



**Universidad Nacional Autónoma de México**  
**Facultad de Ingeniería**  
**Ingeniería en Computación**



**Alumnos**

Julián Bolaños Guerrero  
Juan Pablo Zurita Cámara

**Materia**

Sistemas Operativos

**Grupo 6**

**Proyecto**

(Micro) sistema de archivos multihilos

**Fecha de entrega:** 19 de mayo del 2024

## OBJETIVO DEL PROYECTO

- Desarrollar un programa que pueda obtener, crear y modificar información de un “supuesto” microsistema de archivos de la Facultad de Ingeniería (FiUnamFS).
- Dividir la lógica del programa en componentes concurrentes.

## PRIMER ACERCAMIENTO

- Debido a la naturaleza del proyecto, se simula el *diskette* donde cabría el sistema de archivos con un archivo de longitud fija de 1440 Kilobytes.

$$1 \text{ [KB]} = 1024 \text{ bytes}$$

$$1440 \text{ [KB]} = 1440 \times 1024 = 1\,474\,560 \text{ bytes}$$

Esto nos indica que el archivo con el que se va a trabajar es de 1 474 560 bytes.

- La superficie del disco se divide en sectores de 512 bytes. Cada *cluster* mide cuatro sectores.

$$\frac{1\,474\,560 \text{ bytes}}{512 \left[ \frac{\text{byte}}{\text{sector}} \right]} = 2\,880 \text{ sectores}$$

$$\frac{2\,880 \text{ sectores}}{4 \left[ \frac{\text{cluster}}{\text{sector}} \right]} = 720 \text{ clusters}$$

Esto nos indica que el archivo está dividido por 720 clusters.

$$\frac{1\,474\,560 \text{ bytes}}{720 \text{ clusters}} = 2048 \left[ \frac{\text{bytes}}{\text{cluster}} \right]$$

Esto nos indica que cada cluster tiene una longitud de 2048 bytes.

- No se maneja *tabla de particiones*, sino que se hospeda directamente un *volumen* en la totalidad de su espacio.
- Se maneja únicamente un directorio plano, no se consideran subdirectorios.
  - El sistema de archivos es de asignación contigua. Toda la información de los archivos está en el directorio.
  - Después del directorio, todo el espacio restante es espacio de datos.
  - El directorio está ubicado en los clusters 1 a 4. Cada entrada del directorio mide 64 bytes.

$$4 [\text{cluster}] \times 2048 \left[ \frac{\text{bytes}}{\text{cluster}} \right] = 8\,192 [\text{bytes}]$$

$$\frac{8\,192 [\text{bytes}]}{64 \left[ \frac{\text{bytes}}{\text{entradaDirec}} \right]} = 128 \text{ entradas de Directorio}$$

## SOLUCIÓN

### Validación del archivo a manipular.

El archivo que se piensa utilizar tiene una estructura característica, es por ello, el archivo con el que nuestro programa trabaje debe de cumplir principalmente con estas dos especificaciones:

- Los primeros 7 bytes del archivo a manipular, [0,8), deben de formar una cadena igual **FiUnamFS**.
- Los bytes [10,14) deben de formar una cadena igual a **24-2**.

### Listar el contenido del directorio del sistema de archivos.

El directorio está ubicado en los clústers 1 a 4 del micro-sistema.

Los bytes que debemos de leer del archivo van del 2048 (inicio del cluster 1) hasta el 10239 (final del cluster 4). Lo que se almacena en estos bytes son entradas de directorio, es decir, los metadatos de los archivos almacenados en el microsistema.

Por la estructura que tiene el microsistema sabemos que el primer byte de cada entrada de directorio nos indica si esa entrada está vacía o no. Cuando el byte representa el carácter '-' no está vacía, pero cuando representa el carácter '/' se encuentra vacía.

Esto facilita un poco la lectura del directorio puesto a que en lugar de leer en su totalidad los 8192 bytes que conforman al directorio, podemos leer únicamente el primer byte de cada entrada de directorio y checar si la entrada está vacía. Y solo en el caso de no estar vacía hacemos la lectura de los demás bytes de acuerdo con las especificaciones del microsistema.

