José Sebastian Pirir Romero - 202300335

Manual Técnico — Proyecto 2 MIA

1. Arquitectura General

El sistema se compone de dos capas principales: **frontend (React)** y **backend (Go)**.
El backend implementa toda la lógica del sistema de archivos virtual `.mia`, mientras que el frontend proporciona una interfaz para ejecutar scripts de comandos.

```
Usuario → Frontend (React) → API HTTP → Backend (Go) → Disco virtual (.mia)
```

`Estructura de carpetas`

- **frontend/**
  - `src/App.js`: ejecuta scripts línea por línea con `executeCommands`.
  - `src/components/`: vistas, resultados, logs.
- **backend/**
  - **commands/**: cada comando implementa una función `ExecuteX()` (mkdisk, fdisk, mount, mkfs, mkfile, cat, copy, recovery...).
  - **fs/**: lectura y escritura de estructuras (inodos, bloques, bitmaps). Contiene funciones base del sistema de archivos.
  - **structs/**: definición binaria de estructuras persistentes (MBR, EBR, Superblock, Inode, FolderBlock, FileBlock...).
  - **analyzer/**: parser de scripts; separa tokens y llama `commands.ExecuteX`.
  - **state/**: estado global (sesión activa, particiones montadas, IDs asignados).

Flujo interno de un comando

1. El usuario ejecuta un script en el frontend (ejemplo: `mkfile -path=/test.txt -size=100`).
2. El frontend envía el texto al backend por API.
3. El módulo `analyzer` tokeniza y llama `commands.ExecuteMkfile()`.
4. Este comando usa funciones del paquete `fs` (por ejemplo `fs.CreateFile()`).
5. `fs` lee el superbloque, localiza el inodo padre, asigna bloques e inodos nuevos y actualiza bitmaps.

6. Los cambios se escriben con `binary.Write` sobre el archivo `.mia`.

2. Estructuras Persistentes Clave

Las estructuras en `backend/structs` se escriben directamente al archivo `.mia`.
Cualquier cambio en ellas **rompe la compatibilidad binaria → siempre reformatear con `mkfs`**.

MBR y EBR

- **MBR**
  - Contiene las 4 entradas de partición: `mbr_partitions[4]`.
  - Campos: `Part_status`, `Part_type ('P','E','L')`, `Part_fit`, `Part_start`, `Part_size`, `Part_name`.
- **EBR**
  - Estructura encadenada dentro de la partición extendida.
  - Contiene `Part_next` apuntando al siguiente EBR lógico.

Superblock

Define la geometría del sistema de archivos.

```
type Superblock struct {
     S_inodes_count, S_blocks_count, S_free_inodes_count,
     S_free_blocks_count int32

     S_first_ino, S_first_blo,     S_inode_start, S_block_start int32

     S_bm_inode_start,   S_bm_block_start int32

     S_inode_size, S_block_size int32

     S_filesystem_type, S_magic int32

     S_mtime, S_umtime int64
}
```

- `S_block_size`: tamaño real del bloque (usualmente `binary.Size(FileBlock{})`).
- `S_inode_start, S_block_start: desplazamientos en disco.`

José Sebastian Pirir Romero - 202300335

Inode

Representa archivos o directorios.

```
type Inode struct {
     I_uid, I_gid int32
     I_size int32
     I_atime, I_ctime, I_mtime int64
     I_block [15]int32
     I_type byte // 0 carpeta, 1 archivo
     I_perm int32
}
```

- I_block: 12 directos + 3 indirectos.
- I_type: diferencia entre carpeta y archivo.

FolderBlock y FileBlock

- **FolderBlock**
  Contiene 4 entradas (nombre e inodo asociado):

```
type FolderBlock struct {
     B_content [4]ContentEntry
}
type ContentEntry struct {
     B_name [NAME_MAX]
     byte B_inodo int32
}
```

- **FileBlock**
  Contiene datos reales del archivo.

```
type FileBlock struct {
    B_content [FILE_BLOCK_SIZE]byte
}
```

Importante:
FILE_BLOCK_SIZE ≥ tamaño de FolderBlock.
Al formatear (mkfs), se usa S_block_size =
max(binary.Size(FolderBlock{}), binary.Size(FileBlock{})).

