Taller File System

Tomás Felipe Melli

Noviembre 2024

Índice

1 Introducción **2**
- 1.1 Conocimientos previos . . . 2
  - 1.1.1 Definición de File System . . . 2
  - 1.1.2 Definición de Archivo . . . 2
  - 1.1.3 Definición de Inodo . . . 2

2 Funciones **5**
- 2.1 blockgroup_for_inode (unsigned int inode) . . . 5
- 2.2 blockgroup_inode_index(unsigned int inode) . . . 5
- 2.3 load_inode(unsigned int inode) . . . 5
- 2.4 get_block_address(struct Ext2FSInode * inode, unsigned int block_number) . . . 6
- 2.5 get_file_inode_from_dir_inode(struct Ext2FSInode * from, const char * filename) . . . 7
- 2.6 inode_for_path(const char * path) . . . 9

3 Ejercicio tipo parcial : recorrer en EXT2 y en FAT **10**

1 Introducción

En este taller se pondrán en práctica los conocimientos de File System. En particular, se trabajará con un FS basado en **inodos**. La idea es poder, implementar la lógica de las funciones principales para trabajar con estas estructuras :

a) blockgroup_for_inode : para poder encontrar el **grupo** en que se encuentra el **inodo** pasado por parámetro.

b) blockgroup_inode_index : para encontrar el índice de cierto inodo dentro de la **tabla de inodos**.

c) load_inode : para cargar el **bloque** que contiene al inodo que buscamos en memoria y devolverlo. Le pasamos el nro de inodo.

d) get_block_address : con esta función podremos obtener la **LBA** o **logical block address** del bloque que contiene al inodo que pasamos por parámetro (también con su nro de bloque).

e) get_file_inode_from_dir_inode : es una función que a partir del **filename** queremos traernos su inodo. Tenemos la entrada de cierto directorio para buscar allí.

1.1 Conocimientos previos

1.1.1 Definición de File System

Un **File System** o sistema de archivos es un **ordenamiento lógico dentro de un disco**. Es decir, se busca establecer un orden para la información del usuario y del SO dentro de un dispositivo de almacenamiento. El responsable de este orden es un **módulo del kernel**. Este se encargará de :

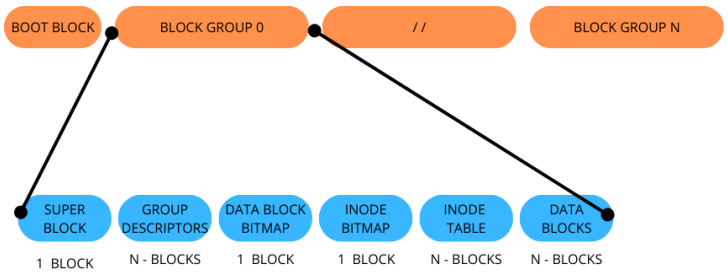
- **Organización interna** : estructura interna del archivo.
- **Organización externa** : cómo se ordenan los archivos ? Hay limitaciones (mismo nombre,...)? Aparece el concepto de **directorio** donde se conforma la estructura jerárquica de árbol

1.1.2 Definición de Archivo

Hablamos mucho de ordenar los archivos, de darle cierta estructura arbórea... pero qué es un archivo ? La respuesta es que es **una secuencia sin estructura de bytes(a nivel SO)**. El SO los identifica con un nombre y la extensión le tira medio un centro sobre lo que **podría tener, esto no es determinante en cuanto a lo que tendrá dentro**.
** Existen tipos : d (directory), c (char device), b (block device) , - (regular), ... tirá ls -l en algún lado y chequealo.
Y cómo hace este módulo para modelar un sistema lleno de secuencias de bytes sin estructura ?