### Copiar uno de los archivos de dentro de FiUnamFS hacia tu sistema

De parte del sistema\computadora a donde se va a copiar el archivo lo que se necesita es una dirección/ruta de memoria válida a donde se va a almacenar el archivo copiado.

De parte del microsistema se necesita conocer el espacio de memoria donde está almacenado el contenido del archivo que se quiere copiar para leerlo y pegarlo en el archivo de la computadora.

¿Cómo encontrar el contenido de un archivo en el microsistema?

Por las especificaciones del microsistema sabemos que algunos de los metadatos que se guardan de cada archivo son su clúster inicial y su tamaño en bytes. Con esto fácilmente podemos saber que, para un archivo existente en el microsistema, su contenido se encuentra en los bytes contenidos en  $[\text{cluster\_inicial} * 2048, \text{cluster\_inicial} * 2048 + \text{tamaño})$ .

### **Copiar un archivo de tu computadora hacia tu FiUnamFS**

Primero que nada, el archivo que se quiere copiar de nuestra computadora a FiUnamFS debe de existir.

Debemos de obtener el tamaño del archivo a copiar para buscar si existe un espacio disponible en el microsistema.

¿Cómo podemos identificar si existe el espacio necesario para almacenar nuestro archivo?

Esto se logra conociendo el número de clústers libres restantes.

¿Cómo podemos saber si un clúster está ocupado?

Supongamos que tenemos un archivo “a.txt” almacenado en el microsistema con los siguientes metadatos:

- Cluster\_inicial = 45
- Tamaño = 4659 bytes.

Con esta información sabríamos que el contenido de este archivo se encuentra en los siguientes bytes [92160, 96819).

Un archivo “b.txt” con los siguientes metadatos:

- Cluster\_inicial = 47
- Tamaño = 2345 bytes

Supuestamente su contenido estaría contenido en los bytes [96256, 98601). No obstante, no podría existir debido a que parte de su contenido estaría dentro de una región que contiene información de otro archivo.

Si bien es cierto que el tamaño que tiene el archivo “a.txt” se almacena en aproximadamente 2.27 clúster debemos de decir que este archivo ocupa 3

clúster, haciendo que los clústers 45, 46 y 47 queden fuera de cualquier “direccionamiento”.

Otro punto importante es que, si bien puede haber espacio en el microsistema para copiar el archivo, debemos de buscar que haya entradas de directorio disponibles para almacenar su información.

### Eliminar un archivo de FiUnamFS

Para eliminar un archivo en el microsistema es tan fácil como eliminar su correspondiente entrada de directorio. Para borrar una entrada de directorio es suficiente con poner el carácter ‘/’ en su primer byte.

Los bytes que se utilizaron para guardar su contenido no son necesario tocarlos puesto a que sencillamente se sobre escribir sobre ellos.

## ENTORNO Y DEPENDENCIA

- **Lenguaje de programación utilizado: Python3.**

Se decidió utilizar este lenguaje de programación debido a su versatilidad y facilidad para prueba con otros usuarios.

- **Módulos usados**

Los módulos utilizados ya vienen implícitos en la descarga del intérprete de Python.

```
import os
import math
import struct
import threading
from datetime import datetime
```

- **Consideraciones**

Se crearon dos archivos **main.py** y **models.py**. El archivo **main.py** es el archivo que se debe de ejecutar, sin embargo, el archivo **models.py** debe de estar guardado en la misma carpeta y en el mismo nivel jerárquico que **main.py** para que el programa se pueda ejecutar de manera correcta.

Para que el programa funcione con el supuesto archivo de FiUnamFS se debe de editar en el archivo **main.py** la ruta donde se encuentra dicho archivo. Por ejemplo:

```
directory_path = 'C:/Users/juanz/Downloads/P4/fiunamfs.img'
try:
    if system_validation(directory_path):
        fiunamfs = FiUnamFS(path = directory_path, directory_entry_size = 64)
```

En este caso estoy diciendo que el archivo con el que debe de trabajar el programa se encuentra en C:/Users/juanz/Downloads/P4/fiunamfs.img. Esto se encuentra en la última parte del archivo **main.py** en la línea 160.