José Sebastian Pirir Romero - 202300335

3. Comportamiento del Sistema de Archivos

1. **mkfs**: inicializa superbloque, bitmaps e inodos raíz.
2. **mkdir / mkfile**: buscan inodos libres, asignan bloques, actualizan bitmaps e índices.
3. **cat / copy**: leen bloques en orden según `I_block`.
4. **journaling (si EXT3)**: cada operación genera una `JournalEntry` antes de aplicarse.
5. **recovery**: al detectar corrupción, recorre journaling y superbloque para reconstruir el estado coherente.

4. Verificaciones y Pruebas de Integridad

Antes de ejecutar pruebas funcionales:

- **Verifica tamaños binarios:**

  ```
  go run tools/check_sizes.go
  ```

(puede imprimir `binary.Size(structs.FileBlock{})` y `binary.Size(structs.FolderBlock{})` para asegurar consistencia).

- **Prueba de lectura/escritura:**
  1. Crea una carpeta `/test` y un archivo `/test/a.txt`.
  2. Usa `cat -file=/test/a.txt`.
  3. Verifica que el contenido sea idéntico a lo escrito.
- **Bitmaps:** tras mkfs, los bitmaps deben reflejar 1s en inodo raíz y primer bloque.

José Sebastian Pirir Romero - 202300335

5. Diagnóstico de Problemas Frecuentes

Problema	Causa	Solución
`users.txt` truncado	Tamaño inconsistente entre `S_block_size` y `FileBlock`	Recalcula FILE_BLOCK_SIZE, recompila y reformatea con `mkfs`.
Errores binarios en lectura	Campos añadidos en structs sin reformatear	Siempre recrear partición tras cambios de struct.
Bloques vacíos al copiar archivos	Función usa tamaño fijo (len=64) en lugar de `sb.S_block_size`	Revisa `fs.WriteFileBlock` y `fs.ReadFileBlock`.
Recovery no restaura	Journaling no fue guardado correctamente o faltan rutas absolutas	Revisa `commands.RecoveryFileSystem()` y journaling path completo.

6. Flujo de Journaling y Recovery (EXT3)

1. Cada comando crea una `JournalEntry`:

```
type JournalEntry struct {
   Operation [10]byte  // MKDIR, MKFILE, MKUSR, etc.
   Path [200]byte
   Content [200]byte
   Date int64
}
```

`2. fs.ApplyJournal()` recorre entradas no ejecutadas:

- Verifica si el inodo existe.
- Reaplica la operación (`MKDIR`, `MKFILE`, etc.).

`3. recovery -id=<id>`:

- Lee superbloque, bitmaps y journaling.
- Reconstruye S_first_ino/S_first_blo y S_free_*.
- Reescribe bitmaps y superbloque actualizados.

José Sebastian Pirir Romero - 202300335

7. Ejemplos de Comandos y Flujo

mkdisk -size=26 -unit=M -fit=FF -path=/home/jose/Discos/DiscoA.mia
fdisk -type=P -unit=M -name=PartA1 -size=8 -path=/home/jose/Discos/DiscoA.mia -fit=BF
mount -path=/home/jose/Discos/DiscoA.mia -name=PartA1
mkfs -type=full -fs=3fs -id=351A
login -user=root -pass=123 -id=351A
mkgrp -name=usuarios
mkusr -user=user1 -pass=pass353 -grp=usuarios
mkdir -p -path=/home/user1/docs
mkfile -r -path=/home/user1/docs/readme.txt -size=128
cat -file=/home/user1/docs/readme.txt
copy -path=/home/user1/docs/readme.txt -destino=/home/user1/docs/readme_copy.txt
recovery -id=351A

8. Buenas Prácticas

- Cada cambio en `structs` → **reformatear particiones (`mkfs`)**.
- Evita valores fijos en tamaños de lectura/escritura → usa `sb.S_block_size`.
- Implementa pruebas unitarias en `fs/` para garantizar integridad binaria.
- Mantén consistentes los offsets (`S_inode_start`, `S_block_start`).
- Usa discos de prueba `.mia` distintos para desarrollos experimentales.

9. Archivos de Referencia

- **Estructuras:** `backend/structs/fs_structs.go`, `inode.go`, `superblock.go`
- **Comandos:** `backend/commands/mkfs.go`, `mkfile.go`, `copy.go`, `cat.go`, `mount.go`, `recovery.go`
- **Analizador:** `backend/analyzer/analyzer.go`
- **Frontend principal:** `frontend/src/App.js`