1.1.3 Definición de Inodo

Un **inodo** es una estructura de datos que se utiliza en el FS que estudiaremos en este taller que se llama EXT2 (Extended File System 2) que se utiliza en Linux (y en otros) para **almacenar la información de un archivo**. Todo es un archivo en EXT2. La estructura del File System es :



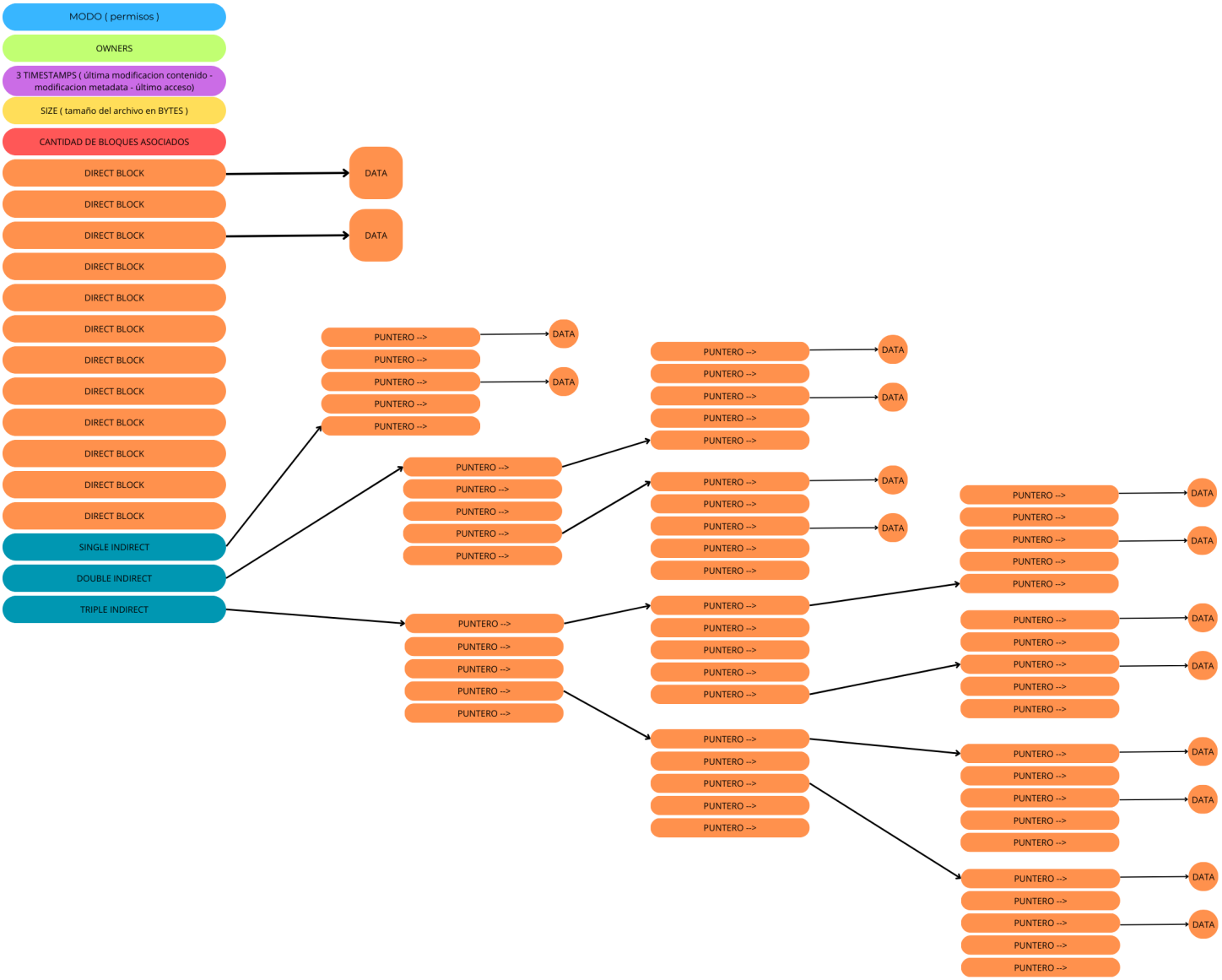
Tendremos :