## IMPLEMENTACIÓN

Se crearon dos modelos (clases) en las cuales se abstrae tanto el directorio FiUnamFS como las entradas de directorio que contienen el nombre del archivo, tamaño, clúster inicial, fecha de creación y fecha de última modificación.

Las clases creadas son FiUnamFS y File. La primer clase FiUnamFS contiene la mayoría de métodos a ejecutar para satisfacer los procesos o actividades solicitadas, en esta clase encontraremos los métodos: *showDetails()*, *getFiles()*, *copyFromSystem()* y *deleteFile()* como métodos principales en la clase, sin embargo, si se observa el código fuente de estas clase en 'models.py' notará que existen más métodos que componen la clase, estos métodos son privados (por convención en Python se emplea el carácter '\_' para indicar que una variable o método es privado) y complementan el funcionamiento de los métodos principales ya mencionados.

Por el otro lado la clase File se creo con el fin de facilitar el desarrollo del programa ya que esta clase nos facilito manejar los datos de cada archivo: nombre, tamaño de archivo, clúster inicial, fecha de creación, fecha de última modificación, etc. Su única función es esta la de administrar esta información. Las instancias File se crean cada vez que el método *getFiles()* de la clase FiUnamFS se manda a llamar o ejecutar. De este modo podemos decir que las instancias de File que existan en el programa serán únicamente las existentes en el directorio en ese momento exacto, si un archivo es eliminado del directorio se ejecuta nuevamente *getFiles()* y la instancia del archivo eliminado pues evidentemente ya no se creara.

A continuación, se explica con más detalle el funcionamiento de cada método:

- *showDetails()*

```
def showDetails(self) → None:
    print(f'Nombre Directorio: {self.system_name}\n' +
          f'Version: {self.version}\n' +
          f'Etiqueta de Volumen: {self.volumen_label}\n' +
          f'Tamaño de Cluster: {self.cluster_size}\n' +
          f'Num de Clusters: {self.num_clusters}\n' +
          f'Num Total de Clusters: {self.num_total_clusters}')
    )
```

Este método tiene como propósito imprimir en pantalla los datos del directorio tales como nombre de directorio, versión, etiqueta de volumen, el tamaño que tiene un clúster en el directorio, el número de clústeres que mide el directorio y el número de clústeres que mide la unidad completa. Estos valores se obtienen directamente del constructor de la clase como se muestra a continuación.

```
class FiUnamFS():

    def __init__(self, path:str, directory_entry_size:int):
        self.path = path
        self.system_name = self._readStrFromFS(0,8)
        self.version = self._readStrFromFS(10, 4)
        self.volumen_label = self._readStrFromFS(20, 15)
        self.cluster_size = self._readIntFromFS(40, 4)
        self.num_clusters = self._readIntFromFS(45, 4)
        self.num_total_clusters = self._readIntFromFS(50, 4)
        self.directory_entry_size = directory_entry_size

    def __str__(self) → str:
        return self.system_name
```

- **Funciones de lectura de la unidad ‘fiunamfs.img’**

Como se observa cada variable en la clase obtiene sus valores por medio de los métodos privados `_readStrFromFS()` o `_readIntFromFS()`, ambas funciones realizan el mismo proceso que es abrir la unidad ‘fiunamfs.img’ y leer su contenido desde un punto específico hasta otro punto dado, en este caso sería desde un byte determinado hasta haber leído n bytes definidos, la diferencia entre ellas es que una lee bytes que representan un número y la otra bytes que representan un string.

```
def _readIntFromFS(self, start:int, reading_size:int):

    with open(self.path, 'rb') as fs:
        fs.seek(start)
        content = fs.read(reading_size)
        c, = struct.unpack('<I', content)
        return c

def _readStrFromFS(self, start:int, reading_size:int):

    with open(self.path, 'rb') as fs:
        fs.seek(start)
        content = fs.read(reading_size)
        return content.decode('ascii').strip()
```

Como se observa ambos métodos reciben dos argumentos, *start* indica el byte desde donde se comenzará a hacer la lectura, mientras que *reading\_size* es la cantidad de bytes que se van a leer, nótese que ambos deben ser enteros. Finalmente, las funciones retornan el contenido leído como una cadena de caracteres o un entero según sea el caso.

