nell'array di stringhe passato nei parametri (le directory verranno inserite come "dir_name/" e i file semplicemente come "filename");
- **dir_remove_entry(Inode *dir, char *name)**: rimuove una entry (con il nome "name") dalla directory identificata da dir;
- **is_dir_empty(int dir)**: verifica se una directory è vuota.

5.4 file_util.c / file_util.h

- **file_read(int inode)**: legge e stampa il contenuto di un file;
- **file_write(int inode, const char *text)**: scrive dati in append;

5.5 gen_util.c / gen_util.h

- **clean(char *s)**: rimuove spazi e caratteri di controllo da una stringa;
- **tokenize(char *line, char **argv, int max)**: suddivide la riga in input in argomenti che inserisce in argv;
- **die(const char *msg)**: stampa un errore e termina il programma;
- **check_ext(const char *name)**: controlla se l'immagine ha l'estensione corretta (.img);
- **img_dir()**: controlla se esiste la cartella img dove andrà il file persistente, ed eventualmente in caso negativo la crea;
- **ensure_opened()**: controlla se un filesystem è aperto;
- **get_cwd_label()**: cerca e ritorna il nome della directory corrente;
- **build_prompt()**: costruisce la stringa del prompt;
- **dot_case(char *name)**: ritorna 1 se "name" è `.` o `..`, altrimenti 0.

Capitolo 6

Testing e risultati

6.1 Script di test automatico

```
./shell <<'EOF'
format test.img 1048576
open test.img
mkdir /a
mkdir /a/b
touch /a/b/x.txt
append /a/b/x.txt "ABC"
cat /a/b/x.txt
ls /a/b
rm /a/b/x.txt
rm -r /a
close
exit
EOF
```

6.2 Risultati

I test hanno confermato:

- corretta formattazione e inizializzazione del file system;
- corretta gestione delle directory annidate;
- persistenza dei dati tra diverse esecuzioni;
- robustezza nella gestione di errori (spazio insufficiente, path errati);
- comportamento coerente con l'uso ricorsivo di `rm -r` (e `rm -rf`).

Conclusioni

Lo sviluppo del progetto ha rappresentato un esercizio tecnico volto a tradurre in pratica i concetti teorici affrontati nel corso di *Sistemi Operativi*, con particolare attenzione alla gestione delle risorse, alla struttura del file system e all'interazione con le chiamate di sistema POSIX. L'obiettivo principale era la realizzazione di una **shell funzionante** dotata di un **file system persistente** interamente progettato, curando gli aspetti di architettura, affidabilità e coerenza del sistema.

L'attività ha richiesto una pianificazione accurata delle componenti e la definizione di un'architettura modulare e scalabile. Il progetto è stato organizzato in sottosistemi distinti — shell, comandi, gestione di file e directory, moduli di utilità — secondo una chiara separazione delle responsabilità, che ha favorito la manutenibilità e la leggibilità del codice.

Il file system è stato implementato mediante l'utilizzo di un **file immagine** gestito tramite **mappatura di memoria** (`mmap`), approccio che consente l'accesso diretto ai dati e garantisce la persistenza delle informazioni tra sessioni diverse. Questa soluzione ha permesso di approfondire concetti fondamentali relativi all'**organizzazione fisica dei dati**, alla **gestione dei blocchi di memoria** e ai meccanismi di **lettura e scrittura**, replicando in modo realistico il funzionamento di un file system reale in ambiente simulato.

Una parte significativa del lavoro è stata dedicata alla **gestione degli errori** e alla salvaguardia della **consistenza dei dati**. Le operazioni critiche sono state protette attraverso controlli di validità, verifiche di allocazione della memoria e procedure di recupero per garantire il corretto stato del sistema anche in presenza di anomalie. Questo approccio ha ridotto il rischio di corruzione del file system e migliorato l'affidabilità complessiva del software.

Dal punto di vista metodologico, lo sviluppo è stato condotto secondo principi di **ingegneria del software di basso livello**, privilegiando la chiarezza strutturale e la riproducibilità. L'uso di un **Makefile** ha automatizzato il processo di compilazione e semplificato la gestione delle dipendenze, mentre l'adozione di **naming convention coerenti** e di una documentazione essenziale ha contribuito a mantenere ordine e coerenza tra i moduli.

Nel complesso, il risultato è un sistema stabile e ben strutturato, capace di simulare efficacemente le dinamiche di un ambiente operativo reale. Il progetto ha consentito di consolidare competenze avanzate nella programmazione di sistema, nella gestione della memoria e nell'uso delle API POSIX, fornendo una comprensione più profonda dei meccanismi interni dei sistemi operativi e del loro disegno architetturale.