- **Superblock** : contiene los metadatos críticos del FS:

```
struct __attribute__((__packed__)) Ext2FSSuperblock
{
    unsigned int inodes_count;
    unsigned int blocks_count;
    unsigned int reserved_blocks_count;
    unsigned int free_blocks_count;
    unsigned int free_inodes_count;
    unsigned int first_data_block;
    unsigned int log_block_size;
    unsigned int log_fragment_size;
    unsigned int blocks_per_group;
    unsigned int fragments_per_group;
    unsigned int inodes_per_group;
    unsigned int mount_time;
    unsigned int write_time;
    unsigned short mount_count;
    unsigned short max_mount_count;
    unsigned short magic_bytes;
    unsigned short state;
    unsigned short errors;
    unsigned short minor_revision_level;
    unsigned int lastcheck;
    unsigned int checkinterval;
    unsigned int creator_os;
    unsigned int revision_level;
    unsigned short default_reserved_userid;
    unsigned short default_reserved_groupid;
    // -- EXT2_DYNAMIC_REV Specific --
    unsigned int first_inode;
    unsigned short inode_size;
    unsigned short block_group_number;
    unsigned int feature_compatibility;
    unsigned int feature_incompatibility;
    unsigned int readonly_feature_compatibility;
    char  uuid[16];
    char volume_name[16];
    char last_mounted[64];
    unsigned int algo_bitmap;
    // Performance hints
    unsigned char prealloc_blocks;
    unsigned char prealloc_dir_blocks;
    char alignment[2];
    // Journaling support
    char journal_uuid[16];
    unsigned int journal_inode;
    unsigned int journal_device;
    unsigned int last_orphan;
    // Directory indexing support
    unsigned int hash_seed[4];
    unsigned char default_hash_version;
    char padding[3];
    // Other options
    unsigned int default_mount_options;
    unsigned int first_meta_bg;
    char unused[760];
};
```

- **Descriptor de grupo** : ayuda al SO a manejar permisos y acceso a recursos.
- **Tabla de inodos** : contiene los inodos del grupo
- **Bloques de datos**
- **Bitmaps** de inodos y de bloques de datos

Y el Inodo luce algo como esto :



Y su representación en C es como sigue :

```c
struct __attribute__((__packed__)) Ext2FSInode {
    unsigned short mode;
    unsigned short uid;
    unsigned int size;
    unsigned int atime;
    unsigned int ctime;
    unsigned int mtime;
    unsigned int dtime;
    unsigned short gid;
    unsigned short links_count;
    unsigned int blocks;
    unsigned int flags;
    unsigned int os_dependant_1;
    unsigned int block[15];
    unsigned int generation;
    unsigned int file_acl;
    unsigned int directory_acl;
    unsigned int faddr;
    unsigned int os_dependant_2[3];
};
```

La función principal de esta estructura es representar un **archivo**. Pero qué pasa con los directorios ? También se representan con estos, la estructura es igual. Lo que cambia es que cada bloque apuntado, contendrá **DirEntries** de la siguiente forma :

```
struct __attribute__((__packed__)) Ext2FSDirEntry {
        unsigned int inode;
        unsigned short record_length;
        unsigned char name_length;
        unsigned char file_type;
        char name[];
};
```

2 Funciones

2.1 blockgroup_for_inode (unsigned int inode)

Hablamos un poco del **superbloque**, este tiene mucha info, nos interesará saber **cuántos inodos hay por grupo** ya que, como sabemos el número del inodo podemos saber en qué grupo estará de la siguiente forma :

$$\frac{\text{inode} - 1}{\_superblock \rightarrow \text{inodes\_per\_group}}$$

2.2 blockgroup_inode_index(unsigned int inode)

En esta función queremos saber en qué posición dentro de la **inode table** dentro del grupo estará el inodo.