- *getFiles()*

Esta función manda a llamar a 8 hilos para dividirse la tarea de leer las 128 entradas de directorio. Cada uno de estos hilos ejecuta de manera concurrente la función *\_insertFiles()* que tienen como objetivo crear instancia de tipo File de acuerdo con cada entrada de directorio NO VACÍA que se encuentre y agregarlo a una lista compartida por todos estos hilos que contenga los objetos File de los archivos encontrados.

```
def getFiles(self):
    # Este número indica la cantidad de hilos entre la que se va a repartir la tarea
    # Cada hilo pretende leer una parte del directorio para obtener la info de los archivos
    num_td = [8]

    num_files = (self.cluster_size * self.num_clusters) // self.directory_entry_size

    # Para que funcione bien la función, el número de hilos debe de ser divisor
    # del número de archivos. Debido a las especificaciones del proyecto sabemos
    # que 8 es un número válido.
    if num_files % num_td[0] == 0:
        files = []
        displacement = num_files // num_td[0]

        mutex1 = threading.Semaphore(1)
        barrier1 = threading.Semaphore(0)
        mutex2 = threading.Semaphore(1)

        for i in range(0, num_files, displacement):
            threading.Thread(target = self._insertFiles, args = [i, displacement, mutex1, barrier1, num_td, files],
                             daemon=True).start()

            barrier1.acquire()
            barrier1.release()

        return files
    else:
        return []
```

Método *getFiles()*



```

def _insertFiles(self, pointer:int, displacement:int, mutex:object, barrier:object, num_td:object, data_storage:obje

    for i in range(pointer, pointer + displacement):
        start = self.cluster_size + (i * self.directory_entry_size)
        file_name = self._readStrFromFS(start, 15)

        if '-' in file_name:
            # Debido a que data_storage es una variable que puede ser editada por muchos hilos
            # es necesaria asegurar su región crítica
            mutex2.acquire()
            data_storage.append(
                File(
                    name = file_name[1:].strip(),
                    size = self._readIntFromFS(start + 16, 4),
                    initial_cluster = self._readIntFromFS(start + 20, 4),
                    creation_date = self._readStrFromFS(start + 24, 13),
                    update_date = self._readStrFromFS(start + 38, 13),
                    path_directory = self.path
                )
            )
            mutex2.release()

        mutex.acquire()
        num_td[0] = num_td[0] - 1
        if num_td[0] == 0:
            barrier.release()
        mutex.release()

```

### Método `_insertFiles()`

- **`copyFromSystem()`**

Este método es el principal encargado de copiar un archivo de nuestra computadora hacia el directorio FiUnamFS.

```

def copyFromSystem(self, path:str) → int:

    # Tamaño del archivo a copiar
    new_file_size = os.path.getsize(path)

    # Obtenemos el cluster inicial de donde se comenzará a
    # almacenar el contenido del archivo a copiar.
    initial_cluster = self._searchSpace(new_file_size)

    if initial_cluster != None:
        # Movemos nuestro 'apuntador' a donde inicia el cluster obtenido
        # (cluster inicial * tamaño de cluster)
        start = self.cluster_size * initial_cluster
        content = self._getContentFile(path)

        try:
            with open(self.path, 'rb+') as new_file:
                new_file.seek(start)
                new_file.write(content)

            # Esta función devuelve el status de la copia al directorio
            return self._insertIntoDirectory(path, initial_cluster)
        except:
            return 5
    return 6

```

La forma en que funciona es que recibe la ubicación del archivo a copiar y obtiene su tamaño para comenzar a buscar espacio disponible dentro en la unidad, el método encargado de buscar este espacio es `_searchSpace()`, si este método encuentra una secuencia de clústeres libres y consecutivos para almacenar el contenido de todo el archivo retorna el clúster inicial de donde se debe comenzar a inserta dicho contenido, si no encuentra espacio retorna `None`.

Posteriormente si existe espacio disponible obtenemos su contenido mediante el método `_getContentFile()`, este método abre el archivo a copiar en modo de lectura binaria y lee todo el archivo y nos devuelve esa información leída. Finalmente, este contenido se inserta desde el byte que se obtiene del producto del clúster inicial por el tamaño de un clúster. Si todo este proceso es exitoso entonces insertaremos en el directorio (clústeres 1-4) los datos del archivo (nombre, tamaño, etc.) esto lo hacemos con el método `_insertIntoDirectory()`. Este método simplemente obtiene los datos necesarios con ayuda de la librería `os` de Python y busca dentro del directorio FiUnamFS cualquier entrada de directorio que contenga el carácter `'/'` en el nombre, esto nos indica que el espacio o la posición está disponible e inserta los datos en ese espacio encontrado de 64 bytes.

- `deleteFile()`

Este método se encarga de eliminar un archivo del directorio FiUnamFS.

```
def deleteFile(self, file_name:str) → int:

    # Validamos que el archivo exista en 'FiUnamFS'.
    files_in_directory = self.GetFiles()
    for file in files_in_directory:
        if file.name == file_name:

            num_files = (self.cluster_size * self.num_clusters) // self.directory_entry_size
            start = self.cluster_size

            # Eliminamos la entrada de cada archivo
            for i in range(num_files):
                file_to_delete = self._readStrFromFS(start + (i * self.directory_entry_size), 15)

                if file_name == file_to_delete[1:].strip():

                    try:
                        with open(self.path, 'rb+') as _FiUnamFS:
                            _FiUnamFS.seek(start + (i * self.directory_entry_size))
                            _FiUnamFS.write('#####'.encode('utf-8'))

                        return 1
                    except:
                        return 2

            # Si el archivo a eliminar no existe en el directorio
            return 3
```

Recibe el nombre del archivo que se quiere borrar y hace una validación para saber si ese archivo existe mediante su nombre, sino existe simplemente nos retorna un 3 entero, esto nos indicará que no existe. Por otro lado si existe, sustituimos el nombre de ese archivo por la cadena `'/#####'` en la entrada de directorio que corresponde a dicho archivo (esta misma función obtiene el dato) de este modo la siguiente vez que se quiera obtener los archivos con el método *getFiles()*, este archivo que se eliminó ya no aparecerá.

- ***copyToSystem()***

Este método es el principal encargado de copiar cualquier archivo existente en el directorio FiUnamFS hacia nuestra computadora.

```
def copyToSystem(self, path:str) → bool:

    if not os.path.exists(path + f'/{self.name}'):
        content = self._getFileContent()

        try:
            with open(path + f'/{self.name}', 'wb') as new_file:
                new_file.write(content)
            return True
        except:
            return False
```

Este método no existe en la clase FiUnamFS sino en la clase File, por tales motivos es propia de cada archivo. Su funcionamiento es simple, recibe una ubicación de donde se copiará el archivo, si el archivo ya existe en ese lugar no se realiza la copia, sino existe entonces traemos el contenido desde la unidad fiunamfs.img con *\_getFileContent()* (recordemos que cada instancia de tipo File se realizó por comodidad para el manejo de sus datos como nombre, tamaño, clúster inicial, etc. Ya que en el programa no almacenamos otro tipo de información), una vez que tenemos el contenido del archivo abrimos el nuevo archivo en nuestra computadora y pegamos todo el contenido mediante el modo de escritura `'wb'` que nos permite hacer Python.