$$(\text{inode} - 1) \bmod \_superblock \rightarrow \text{inodes\_per\_group}$$

2.3 load_inode(unsigned int inode)

La idea es cargarlo en memoria al inodo. Los pasos son :

a) Saber en qué número de grupo está con blockgroup_for_inode.

b) Conocer el **descriptor del grupo** con la función **block_group(unsigned int grupo)** que básicamente indexa en una tabla que tiene los descriptores de los grupos. Este objeto tiene la **dirección de la inode table**.

c) Una vez capturada la dire de la tabla, tenemos que saber dónde indexar según el inodo que tenemos. Llamamos a blockgroup_inode_index.

d) **Aclaración : un inodo normalmente ocupa mucho menos espacio que lo que ocupa un bloque entero en memoria. Es por ello que tenemos que considerar esto a la hora de hacer la carga del inodo en memoria. Dicho esto, capturamos el **tamaño de bloque**, este dato lo tiene el **superblock** bajo el atributo $\_superblock \rightarrow log\_block\_size$, pero ojo, porque este valor es log_2 del tamaño. Entonces tenemos que llevarlo a KB (simplemente se multiplica por 2^{10} bytes. Entonces, una vez calculado el tamaño en KB, queremos saber **cuántos inodos habrá por bloque**

$$\text{cant\_inodos\_por\_bloque} = \frac{\text{tam\_bloque}}{\_superblock \rightarrow \text{inode\_size}}$$

e) Pensemos en algo antes, la tabla de inodos está en memoria también, y no necesariamente ocupa 1 bloque, entonces si nuestro inodo está dentro de esa tabla, en particular podría no estar en el bloque 1 que la contiene. Para asegurarse qué bloque que contiene la tabla de inodos contiene al inodo que buscamos hacemos la siguiente cuenta :

$$\text{bloque\_inodo} = \frac{\text{índice}}{\text{cant\_inodos\_por\_bloque}}$$

f) Una vez que sabemos el bloque de la tabla de inodos que contiene el inodo que buscamos, necesitamos el **offset** adecuado :

$$\text{offset\_inodo} = (\text{índice}\%\text{cant\_inodos\_por\_bloque}) \times \_superblock \rightarrow \text{inode\_size}$$

g) Nos traemos el bloque con **read_block(unsigned int block_address, unsigned char * buffer)** y en el lugar adecuado leemos y copiamos la info que nos interesa.

Solución:

```
struct Ext2FSInode * Ext2FS::load_inode(unsigned int inode_number)
{
    unsigned int nro_grupo = blockgroup_for_inode(inode_number);
    Ext2FSBlockGroupDescriptor* descriptor = block_group(nro_grupo);
    unsigned int dire_tabla = descriptor->inode_table;
    unsigned int indice = blockgroup_inode_index(inode_number);
    unsigned int tam_bloque = 1024 << _superblock->log_block_size;
    unsigned int cant_inodos_por_bloque =  tam_bloque / _superblock->inode_size;
    unsigned int bloque_inodo = indice / cant_inodos_por_bloque ;
    unsigned int offset_inodo = (indice % cant_inodos_por_bloque) * _superblock->inode_size;

    unsigned char* res  = (unsigned char*) malloc(sizeof(Ext2FSInode));
    unsigned char* lectura = (unsigned char*) malloc(sizeof(tam_bloque));

    read_block(dire_tabla + bloque_inodo, lectura);

    for (int i = 0; i < sizeof(Ext2FSInode); i++)
    {
        res[i] = lectura[offset_inodo + i];
    }

    delete[] lectura;
    return (Ext2FSInode*)res;
}
```