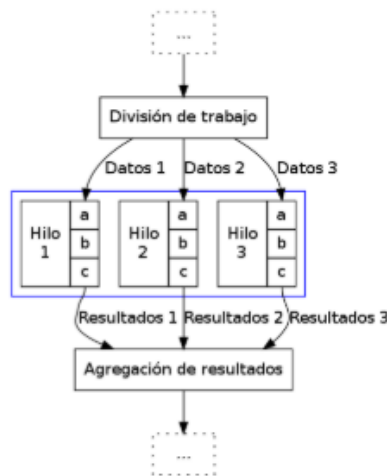
## DESCRIPCIÓN DE SINCRONIZACIÓN UTILIZADA

Para la parte de implementación de hilos en nuestro programa, nosotros hicimos uso de los patrones de trabajo con hilos de **equipo de trabajo** y **línea de ensamblado**.

### ▪ Equipo de trabajo mezcla con jefe/trabajador

Este patrón consiste básicamente en crear hilos idénticos para realizar la misma tarea sobre diferentes datos. Algunas características son:

- Los resultados generados por los diferentes hilos son agregados o totalizados al terminar su procesamiento.
- Los hilos son sincronizados para continuar con la ejecución lineal



### ¿Dónde se implementó?

Una operación muy recurrente en nuestro programa es la de obtener información de los archivos del microsistema. Para obtener dicha información es necesario checar las 128 entradas de directorio que existen. Esencialmente para cada entrada de directorio se lleva a cabo el mismo procedimiento, es por ello por lo que se nos ocurrió dividir este “proceso” en varios hilos.

NOTA: En el código se puede encontrar esta parte en la función de la clase FiUnamFS “\_getFiles”.

En el código se decidió utilizar 8 hilos, por lo tanto, cada uno de estos hilos va a checar 16 entradas de directorio diferentes. Cuando uno de estos hilos encuentra una entrada de directorio NO VACÍA lee su información correspondiente para crear objetos de tipo File con la información correspondiente a dicho archivo al que se refiere dicha entrada de directorio y lo almacena en una lista que es compartida por

todos los demás hilos. Como se puede leer, si se cumple con las características de un **equipo de trabajo**.

Debido a que se trabaja con una variable compartida por cada hilo, se utiliza un mutex. Asimismo, como nos interesa los resultados generados por cada hilo tras su ejecución completa, utilizamos el mecanismo de sincronización de **barrera** para que el hilo “jefe” que manda a realizar la división de trabajo continúe con su ejecución hasta que cada uno de estos hilos “trabajadores” hayan terminado.

- **Línea de trabajo mezcla jefe/trabajador**

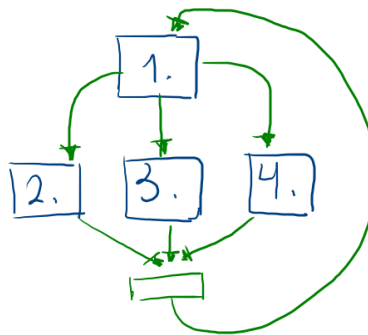


Se nos ocurrió utilizar este patrón de trabajo debido a que hay 4 cosas esenciales que debe de hacer nuestro programa:

1. Desplegar el menú.
2. Listar los contenidos del directorio.
3. Copiar uno de los archivos dentro del FiUnamFS hacia tu sistema.
4. Copiar un archivo de tu computadora hacia tu FiUnamFS.

Entonces la idea es que exista un hilo para cada una de estas tareas.

Decimos que es una mezcla con jefe/trabajador debido a que el hilo que maneje la tarea de menú va a ser el “jefe” pues es el que le va a dar paso a que los demás hilos se puedan ejecutar. Además, cuando los demás hilos terminan le devuelven el control al hilo “jefe”.