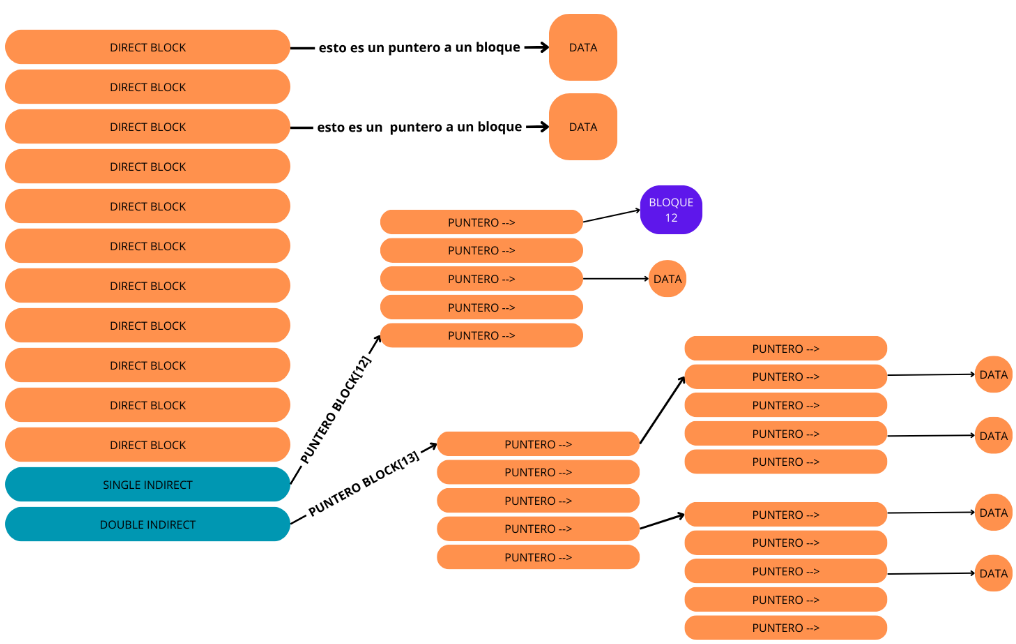
2.4 get_block_address(struct Ext2FSInode * inode, unsigned int block_number)

Recordemos la estructura del inodo. Teníamos un array llamado **block[15]**. Estos son direcciones de bloques en memoria. Se llaman LBA como ya hablamos. **no son punteros** son enteros sin signo que se intepretan como direcciones lógicas en el disco. Nos interesará lo siguiente : **obtenerla**. Para ello, como se intuye, vamos a tener que recorrer el inodo en busca de ella. Hay ciertas situaciones a las que tendremos que estar atentos :

a) Si el número de bloque se encuentra en los **bloques directos** entonces simplemente devolvemos inode → block[block_number].

b) Si el número de bloque excede el ítem anterior, tenemos que buscar en el **bloque de indirección simple**. En este bloque vamos a encontrar muchas LBAs, cuántas ? cuál es el número de bloque más grande que puede ser **contenida** por este bloque de indirección simple ? Estas son las preguntas adecuadas para poder encontrar la LBA del bloque que nos piden.

c) Si el número de bloque excede el indirecto simple, hay que moverse al **indirecto doble**, esta estructura se vuelve más pesadita de leer y todo. Hay que buscar en una LBA que tiene LBAs de bloques que tienen otras LBAs que esas últimas pueden ser las que apuntan al nro de bloque que nos interesa.