## EJEMPLOS DE USO

1. Desplegar menú.

```
PS C:\Users\juanz\Downloads\P4> python main.py

Bienvenido a FiUnamFS

(1) - Listar los contenidos del directorio.
(2) - Copiar uno de los archivo dentro de FiUnamFS hacia tu sistema.
(3) - Copiar un archivo de tu computadora hacia FiUnamFS.
(4) - Eliminar un archivo del FiUnamFS.
(5) - Salir.
Opcion -> 1
```

2. Listar contenido de directorio.

```
PS C:\Users\juanz\Downloads\P4> python main.py

Bienvenido a FiUnamFS

(1) - Listar los contenidos del directorio.
(2) - Copiar uno de los archivo dentro de FiUnamFS hacia tu sistema.
(3) - Copiar un archivo de tu computadora hacia FiUnamFS.
(4) - Eliminar un archivo del FiUnamFS.
(5) - Salir.
Opcion -> 1

|Archivos en FiUnamFS|

1. README.org    31222 Bytes    2024-05-08 13:17:5
2. logo.png     126423 Bytes   2024-05-08 13:17:5
3. mensaje.jpg   254484 Bytes   2024-05-08 13:17:5
```

3. Copiar uno de los archivos dentro del FiUnamFS hacia tu sistema.

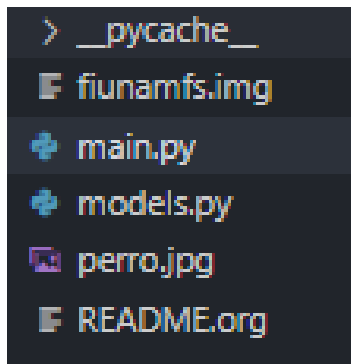
```
Bienvenido a FiUnamFS

(1) - Listar los contenidos del directorio.
(2) - Copiar uno de los archivo dentro de FiUnamFS hacia tu sistema.
(3) - Copiar un archivo de tu computadora hacia FiUnamFS.
(4) - Eliminar un archivo del FiUnamFS.
(5) - Salir.
Opcion -> 2

|Copiar archivo a mi Computadora|

Nombre de archivo a copiar: README.org
Ingresa la dirección destino: C:\Users\juanz\Downloads\P4

Se copió "README.org" en "C:\Users\juanz\Downloads\P4"
```



```
> __pycache__
fiunamfs.img
main.py
models.py
perro.jpg
README.org
```

4. Copiar un archivo de tu computadora hacia tu FiUnamFS.

```
|Copiar archivo de mi Computadora a FiUnamFS|

Ingresa la ubicacion del archivo a copiar, por ejemplo: D:/my_files/archivo_a_copiar.jpg

[* NOTA: Si no encuentra el archivo posiblemente es por que falta su extensión *]

Ubicacion: C:\Users\juanz\Downloads\P4\perro.jpg

Se copió el archivo de manera satisfactoria

Bienvenido a FiUnamFS

(1) - Listar los contenidos del directorio.
(2) - Copiar uno de los archivo dentro de FiUnamFS hacia tu sistema.
(3) - Copiar un archivo de tu computadora hacia FiUnamFS.
(4) - Eliminar un archivo del FiUnamFS.
(5) - Salir.
Opcion -> 1

|Archivos en FiUnamFS|

1. README.org    31222 Bytes    2024-05-08 13:17:5
2. perro.jpg    82850 Bytes    2024-05-19 22:18:1
3. logo.png    126423 Bytes    2024-05-08 13:17:5
4. mensaje.jpg  254484 Bytes    2024-05-08 13:17:5
```

## 5. Eliminar un archivo de FiUnamFS

### |Eliminar archivo de FiUnamFS|

Ingresa el nombre de archivo a eliminar: README.org

README.org se eliminó correctamente.

### Bienvenido a FiUnamFS

- (1) - Listar los contenidos del directorio.
- (2) - Copiar uno de los archivo dentro de FiUnamFS hacia tu sistema.
- (3) - Copiar un archivo de tu computadora hacia FiUnamFS.
- (4) - Eliminar un archivo del FiUnamFS.
- (5) - Salir.

Opcion -> 1

### |Archivos en FiUnamFS|

|                |              |                    |
|----------------|--------------|--------------------|
| 1. perro.jpg   | 82850 Bytes  | 2024-05-19 22:18:1 |
| 2. logo.png    | 126423 Bytes | 2024-05-08 13:17:5 |
| 3. mensaje.jpg | 254484 Bytes | 2024-05-08 13:17:5 |