Solución:

```
unsigned int Ext2FS::get_block_address(struct Ext2FSInode * inode, unsigned int block_number)
{
    if (block_number < 12)
    {
        return inode->block[block_number];
    }
    else
    {
        unsigned int res;
        unsigned int limite_indirecto_simple = _superblock->log_block_size / sizeof(unsigned int);

        if (block_number < limite_indirecto_simple)
        {
            unsigned char* lectura = (unsigned char*) malloc(1024 << _superblock->log_block_size);
            unsigned int dire_bloque_indirectos_simple = inode->block[12];
            read_block(dire_bloque_indirectos_simple, lectura);
            unsigned int dire_bloque = block_number - 12;
            res = ((unsigned int*)lectura)[dire_bloque];
            delete[] lectura;
        }
        else
        {
            unsigned int dire_bloque_indirectos_doble = inode->block[13];
            unsigned char* lectura = (unsigned char*) malloc(1024 << _superblock->log_block_size);
            read_block(dire_bloque_indirectos_doble, lectura);
            unsigned int primero = block_number - limite_indirecto_simple;
            unsigned int dire_bloque_apuntado_primer_puntero = ((unsigned int*) lectura)[primero];
            read_block(dire_bloque_apuntado_primer_puntero, lectura);
            unsigned int offset = primero % _superblock->inode_size;
            res = ((unsigned int*)lectura)[offset];
            delete[] lectura;
        }
        return res;
    }
}
```

2.5 get_file_inode_from_dir_inode(struct Ext2FSInode * from, const char * filename)

Ya hablamos sobre los **directorios** y cómo se representan. Lo que nos piden es básicamente iterar sobre toda la info que tiene un inodo, llamado **from**, y dentro de él vamos a encontrar **DirEntries**. Como ya hicimos toda la movida de sacar la dire en el ejercicio anterior, no nos preocuparemos por la indirección. Sólo, traer un bloque lleno de DirEntries, chequear si el nombre coincide con el **filename** que nos pasan, sino continue;.
**Aclaraciones : cuando hablamos de FS y cómo el módulo maneja las limitaciones, mencionamos los nombres, lo que no dijimos fue : hay límite de tamaño del nombre ? Eso influye en el tamaño de un DirEntry a otro a la hora de recorrerlos. Puede suceder que comience en un bloque y termine el el siguiente ? Sí y, por ese motivo hay que tener 2 bloques de lectura, porque no sabemos a priori qué datos caen el uno y qué datos caen en otro, por tanto si hacemos DirEntry-¿record_length tal vez ese dato está en el siguiente bloque y se rompe todo. La estrategia es :

a) Primero necesitamos saber de qué tamaño son los bloques.

b) Como dijimos que tenemos que tener 2 bloques de datos por si una dir entry arranca en un bloque y termina en el consecutivo, pedimos memoria, dos blocksize y mantenemos referencia al punto en donde comienza el segundo bloque.

c) Arrancamos leyendo por el bloque 0 con el offset en 0.

d) Necesitamos la direcciones (LBAs) de los bloques a leer. Usamos la función anterior para ello.

e) Hacemos las lecturas

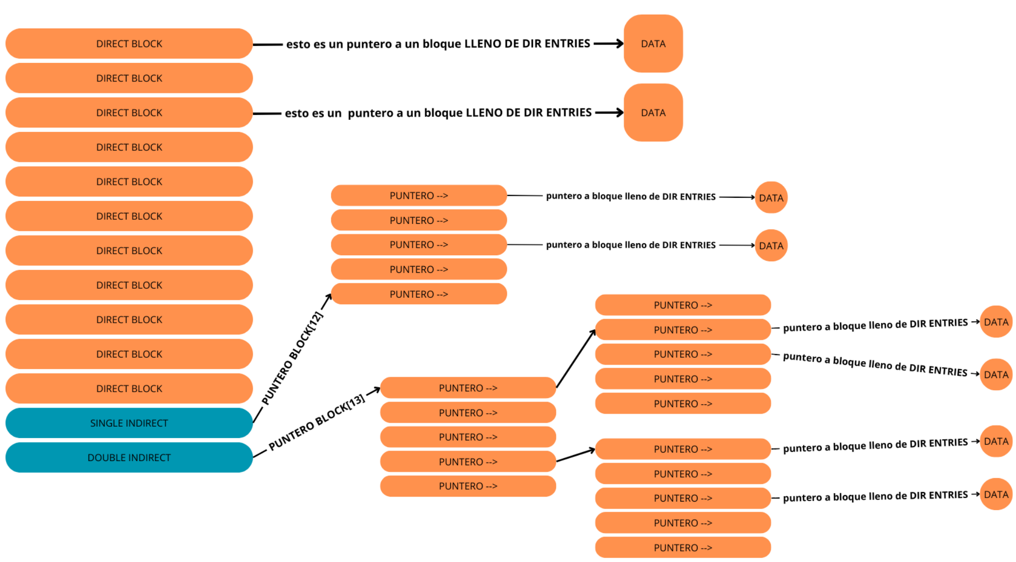
f) Ingresamos al bucle : tenemos que ver si ya terminamos de leer todas las entradas del un bloque. Para ello, miramos si el offset se pasó del tamaño de bloque.

g) Si se pasó, hay que traer el siguiente bloque, acomodar el offset y la dinámica es la misma, capturamos las LBAs y leemos.

h) Si no se pasó, tenemos que capturar cierta dir entry y hacer la lógica de la comparación. Chequeamos si el filename coincide, en caso de coincidir, cargamos el inodo. Caso contrario, continuamos buscando.

Solución:

```
struct Ext2FSInode * Ext2FS::get_file_inode_from_dir_inode(struct Ext2FSInode * from, const char * filename)
{
	if(from == NULL)
		from = load_inode(EXT2_RDIR_INODE_NUMBER);
	//std::cerr << *from << std::endl;
	assert(INODE_ISDIR(from));

	unsigned int block_size = 1024 << _superblock->log_block_size;

	unsigned char * block_buf = (unsigned char *) malloc(2 * block_size);
	unsigned char * block_sig = (unsigned char *) block_buf + block_size;

	unsigned int block_number = 0;
	unsigned int offset = 0;

	unsigned int block_address = get_block_address(from, block_number);
	unsigned int block_address2 = get_block_address(from, block_number+1);

	read_block(block_address, block_buf);
	read_block(block_address2, block_sig);

	while (true) {
		if(offset > block_size) {
			block_number++;

			offset -= block_size;

			block_address = get_block_address(from, block_number);
			block_address2 = get_block_address(from, block_number+1);

			read_block(block_address, block_buf);
			read_block(block_address2, block_sig);
		}

		Ext2FSDirEntry *entrada_actual = (Ext2FSDirEntry*) (block_buf + offset);
		int size_name = entrada_actual->name_length;

		char* filename_inode = entrada_actual->name;

		if(size_name == strlen(filename) && !strncmp(filename, filename_inode, size))
		{
		    unsigned int inodo_actual = entrada_actual->inode;
				free(block_buf);
		    return load_inode(inodo_actual);
		}

		offset += entrada_actual->record_length;
	}
	return NULL;
}
```



2.6 inode_for_path(const char * path)

La idea es, yo te paso **/home/user/documentos/archivo.txt** con la función **strtok(mypath, "/");** separamos todos los directorios y **retornamos el inodo**. Mientras tengamos directorios para recorrer, ejecutamos la función anterior get_file_inode_from_dir_inode(inodo_anterior, filename que nos interesa encontrar en este), que nos permitirá actualizar el inodo para buscar el filename del siguiente y así hasta dar con el inodo del archivo.txt.

```
// Estructura Ext2FSInode * Ext2FS::inode_for_path
struct Ext2FSInode * Ext2FS::inode_for_path(const char * path)
{
    char * mypath = new char[strlen(path)+1];
    mypath[strlen(path)] = '\0';
    strncpy(mypath, path, strlen(path));

    char * pathtok = strtok(mypath, PATH_DELIM);
    struct Ext2FSInode * inode = NULL;

    while(pathtok != NULL)
    {
        struct Ext2FSInode * prev_inode = inode;
        // std::cerr << "pathtok: " << pathtok << std::endl;
        inode = get_file_inode_from_dir_inode(prev_inode, pathtok);
        pathtok = strtok(NULL, PATH_DELIM);

        delete prev_inode;
    }

    delete[] mypath;
    return inode;
}
```

3 Ejercicio tipo parcial : recorrer en EXT2 y en FAT

```c
int my_copy_ext2(char* src_path, char* dst_path) {
    // Obtenemos los inodos para los archivos src y dst identificados por su path.
    struct Ext2FSInode* src_inode = inode_for_path(src_path);
    struct Ext2FSInode* dst_inode = inode_for_path(dst_path);

    // Creamos un buffer donde vamos a cargar los bloques que vamos copiamos.
    unsigned char* buffer = (unsigned char*) malloc(BLOCK_SIZE);

    // Calculamos el total de bloques a copiar a partir del atributo size del
    // archivo fuente (src). Notar que redondeamos hacia arriba con ceil().
    int block_count = ceil(src_inode->size / BLOCK_SIZE);

    // Copiamos bloque a bloque de src a dst.
    for (int block_number = 0; block_number < block_count; block_number++) {
        // Obtenemos la direcci n (LBA) del bloque a copiar de src.
        unsigned int src_block_address = get_block_address(src_inode, block_number);

        // Leemos el bloque de src y lo cargamos en el buffer.
        read_block(src_block_address, buffer);

        // Pedimos un bloque libre y lo asignamos al inodo de dst con el
        // n mero de bloque correspondiente.
        unsigned int dst_block_address = get_free_block_address(dst_inode);
        add_block_address_to_inode(dst_inode, dst_block_address, block_number);

        // Escribimos el buffer en el bloque asignado a dst.
        write_block(dst_block_address, buffer);
    }

    free(buffer);
    return 0;
}

int my_copy_fat(char* src_path, char* dst_path) {
    // Obtenemos las direcciones (LBA) del primer bloque de cada archivo.
    unsigned int src_block_address = get_init_block_for_path(src_path);
    unsigned int dst_block_address = get_init_block_for_path(dst_path);

    // Creamos un buffer donde vamos a cargar los bloques que vamos copiamos.
    unsigned char* buffer = (unsigned char*) malloc(BLOCK_SIZE);

    // A priori no sabemos cu ntos bloques vamos a copiar.
    while (src_block_address != EOF) {
        // Leemos el bloque de src y lo escribimos en dst.
        read_block(src_block_address, buffer);
        write_block(dst_block_address, buffer);

        // Navegamos la tabla de FAT del archivo src. En src_block_address ya
        // tenemos el pr ximo bloque de datos.
        src_block_address = FAT[src_block_address];
        if (src_block_address == EOF) {
            // Si llegamos al EOF, marcamos en la tabla de dst el EOF y terminamos.
            FAT[dst_block_address] = EOF;
        } else {
            // Caso contrario, pedimos un nuevo bloque libre para dst y lo
            // configuramos como el siguiente bloque de datos de dst para la
            // pr xima iteraci n del while.
            next_dst_block_address = get_free_block_address();
            FAT[dst_block_address] = next_dst_block_address;
            dst_block_address = next_dst_block_address;
        }
    }

    free(buffer);
    return 0;
}

// Versi n alternativa con while(true).
int my_copy_fat_while_true(char* src, char* dst) {
    uint src_block_addr = get_init_block_for_path(src);
    uint dst_block_addr = get_init_block_for_path(dst);

    unsigned char* buffer = (unsigned char*) malloc(BLOCK_SIZE);
```

```
74
75      while (true) {
76          read_block(src_block_addr, buffer);
77          write_block(dst_block_addr, buffer);
78
79          if (FAT[src_block_addr] == EOF) {
80              FAT[dst_block_addr] = EOF;
81              break;
82          } else {
83              src_block_addr = FAT[src_block_addr];
84              next_dst_block_addr = get_free_block_address();
85              FAT[dst_block_addr] = next_dst_block_addr;
86              dst_block_addr = next_dst_block_addr;
87          }
88      }
89
90      free(buffer);
91      return 0;
92  }
```